ABTS: A Transformation-Based Consistency Control Algorithm for Wide-Area Collaborative Applications

Bin Shao †, Du Li ‡, Ning Gu †
† School of Computer Science, Fudan University, Shanghai, China
‡ Nokia Research Center, Palo Alto, California, USA
Email: binshao@fudan.edu.cn; lidu008@gmail.com; ninggu@fudan.edu.cn

Abstract

Operational transformation (OT) is an established optimistic consistency control method in collaborative applications. Most existing OT algorithms are developed under a well-accepted framework with a condition that cannot be formally proved. In addition, they generally support two character-based primitive operations, insert and delete, in a linear data structure. This paper proposes a novel OT algorithm that addresses the above two challenges as follows: First, it is based on a recent theoretical framework with formal conditions such that its correctness can be proved. Secondly, it supports two string-based primitive operations and handles overlapping and splitting of operations. As a result, this algorithm can be applied in a wide range of practical collaborative applications.

1. Introduction

Operational transformation (OT) [1, 10] is an optimistic consistency control method that lies in the heart of many collaborative applications such as group editors [11] and Google Wave. [1] The method replicates the shared data at cooperating sites. Local operations are always executed as soon as they are generated by the user. Remote operations are transformed before execution to repair inconsistencies. As a result, local responsiveness is not sensitive to communication latencies, which makes the method particularly appealing to collaborative applications running over wide-area networks with high and nondeterministic latencies.

A plethora of OT algorithms have been proposed over the past two decades, e.g., [4, 9, 10, 11, 12]. There are two open challenges: First, most of them are developed under the framework of Sun et al [11], which includes an informal condition called "intention preservation". As a consequence, their correctness cannot be formally proved and counterexamples are often reported, as confirmed in [4, 7, 5, 8]. Secondly, except for [11], all other OT algorithms only consider two character-based primitive operations. Although this simplification is theoretically acceptable, there is a practicality gap when applying those algorithms to real collaborative applications in which string-based operations are common. The handling of string operations is very intricate, as confirmed in [11].

To address the above two challenges, this paper proposes a novel OT algorithm called Admissibility-Based Transformation with Strings (ABTS): First, it is based on the ABT framework [6, 5] which formalizes two correctness condition, causality and admissibility preservation. Conceptually, admissibility requires that the execution of every operation not violate the relative position of effects produced by operations that have been executed so far. As a result, the original ABT algorithm and the derived ABTS algorithm can be formally proved. Secondly, ABTS supports two string-based primitive operations and their overlapping and splitting when concurrent operations are transformed. As a result, the algorithm can be directly applied in a range of collaborative applications that require string operations. Moreover, the design of ABTS will provide a new starting point when extending OT algorithms to support composite and block operations that semantically must be applied together, such as cut-paste and find-replace.

Section 2 gives the background of this research. Section 3 introduces notations. Section 4 presents the ABTS algorithm. Sections 5 and 6 analyze its correctness and complexities, respectively. Finally, Section 7 concludes.

2. Background and Related Work

To illustrate the basic ideas of OT, consider a scenario in which two users, A and B, collaboratively edit a shared document which includes a list of guests to invite for a party. The