Constant-Time Operation Transformation and Integration for Collaborative Editing

Weihai Yu
University of Tromsø, Norway
Email: weihai@cs.uit.no

Abstract—**Operational transformation (OT) is the concurrency control mechanism for collaborative editors, due to its high responsiveness to local editing operations. However, collaborative editing is still not widely practiced. One of the reasons is that operation transformation and integration are computation intensive and time consuming. The state-of-the-art time complexity is currently $O(|H|)$, where $|H|$ is the length of operation histories, which can be large and grow indefinitely. Moreover, most of the published work is limited with character operations, leading to long operation histories and impractically large number of small messages over the network. This paper presents an approach that supports string operations and constant-time operation transformation and integration. The approach is based on admissibility preservation, a correctness criterion with which the correctness of the approach can be formally proven.**

Keywords: group editor, operational transformation, concurrency control, performance, admissibility preservation

I. Introduction

A collaborative editor allows multiple users to simultaneously edit the same document from different places. It is known that concurrency control mechanisms based on mutual exclusion do not provide suitable responsiveness for collaborative editing. Over the years, operational transformation (OT) has been well established as a concurrency control mechanism commonly regarded as appropriate for collaborative editors [1]–[12].

Since the initial work of Ellis and Gibbs two decades ago [1], OT has been an active research subject and significant progress has been made. However, collaborative editing is still not widely practiced. There are several reasons for this.

- Correctness. Although the concept of OT is intuitive at its appearance, designing correct operation transformation functions is non-trivial. Counterexamples have been found for many published transformation functions. It was as late as last year that a correctness criterion was introduced that can be practically used to formally prove the correctness of OT systems [5].
- Performance. Although OT provides high responsiveness for local editing operations, the time for transforming and integrating remote operations is often dependent on $|H|$, the length of the history of editing operations. Most of the earlier work has time complexity $O(|H|^2)$. The state-of-the-art is $O(|H|)$ [9], [10]. $|H|$ is typically large and even grows indefinitely. Operation transformation and integration are therefore prohibitively computation intensive and time consuming.
- Operation granularity. In most work, transformation functions are designed for insertion and deletion of single characters. This leads to excessively large amount of small messages over the network and long operation histories. [8] and [12] are the only published work with support for string operations, of which a counterexample of [12] was found in [4].

This paper reports a novel approach that supports string operations and the algorithm for operation transformation and integration has time complexity $O(1)$ with respect to $|H|$. It is based on admissibility preservation [5], so correctness can be formally proven. Furthermore, it has some additional desirable features: the data model can be suppressed to reduce data complexity; remote operations can be partially rendered to accomodate to system load and to reduce interference with user's current editing activities.

The paper is organized as follows. Section II presents background and related work. Section III presents the approach in detail. Sections IV and V discuss its correctness and performance. Section VI concludes.

II. Background and Related Work

OT was first introduced by Ellis and Gibbs in [1]. The basic idea is as the following. A shared document is replicated at different peers. An editing operation is first executed at a local peer and then propagated to remote peers. Suppose two peers start with "012". Peer 1 inserts "a" between "0" and "1" with $ins(1, \text{"a"})$ and Peer 2 deletes "2" with $del(2)$. The states after local executions at the two peers are "0a12" and "01". Now if the two peers execute the remote operations as is, the states at these peers become "0a2" and "0a1", which are inconsistent. With OT, the remote operations are transformed to *include* the executed concurrent operations, into $ins(1, \text{"a"})$ and $del(3)$ respectively. The two peers are in consistent state "0a1" after executing the transformed operations.

There are some challenges with this basic approach. First, a remote operation can only be transformed to include a concurrent operation that is *compatible*, i.e., the two operations operated on exactly the same state. To achieve this, a peer usually has to first transpose the history of operations to make the operations compatible, and then include the effects of compatible operations. The transposition process involves the transformation of both the remote operation and operations in the local history. This whole process is usually called *operation integration*. Operation transformation and integration

COLLABORATECOM 2011, October 15-18, Orlando, United States
Copyright © 2012 ICST
DOI 10.4108/collaboratecom.2011.37

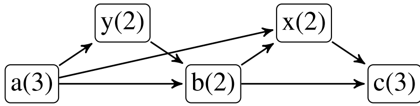


Fig. 1. A global effects-relation graph

algorithms usually have time complexities dependent on the length of the operation history.

Another challenge is that transformation functions are difficult to be made correct. Counterexamples were found for many of the published transformation functions. For instance, [2]–[4], [6], [11] reported counterexamples of earlier work. [5]–[11] are among the few that have no counterexample reported. According to [6], all these counterexamples are due to the same basic problem. Given three peers starting with "012". Peers 1, 2 and 3 issue concurrent operations $ins(2, \text{"x"})$, $del(1)$ and $ins(1, \text{"a"})$ respectively. Because "x" is inserted to the right of "1" and "a" to the left of "1", the final states at all peers must be "0ax2". The deleted character "1", called a *landmark character* in [3], determines the ordering between "a" and "x". However, because "1" is deleted at Peer 2, the two inserted characters thus tie at Peer 2. The counterexamples are due to failure to break the tie of this type in different combinations of concurrent operations.

One of the reasons that identifying counterexamples was non-trivial was that there lacked general correctness criteria with which the correctness of transformation functions can be formally proven.

Li and Li in [5] introduced a correctness criterion called admissibility preservation, with which correctness of admissibility-based transformation (ABT) functions can be formally proven. Key to this correctness criterion is the effects relation among characters and a global effects-relation graph that is only for correctness analysis purposes and need not be implemented. Fig. 1 shows an example effects-relation graph. Two characters are *effects related*, if there exists a path from one node to the other in the graph. In a node, the number in the parenthesis next to the character indicates the number of peers currently "see" the character. Suppose there are three peers. In the figure, characters "a" and "c" are visible at all peers whereas characters "y", "b" and "x" are only visible at two peers. A character is only visible at some of the peers when an operation is not yet integrated at all peers. Eventually, a character is visible either at all peers or at none.

The correctness criterion *admissibility preservation* states that the execution of any operation does not violate the effects relation that has been established in the global effects-relation graph.

[5] states that the effects relation established in the global effects-relation graph is not a total order among characters. This avoids unnecessary introduction of "artificial" ordering among characters. For example, in Fig. 1, when "x" is inserted at a peer where "b" has been deleted, no effects relation is established between "x" and "b". On the suface, it seems that only a total order on visible characters need be established. In fact, a total order among all characters will eventually be established anyway. Because deleted characters are landmarks, their effects relations are established either explicitly in the model in [6], [7], or implicitly in the integration algorithm in [5], [8]–[11]. In fact, by explicitly exploring the total effects-relation order, [10] improved the time complexity of operation transformation and integration from $O(|H|^2)$ to $O(|H|)$.

With ABT [5], landmark characters are brought into place by operation transformations. A peer maintains the operation history in the form of $H_i \cdot H_d$ where H_i and H_d consist only of insertion and deletion operations respectively. When a local operation is executed, it is first transformed to exclude the deletions in H_d. This essentially brings the landmark characters in effect. This transformation has time complexity $O(|H_d|)$. This transformed operation is then sent to remote peers to be transformed and integrated, with time complexity $O(|H_i|^2 + |H_d|)$. ABTS [8] extends ABT to support string operations. ABTU [10] arranges the operation history in total effects-relation order and improves the time complexity to $O(|H|)$. With all of ABT, ABTS and ABTU, $|H|$ grows indefinitely.

In [7], a peer keeps a view and a model. A view consists of the characters the end user currently sees. Characters in the model include landmark characters and are uniquely identified. Positions in model operations are relative to characters' identifiers and can be transformed and integrated without an operation history. [7] uses intention preservation as its correctness criterion. Intension preservation, as originally presented in [12], is not formally defined. [7] reifies it as the effects relation established at insertions, which essentially is the same as admissibility preservation.

Our approach is built on the ideas from [5], [7] and [10]. A peer consists of a view and a model. The model is an effects-relation graph, like the global effects-relation graph in [5], but the nodes in the graph follow a total effects-relation order, like a model in [7] and a history in [10]. The nodes, however, represent strings rather than characters. Effects of insertion and deletion operations are encapsulated in the graph. Our model can thus be regarded as capturing both the model in [7] and the history in [10]. Most importantly, the transformation and integration of a string operation is a single graph update, which has constant-time complexity $O(1)$ when nodes are hash indexed. Some other desirable features of the approach include: the model graph can be suppressed so that its size (in terms of number of nodes) can be reduced; the effects of remote operations can be partially rendered to the view, thus reducing both the system load and the interference with the user's current editing efforts.

III. Approach

Sub-section III-A first gives an overview of the approach. Sub-section III-B then presents the main components of a peer. The following sub-sections present in more detail the specific algorithms and procedures.

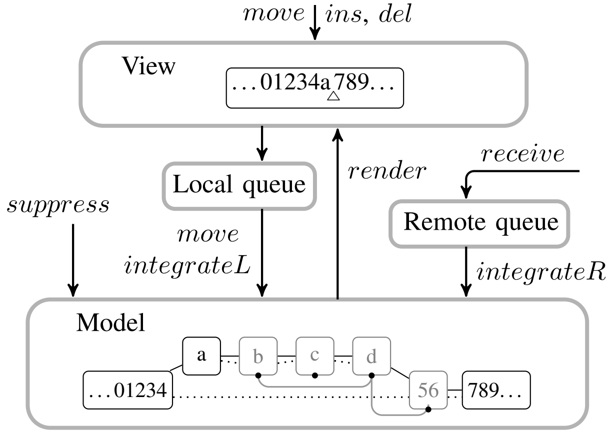


Fig. 2. View, model and operations

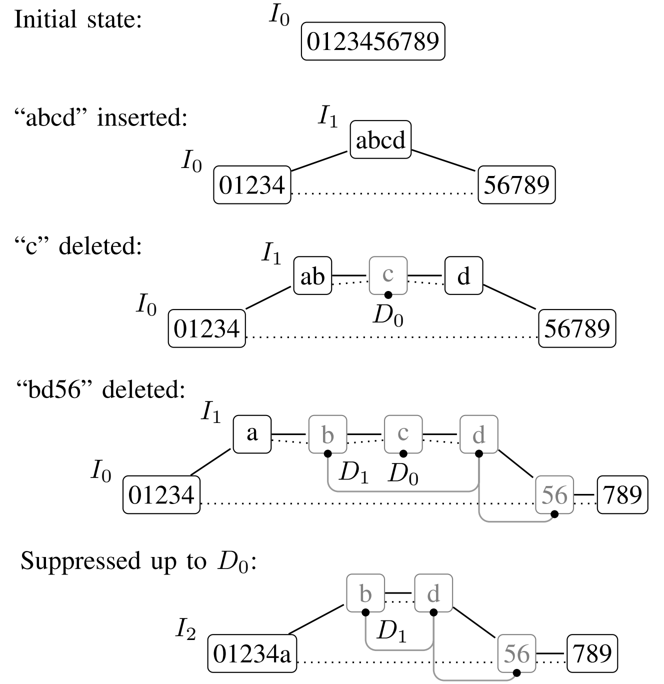


Fig. 3. Model graphs

A. Overview

A document is collaboratively edited by a number of peers at different sites. Every peer consists of a view, a model and two queues (Fig. 2). A view is a string of characters together with a current position in the string. A user at a peer can move the current position, and insert or delete sub-strings at the current position. The user's operations take immediate effect in the view and are enqueued in the local operation queue to be integrated later in the model.

A model is a graph where the nodes are sub-strings and the links of different types maintain a total-ordered effects relation on the nodes as well as the insertion and deletion operations. Initially a model has a single node with the initial string in the view. When operations are integrated, the existing nodes are subsequently split at operation boundaries, and either new nodes are inserted or existing nodes are marked as deleted. The graph thus grows while the document is being edited. Later on, the graph can be suppressed when it gets too large and complex. Fig. 3 illustrates a model graphs in different states.

Periodically, a peer runs procedure $integrate$. The operations in the local queue may be re-arranged before they are integrated in the model, such that the operations are of appropriate granularity and that the operations are integrated in some particular order, for instance from left to right, to reduce the total overhead of integration.

Procedure $integrate()$

1 $messages \leftarrow []$
2 **while** $op \leftarrow localq.dequeueOp$ **do**
3 | **if** $op = move(\delta)$ **then** $model.move(curr, pos, \delta, 0)$
4 | **else** $messages.push(model.integrateL(op))$
5 $broadcast(messages)$
6 **while** $op \leftarrow remoteq.dequeueReadyOp$ **do**
7 | $model.integrateR(op)$
8 $model.render(curr, pos, \delta_0, 0)$

A remote operation in a message is represented as a graph update. Operation transformation and integration are actually the realization of graph updates. When the nodes in the graph are hash indexed, an update has constant-time complexity. It is crucial that the integration preserves the effects-relation order established at the peer originating the operation.

When a peer renders a model, the updates of remote peers that have been integrated in the model are shown in the view. A rendering can be partial. A partial rendering is centered around the current position, so the concurrent remote operations that are most relevant to the current local editing activities are immediately visible in the view. With partial rendering, the user is not overwhelmingly distracted by the less relevant concurrent remote operations. Furthermore, the overhead of rendering can be restricted.

B. Peer, view and model

Consider a document edited by N peers. A peer is a 6-tuple $(id, v, view, model, localq, remoteq)$ consisting of an identifier, a state vector of N elements, a view, a model and two queues. The value of the i-th element of v, $v[i]$, indicates the operations of peer i that have been integrated into the peer's model. $peer.tick[i]$ increments $peer.v[i]$ by 1.

A *view* is a pair (str, pos) where str is the character string currently visible to the user and pos is the current position between two characters. For example, with view $(\text{“0123”}, 0)$, the current string is "0123" and the current position is left to character "0"; with view $(\text{“0123”}, 2)$, the current position is between characters "1" and "2".

A user may run the following operations in the view:

- $move(\delta)$ moves the current position $|\delta|$ characters. If δ is positive, the current position is moved to the right;

otherwise, it is moved to the left.

- $ins(str)$ inserts string str at the current position and the new current position is placed at the right end of str. $ins(\text{"xyz"})$ changes view from $(\text{"0123"}, 2)$ to $(\text{"01xyz23"}, 5)$.
- $del(len)$ deletes len characters right to the current position. $del(2)$ changes view from $(\text{"0123"}, 1)$ to $(\text{"03"}, 1)$.

A *model* is a triple $(nodes, curr, pos)$, where $nodes$ is the set of nodes of the model graph, $curr$ is the current node and pos is a position in $curr$. $curr$ and pos together refer to the current position in the model.

A node of the model graph is a 10-tuple $(pid, v, offset, str, dels, rendered, l, r, il, ir)$:

- pid and v, the peer identifier and state vector of the ins operation that created this node.
- $offset$, the distance from its leftmost position to the leftmost position of the original inserted string. When a node is first inserted after an ins operation, $offset$ is 0. Splitting the node at position pos leads to two nodes, with $offset$s 0 and pos respectively.
- str, the character string of the node. We refer to the length of a node as $node.len = node.str.len$.
- $dels$, a set of del elements related to deletions.
- $rendered$, $true$ if the node has been rendered to the view.
- l, r, the left and right nodes in effects relation. In the figures, the effects-relation links are illustrated with solid lines (___) connecting nodes.
- il, ir, the left and right nodes of the same ins operation. In the figures, the links between the nodes of the same ins operation are illustrated with dotted lines (......).

An *insertion* consists of the nodes chained with the il and ir links. A *deletion* consists of the nodes containing the del elements of the same del operation. $node.dels$ may contain multiple del elements. When this is the case, $node.str$ has been deleted concurrently by different peers.

A del element is a quadruple (pid, v, l, r):

- pid and v, the peer identifier and state vector of the del operation.
- l and r, the left and right nodes of the same del operation.

In the figures, a del element is depicted with a dot on the bottom edge of the node ( ). The links between the nodes of the same del operation are illustrated with lines connecting the dots ( ). A del element of a node can be obtained by its state vector with $node.del(v)$. Thus $node.del(v).l$ gives the left node of the same deletion.

Definition (visible node)**.** *A node is visible if its $dels$ is empty.*

Function $node.visible$ returns $true$ if $node$ is visible. Function $node.visibleLeft$ (or $node.visibleRight$) returns the next visible node to the left (or right) of $node$ in effects-relation order. In Fig. 3, after "bd56" was deleted, calling $visibleRight$ on node "a" returns node "789".

Locally in a peer, a node can be directly referred to via its reference. So the links l, r, il and ir in nodes, and l and r in del elements refer to other nodes with their references.

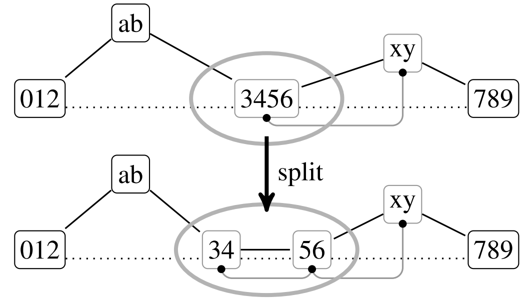


Fig. 4. Splitting node "3456"

Node references, however, are meaningless across peer boundaries. Fortunately, a node can be uniquely identified by the state vector of the $ins(str)$ operation that inserted the string str, together with the offset of the node's leftmost position in str. In Fig. 3, suppose the state vector value of $I_0 = ins(\text{"0123456789"})$ is $(0, 0, 0)$. The node with string "56" can be uniquely identified with $((0, 0, 0), 5)$. In our implementation, every node is hash-indexed in $model.nodes$ with $(v, offset)$. Therefore given $(v, offset)$, a node can be obtained in constant time at any peer.

Many model updates involve splitting of nodes. Procedure $node.split(pos)$, where $node.offset < pos < node.offset + node.len$, splits $node$ at pos and returns a pair of the new left and right nodes. Fig. 4 shows the effect of splitting a node. The procedure takes care that the new nodes are hash-indexed in $model.nodes$ and that the links are updated properly so that the insertions and deletions of the model remain unchanged.

C. Integrating a local insertion

Procedure $integrateL$ integrates a view operation at the current position of the model and returns a representation of the update to be sent to remote peers. We describe the integration of insertion and deletion operations separately. First, the integration of an insertion operation.

Procedure $integrateL(Ins)$

1 $peer.tick(peer.id)$
2 $nd \leftarrow newNode(pid : peer.id, v : peer.v, str : Ins.str)$
3 $nodes[(nd.v, 0)] \leftarrow nd$
4 **if** $curr.offset < pos < curr.offset + curr.len$ **then**
5 | $(nd_l, nd_r) \leftarrow curr.split(pos)$
6 **else if** $pos = curr.offset$ **then**
7 | $nd_l \leftarrow curr.visibleLeft$; $nd_r \leftarrow nd_l.r$
8 **else** // $pos = curr.offset + curr.len$
9 | $nd_l, nd_r \leftarrow curr, curr.r$
10 $nd_l.r, nd_r.l, nd.l, nd.r \leftarrow nd, nd, nd_l, nd_r$
11 $curr, pos \leftarrow nd, nd.offset + nd.len$
12 **return** $encode(INS, peer.id, peer.v, nd, [nd_l, nd_r])$

The peer advances its state vector (Line 1), creates a new node for the insertion (Line 2), and hash-indexes the node in

$nodes$ (Line 3).

The new node will be inserted between two nodes nd_l and nd_r (Line 10). To determine nd_l and nd_r, there are three cases to consider: the current position pos is in the middle of the current node $curr$ (Line 4); pos is at the left end of $curr$ (Line 6); or pos is at the right end of $curr$ (Line 8). In the first case, $curr$ is split at pos (Line 5) and the new node is inserted in between.

In the other two cases, the new node is inserted between two existing nodes. When there are invisible nodes involved, Policy DI is applied.

Policy (DI)**.** *When a new node is inserted between two visible nodes in the model graph, all existing invisible nodes between the two nodes are placed to the right of the new node.*

With all peers agreeing on the same Policy DI, conflicts among concurrent insertions at the same view position are not "accidentally" resolved by the deleted characters.

Other work introduced a similar policy that is enforced before a local operation is sent to remote peers. For example, in [5], [8]–[11], this is fulfilled when a transformation excludes a deletion; in [6] and [7], this is fulfilled when an insertion is integrated in the model.

When pos is at the left end of $curr$, the new node is inserted between $curr.visibleLeft$ and its right neighbor (Line 7). When pos is at the right end of $curr$, the new node is inserted between $curr$ and its right neighbor (Line 9).

The new current position is now set to the right end of the new node (Line 11).

The procedure finally returns the encoded representation of the graph update, which includes the identifier and state vector of the peer, the new node nd, and the place $[nd_l, nd_r]$ at which this new node is inserted, i.e., the right end of nd_l, encoded with $(nd_l.v, nd_l.offset, nd_l.len)$, and the left end of nd_r, encoded with $(nd_r.v, nd_r.offset)$. Procedure $decode(Ins)$ at a remote peer will return a triple (nd', nd'_l, nd'_r) such that:

$$\begin{aligned} & nd'.pid = nd.pid \wedge nd'.v = nd.v \wedge nd'.str = nd.str \\ & \wedge nd'.rendered = false \wedge nd'_l.v = nd_l.v \\ & \wedge nd'_l.offset + nd'_l.len = nd_l.offset + nd_l.len \\ & \wedge nd'_r.v = nd_r.v \wedge nd'_r.offset = nd_r.offset \end{aligned}$$

D. Integrating a local deletion

The main part of the integration is a loop that associates all nodes corresponding to the deleted characters with the del elements of the deletion. In every iteration, pos is at the left end of $curr$. If initially the current position pos is in the middle of $curr$, $curr$ is split at pos and the new right node becomes $curr$ (Lines 2 and 3). If pos is at the right end of $curr$, $curr$'s right visible neighbor becomes the new $curr$ (Lines 4–5). len is the number of characters yet to be handled. nds contains the nodes of this deletion that have been processed so far. nd is the next visible node to be processed.

Lines 8–16 set nd, $curr$ and len properly in different cases. Line 17 associates nd with a del of the deletion and Line 20 pushes nd into nds. Line 19 takes care that the del elements are chained correctly. After the loop, the new current position is placed at the left end of $curr$ (Line 21).

Procedure $integrateL(Del)$

```
1  v ← peer.tick(peer.id);
   startv, startp ← curr.v, cuu.offset
2  if curr.offset < pos < curr.offset + curr.len then
3  |  (−, curr) ← curr.split(pos)
4  else if pos = curr.offset + curr.len then
5  |  curr ← curr.visibleRight
6  len, nds ← Del.len, []
7  while len > 0 do
8  |  if len = curr.len then
9  |  |  nd ← curr; curr ← curr.visibleRight
10 |  |  len ← 0
11 |  else if len < curr.len then
12 |  |  (nd, curr) ← curr.split(curr.offset + len)
13 |  |  len ← 0
14 |  else // len > curr.len
15 |  |  nd ← curr; curr ← curr.visibleRight
16 |  |  len ← len − curr.len
17 |  nd.dels.push(newDel(v))
18 |  if nds ≠ [] then
19 |  |  nds.last.del(v).r, nd.del(v).l ← nd, nds.last
20 |  nds.push(nd)
21 pos ← curr.offset
22 return encode(DEL, peer.id, v, startv, startp, nds)
```

Finally, the procedure returns the encoded representation of the graph update (Line 22). Every node nd in nds is encoded with $(nd.v, nd.offset, nd.len)$. Function $decode(Del)$ at a remote peer will return (pid', v', nds') such that:

$$\begin{aligned} & pid' = peer.id \wedge v' = v \\ & \wedge \forall (nd', len') \in nds', \\ & \quad (nd'.v = nd.v \wedge nd'.offset = nd.offset \\ & \quad \wedge len' = nd.len) \end{aligned}$$

$startv$ and $startp$ are encoded to ensure that even if the start node is split (Line 3), a remote peer can still obtain the node via its hash index. Note that except the first and the last nodes, $nd'.len \leq nd.len$, because at a remote peer a corresponding node may have been split by a concurrent operation.

E. Integrating a remote insertion

A peer only integrates a remote update when it is causally ready, i.e., when all the updates the remote update "saw" upon its generation have been integrated in this peer.

Definition (causally ready)**.** *A remote update with state vector* v_r *from peer* i *is causally ready at a peer with state vector* v_p *iff* $v_r[i] = v_p[i] + 1 \wedge \forall j \neq i, v_r[j] \leq v_p[j]$.

Next, we introduce a few more terms that are relevant for the integration procedure.

Definition (happens before). *Operation op_1 with state vector v_1 happens before operation op_2 with state vector v_2, denoted by $op_1 \rightarrow op_2$, if $\forall i, v_1[i] \leq v_2[i] \wedge \exists j, v_1[j] < v_2[j]$.*

Definition (concurrent operations). *Operation op_1 with state vector v_1 and operation op_2 with state vector v_2 are concurrent, denoted by $op_1 \parallel op_2$, if $op_1 \not\rightarrow op_2 \wedge op_2 \not\rightarrow op_1$.*

Two operations are compatible if they "see" exactly the same states.

Definition (compatible operations). *Operation op_1 with state vector v_1 and operation op_2 with state vector v_2 are compatible, denoted by $op_1 \sqcup op_2$, if $op_1 \parallel op_2$ and there is no op_3, such that $(op_3 \rightarrow op_1 \wedge op_3 \parallel op_2) \vee (op_3 \rightarrow op_2 \wedge op_3 \parallel op_1)$.*

Two concurrent operations are compatible if there does not exist a third operation that happens before one of the operations and is concurrent with the other operation. In the definition, if $op_3 \rightarrow op_1$ and $op_3 \parallel op_2$, op_1 and op_2 are concurrent but not compatible, because op_1 "sees" the effect of op_3 but op_2 does not.

Procedure $integrateR(Ins)$

1 $(nd, nd_l, nd_r) \leftarrow decode(Ins)$
2 **while** $\neg(nd_l.r = nd_r)$ **do**
3 $\quad nds \leftarrow compatibleNodesBetween(nd, nd_l, nd_r)$
4 $\quad (nd_l, nd_r) \leftarrow resolveConflicts(nd, nd_l, nd_r, nds)$
5 $nd_l.r, nd_r.l, nd.l, nd.r \leftarrow nd, nd, nd_l, nd_r$
6 $peer.tick(nd.pid)$
7 $nodes[(nd.v, 0)] \leftarrow nd$

Because of causal readiness, a peer can obtain the insertion node and the place of insertion $[nd_l, nd_r]$ by decoding the received insertion update (Line 1). Node nd_l might have been split locally or remotely ($integrateL(Ins)$ Line 5). Since the right end of nd_l was encoded with $nd_l.v$, $nd_l.offset$ and $nd_l.len$, the correct node in this local peer can be obtained by starting from the node hash-indexed with $(nd_l.v, nd_l.offset)$ and then navigating through the splits until position $nd_l.offset + nd_l.len$. If the insertion is in the middle of a node, the node will be split after $decode$.

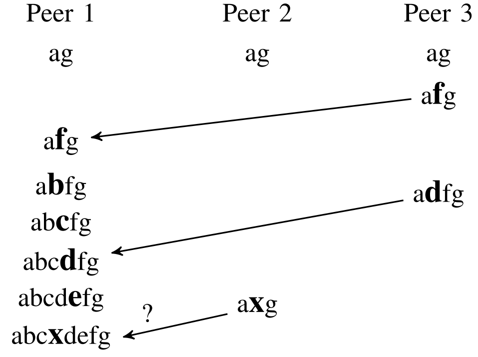
At the local peer, however, there might be nodes between nd_l and nd_r due to conflicting insertions.

Definition (conflicting insertions). *Two concurrent insertions conflict if they are to be inserted at the same place.*

Policy II is used to resolve conflicts between compatible insertion operations.

Policy (II). *For two compatible conflicting insertions op_1 and op_2, op_1 is placed to the left of op_2 if $op_1.pid < op_2.pid$.*

The loop in $ingtegrateR(Ins)$ resolves conflicts among compatible insertions in each iteration. Line 3 obtains the nodes between nd_l and nd_r that are compatible with the remote insertion node nd. Line 4 resolves the conflicts with



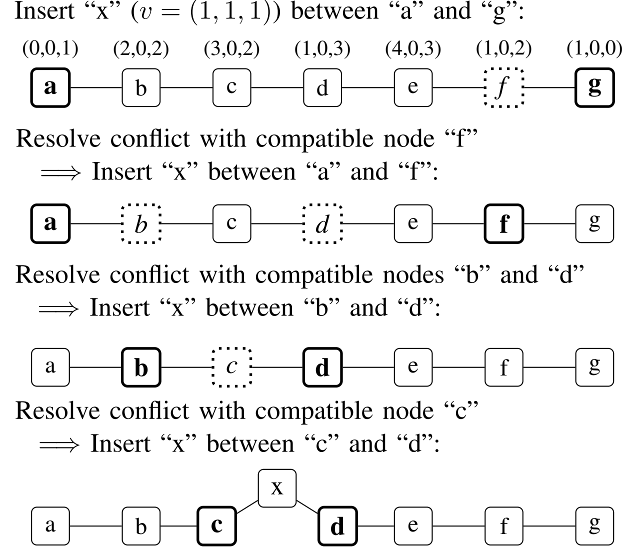


Fig. 5. Resolving conflicts

Policy II to decide the new place, between nd_l and nd_r, where nd is to be inserted. The loop continues until there is no node between nd_l and nd_r, and nd is inserted between nd_l and nd_r (Line 5). Finally, the peer advances its state vector for the integrated insertion (Line 6) and hash-indexes the inserted node (Line 7).

Fig. 5 illustrates the process of how conflicts are resolved. In every iteration, "x" is to be inserted between two nodes ▢. An iteration starts with finding the compatible nodes ⬚ between the two nodes and then resolving conflicts with Policy II. Finally, when there is no node between "c" and "d", "x" is inserted in between. This process is in essence very similar to the algorithm presented in [7].

F. Integrating a remote deletion

A peer integrates a remote deletion by associating the corresponding nodes with the del elements of the deletion. To make our work comparable with related work, we introduce Policy DD.

Policy (DD). *The characters that are deleted concurrently by multiple peers are regarded as being deleted multiple times.*

All previous work adopted a *unit-operation policy*. That is, when a character is deleted by multiple peers, the character is

regarded as being delete by the first deletion. The subsequent deletions are handled as unit operations that have no effect. Unit-operation policy seems to have the same effect as Policy DD with regard to the invisibility of the deleted character. They are, however, different when undo is taken into account. With unit-operation policy, a single undo will bring back the visibility of the deleted character, whereas with Policy DD, the character is only visible when all deletions on it are undone.

```
Procedure integrateR(Del)
1  (pid, v, nds) ← decode(Del)
2  nd_l, split ← nil, false
3  while split ∨ nds ≠ [] do
4      if split then
5          len, nd, split ← (len − nd.len), nd.ir, false
6      else
7          (nd, len) ← nds.removeFirst
8      if len < nd.len then
9          (nd, −) ← nd.split(nd.offset + len)
10     else if len > nd.len then
11         split ← true
12     if nd.visible ∧ nd.rendered then
13         nd.rendered ← false
14     else
15         nd.rendered ← true
16     nd.dels.push(newDel(v))
17     if nd_l then  nd_l.del(v).r, nd.del(v).l ← nd, nd_l
18     nd_l ← nd
19 peer.tick(pid)
```

Decoding a remote deletion generates a list nds of node-length pairs for the deleted characters (Line 1). The main loop associates the corresponding nodes of the local graph with the del elements of the deletion. nd_l is the last node that was associated with a del and must be linked to from the del of nd being handled in the current iteration. The boolean variable $split$ indicates whether a node in nds has been split locally. Lines 4–7 set the node nd to be handled in the current iteration. If the node of the previous iteration had been split locally, nd is set to the right part of the split (Line 5); otherwise nd is the next node in nds (Line 7). There are then three cases to consider: the remote peer did a split this peer is unaware of (Lines 8–9); the local peer did a split the remote peer was unaware of (Lines 10-11); or otherwise (do nothing). nd is marked as not rendered only when the change of visibility makes a change in the view (Lines 12–15). Then nd is associated with the del (Line 16) and the del elements are linked properly (Line 17).

G. Model rendering

Rendering a model brings a view updated with the remote updates that have been integrated in the model. With a partial rendering, only a specific number of characters near the current position are rendered.

A redering calls procedure $render$ recursively, starting with the current node $curr$ by calling $render(curr, pos, \delta_0, 0)$. When it returns, nodes corresponding to characters between pos and $pos + \delta_0$ in the view are known to be rendered.

Procedure $render$ renders from position p of node nd. Except the starting node $curr$, p is the left (or right) end of nd for redering toward right (or left). If δ is positive, δ characters to the right are rendered; otherwise, $-\delta$ characters to the left are rendered. δ_{done} indicates how far the rendering has been done so far. The pre-condition of running $render$ is that all nodes between $curr$ and nd are rendered.

```
Procedure render(nd, p, δ, δ_done)
1  if nd.rendered then
2      if nd.visible then
3          if δ > 0 then  δ_n ← nd.offset + nd.len − p
4          else           δ_n ← p − nd.offset
5  else
6      δ_n ← renderNode(nd, δ_done)
7  if δ > 0 then  // rightwards
8      if δ − δ_n ≤ 0 then  return
9      else  render(nd.r, nd.r.offset, δ − δ_n, δ_done + δ_n)
10 else // leftwards
11     if δ + δ_n ≥ 0 then  return
12     else  render(nd.r, nd.r.offset + nd.r.len,
13                     δ + δ_n, δ_done − δ_n)
```

In the procedure, δ_n is the number of characters that nd contributes to the entire rendering. $render$ calls $renderNode$ if nd has not been rendered (Line 6). The rendering continues recursively until the expected number of characters are rendered (Lines 7–13).

```
Procedure renderNode(nd, δ)
1  view.move(δ)
2  if nd.visible then
3      view.ins(nd.str)
4      if δ > 0 then view.move(−δ)
5      else       view.move(−δ + nd.len)
6  else
7      if δ > 0 then
8          view.del(nd.len)
9          view.move(−δ)
10     else
11         view.move(−nd.len)
12         view.del(nd.len)
13         view.move(−δ − nd.len)
14 nd.rendered ← true
15 return nd.visible? nd.len : 0
```

Procedure $renderNode$ renders node nd and returns the number of characters that are added to the view. It takes

an additional argument δ. The pre-condition of running $renderNode$ is that nd is not rendered, all nodes between $curr$ and nd are rendered, and there are δ visible characters between $(curr, pos)$ and nd. It moves the current view position to the place of the update (Line 1, and additionally Line 11 to the left end of the sub-string to be deleted), makes the view update (Lines 3, 8 and 12), and then brings the current view position back to its original place (Lines 4–5, 9 and 13). If the update is to the left of the current position, the new current position must accommodate the update (Lines 5 and 13).

H. Moving current position

A position move in the model starts with calling $move(curr, pos, \delta, 0)$, which traverses the nodes recursively until it reaches the destination node. If δ is positive, it moves δ characters to the right; otherwise, it moves $-\delta$ characters to the left. δ_{done} is the move that has been done so far. Upon the execution of $move(nd, p, \delta, \delta_{done})$, position p in node nd is δ_{done} characters away from the original position of the move. Except for the starting node, p is the left (or right) end of nd when moving toward right (or left). The pre-condition of running $move$ is that all nodes between $curr$ and nd are rendered, and there are δ_{done} visible characters between $(curr, pos)$ and (nd, p).

Procedure $move(nd, p, \delta, \delta_{done})$

```
1  if ¬nd.rendered then renderNode(nd, δ_done)
2  if nd.visible then
3      if p + δ > nd.offset + nd.len then  // rightwards
4          δ_done ← δ_done + nd.offset + nd.len − p
5          δ ← (δ + p − nd.offset − nd.len)
6          nd ← nd.r; p ← nd.offset
7      else if p + δ < nd.offset then  // leftwards
8          δ_done ← δ_done + nd.offset − p
9          δ ← (δ + p − nd.offset)
10         nd ← nd.l; p ← nd.offset + nd.len
11     else  // destination
12         curr, pos ← nd, (p + δ); return
13 else
14     if δ > 0 then nd ← nd.r; p ← nd.offset
15     else nd ← nd.l; p ← nd.offset + nd.len
16 move(nd, p, δ, δ_done)
```

Along its way, $move$ renders the nodes that have not been rendered (Line 1). It stops when nd is the destination (Line 12); otherwise, it sets the new arguments (Lines 2–10, 14–15) and continues its journey (Line 16).

The cost of $move$ is dependent on the distance of the move and the granularity of the nodes. Note that it is only necessary to call $move$ of a model prior to the integration of a local insertion or deletion. Consecutive moves in the view can be aggregated into a single one. Furthermore, because position moves and editing operations are queued in the local operation queue, they can be integrated in groups. For example, the operations in the group can be integrated from one end of the graph to the other to minimize the total move distance, very much like the elevator algorithm for disk scheduling.

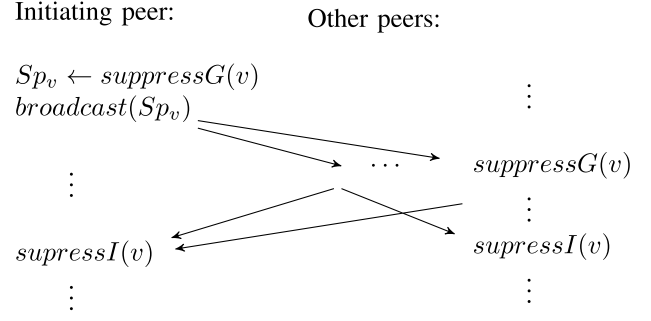


Fig. 6. Two-phase model suppression

I. Model suppression

Model graphs get larger and more complex along with editing operations, leading to larger hash tables of nodes and increasing overhead of procedures like $model.move$. Model suppression reduces the size and complexity of the graph.

One issue is that when a peer suppresses its model graph, other peers may concurrently refer to the nodes prior to the suppression. To address this issue, we introduce a two-phase protocol (Fig 6). In the first phase, a peer first suppresses its graph with $suppressG$. $suppressG$ builds a temporary mapping from the nodes before the suppression to the nodes after the suppression. The peer then broadcasts the suppression command to the other peers that in turn run $suppressG$ to suppress their own graphs. During this phase, when a peer receives remote updates that refer to nodes before the suppression, it uses the temporary mapping to find the correct position in its suppressed graph. When a peer knows that all peers have suppressed their graphs (by checking the state vectors of the incoming updates), it discards the temporary mapping by running $suppressI$.

$suppressG$ suppresses the model graph up to state vector v. A node is *suppressed* if all operations it is part of have state vectors smaller or equal to v. A node *remains* in the suppressed graph if it is part of at least one operation with state vector v_{op} such that $\exists i, v_{op}[i] > v[i]$. $suppressG$ replaces all suppressed nodes with a single new insertion consisting of the visible characters of these nodes. Nodes that remain may split the new insertion into several nodes.

The loop of the procedure walks through the graph from left to right and builds the nodes for the new insertion. In the procedure, nd is the node in the original graph to be handled in the current iteration, nd_i is the node under construction for the new insertion, str is the string currently built for nd_i, p is the offset of the left end of nd_i from the leftmost position of the entire insertion, and nd_l is the last node that remains in the suppressed graph. If the insertion is split by a remaining node, nd_{il} is the node of the new insertion left to nd_i.

Procedure *suppressG*(pid, v_{new}, v)

```
1  nd, str, p, nd_l, nd_il ← leftmostNode, "", 0, nil, nil
2  while nd do
3      discard all del ∈ nd.dels such that del.v ≤ v
4      if nd.v ≤ v ∧ nd.dels = ∅ then  // suppressed
5          nodes[(nd.v, nd.offset)] ← (v_new, p, str.len)
6          if nd.visible then  str ← str + nd.str
7          suppressed_nds.push(nd)
8      else // remains
9          if str.len > 0 then
10             nd_i ← newNode(pid : pid, v : v_new,
11                            offset : p, str : str)
12             if nd_il then nd_il.ir, nd_i.il ← nd_i, nd_il
13             if nd_l then nd_l.r, nd_i.l ← nd_i, nd_l
14             nodes[(v, p)] ← nd_i
15             p, str, nd_il ← p + str.len, "", nd_i
16         nd_l ← nd
17     nd ← nd.r
18 discard nodes in suppressed_nds
19 return encode(SP, peer.id, v_new, v)
```

For node nd in the original graph, all del elements with state vector smaller or equal to v are discarded (Line 3). If nd is to be suppressed (Line 4), a redirection mapping is constructed for nd (Line 5). If nd is visible, its string is appended to str (Line 6).

If nd remains (Line 8) and the string for the insertion node has been partially built (Line 9), nd splits the insertion into a new node nd_i. So nd_i is constructed (Line 10), linked with nd_{il} (Line 12) and nd_l (Line 13), and hash-indexed (Line 14). Line 15 prepares the construction of the new insertion node in the subsequent iterations.

IV. Correctness

This section gives a sketch of the correctness of the approach. According to [5], an OT system is correct if the following conditions hold:

- Causality preservation: an operation is executed at a peer only when all operations that happened before it have been executed at the peer.
- Admissibility preservation: established effects relations are preserved at all peers.

Our approach, as most other approaches, preserves causality, because a peer integrates a remote operation only when the operation is causally ready at the peer.

Admissibility preservation can be analysed as follows. The system must guarantee that all peers eventually have the same set of characters associated with the same total effects-relation order. Characters in the same node follows implicitly an effects-relation order. Splitting a node keeps the same effects-relation order. New characters and effects relations are introduced with insertions. It is thus crucial that effects relations established by insertions are preserved at all peers.

- When a local operation is integrated, there is no conflicting operation that has been integrated but not rendered. This is because (1) all local operations are integrated prior to remote operations (Procedure $ingerate$), (2) the region the end user is editing is rendered (Procedures $integrate$ and $render$), and (3) a position move renders the nodes along the way (Procedure $move$). Therefore a local insertion, upon its integration, never "meets" a conflicting insertion.
- Because a remote insertion can only be integrated at a peer when it is causally ready, the insertion position encoded during $integrateL$ will be uniquely located by $integrateR$.
- When there are concurrent insertions that conflict with ins, a unique ordering will be established at all peers. This can be seen with theoretical global iterations that are established in such a way that in an iteration, all peers have the same set of compatible operations at the insertion position. Because all peers adhere to the same Policy II, a unique ordering among these compatible operations is established. After this iteration, it is as if all peers have executed all these compatible operations. Starting from the state established by this iteration, the following iteration establishes the same ordering among the new compatible operations in the new insertion position of this new state. The process goes on until there is no more concurrent conflicting operations.

Finally, all peers eventually have the same set of visible characters. This is achieved by associating the del elements of the same deletion to the same set of characters at all peers. Again, because a remote deletion operation can only be integrated when it is causally ready, all characters that $integrateL$ are aware of are uniquely located by $integrateR$.

V. Performance

Both $integrateL$ and $integrateR$ have time complexity $O(1)$ with respect to the length of operation history. A graph update is only concerned with the nodes at the place of insertion or deletion. For $integrateL$ it is node $curr$. For $integraeR$, the nodes can be obtained via the hash index. The procedures involve loops or node traversals. The numbers of the nodes depend on the sizes of deletions or the number of concurrent operations and splits, and thus are independent of the length of the operation history.

A model graph may not necessarily take more space than a typical operation history. If we regard history entries as corresponding to graph nodes, every character operation has an entry in history. So the number of history entries is typically an order of magnitude greater than the number of graph nodes.

Moving the current position in the model has time complexity $O(N)$, where N is the number of nodes traversed. The runtime overhead depends on the granularity of nodes and the distance of the move. Model suppressions reduce the sizes of graphes and therefore reduce the overhead of position moves. Furthermore, integrating local operations in groups may reduce the zigzags and thus the total distances of position moves.

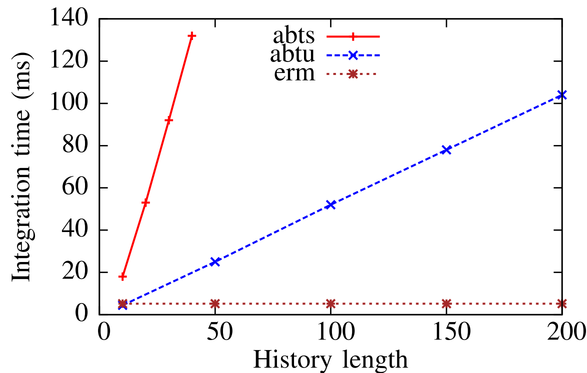


Fig. 7. Integration time

Model rendering has time complexity $O(N)$, where N is the number of traversed nodes. Rendering can be partial. A peer can bound the overhead by only rendering a restricted region displayed by the editor. Again, model suppression helps reduce the overhead of model rendering.

Model suppression has time complexity $O(N)$, where N is the total number of nodes in the model graph. The frequency of model suppressions is a trade-off issue. Frequent suppressions keep the graph small but bear frequent overhead of the suppressions.

We have implemented a prototype of the core procedures in Emacs Lisp. To compare our approach with related work, we have also implemented ABTS [8] and ABTU [10] in Emacs Lisp. ABTS supports string operations and ABTU is one of few that has linear complexity. No special effort was made for performance optimization or byte compilation. We measured the time for local and remote integrations using Emacs Lisp Profiler, which is not really precise but is sufficient to indicate the tendance. With different operation patterns, the overhead of local and remote integrations may be different in ABTS and ABTU. The sum of local and remote integrations seems quite independent of the operation patterns. Fig. 7 shows the sum of local and remote integrations at different history lengths. The measurement was taken under Aquamacs 2.2 (GNU Emacs 23.3.1) running on an iMac with 64-bit Mac OS X 10.6.7, 2.8GHz Intel Core i5 and 4GB RAM. In the figure, "abts" stands for ABTS, "abtu" for ABTU, and "erm" for effects-relation model, i.e. our work. Although there is no operation history in our work, we run the same sequences of operations for all three approaches to make the results comparable (except that "abtu" has character operations whereas "abts" and "erm" have string operations with string lengths between 5 and 10 characters). As Fig. 7 shows, our work takes nearly constant time (2.6ms) for operation integration whereas the time for integration with ABTS and ABTU grow with history length.

VI. Conclusion

OT has been established as the concurrency control mechanism for collaborative editing, because it is non-blocking and therefore may provide the responsiveness that is not achievable with a mutual-exclusion based mechanism. However, this potential responsiveness has not been fully realized in practice due to the time-consuming computation for operation transformation and integration. Currently, the best time complexity of operation transformation and integration is $O(|H|)$ where $|H|$ is the length of operation history. Furthermore, most work supports only character operations. Consequently, operation histories typically have lengths in the order of thousands and grow indefinitely. Transforming and integrating a single remote operation can thus easily involve seconds of intensive computation. Our approach supports transformation and integration of string operations with constant-time complexity. In an implementation i Emacs Lisp without any special efforts in performance optimization, it takes only a couple of milliseconds to transform and integrate an operation of a string of almost arbitrary length. Furthermore, our approach has the following additional desirable features: the data structure can be suppressed at any time to reduce its size and complexity; integrated remote operations can be partially rendered to further reduce both runtime overhead and interference with the user's current editing activities. Our work is based on admissibility preservation, a correctness criterion with which the correctness of our approach can be formally proven.

References

[1] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *SIGMOD Conference*, J. Clifford, B. G. Lindsay, and D. Maier, Eds. ACM Press, 1989, pp. 399–407.

[2] A. Imine, P. Molli, G. Oster, and M. Rusinowitch, "Proving correctness of transformation functions functions in real-time groupware," in *ECSCW*, K. Kuutti, E. H. Karsten, G. Fitzpatrick, P. Dourish, and K. Schmidt, Eds. Springer, 2003, pp. 277–293.

[3] D. Li and R. Li, "Preserving operation effects relation in group editors," in *CSCW*, J. D. Herbsleb and G. M. Olson, Eds. ACM, 2004, pp. 457–466.

[4] ——, "An approach to ensuring consistency in peer-to-peer real-time group editors," *Computer Supported Cooperative Work*, vol. 17, no. 5-6, pp. 553–611, 2008.

[5] ——, "An admissibility-based operational transformation framework for collaborative editing systems," *Computer Supported Cooperative Work*, vol. 19, no. 1, pp. 1–43, 2010.

[6] G. Oster, P. Molli, P. Urso, and A. Imine, "Tombstone transformation functions for ensuring consistency in collaborative editing systems," in *CollaborateCom*. IEEE, 2006, pp. 1–10.

[7] G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for p2p collaborative editing," in *CSCW*, P. J. Hinds and D. Martin, Eds. ACM, 2006, pp. 259–268.

[8] B. Shao, D. Li, and N. Gu, "ABTS: A transformation-based consistency control algorithm for wide-area collaborative applications," in *CollaborateCom*. IEEE, 2009, pp. 1–10.

[9] ——, "A sequence transformation algorithm for supporting cooperative work on mobile devices," in *CSCW*, K. I. Quinn, C. Gutwin, and J. C. Tang, Eds. ACM, 2010, pp. 159–168.

[10] ——, "An algorithm for selective undo of any operation in collaborative applications," in *GROUP*, W. G. Lutters, D. H. Sonnenwald, T. Gross, and M. Reddy, Eds. ACM, 2010, pp. 131–140.

[11] B. Shao, D. Li, T. Lu, and N. Gu, "An operational transformation based synchronization protocol for web 2.0 applications," in *CSCW*, P. J. Hinds, J. C. Tang, J. Wang, J. E. Bardram, and N. Ducheneaut, Eds. ACM, 2011, pp. 563–572.

[12] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, pp. 63–108, 1998.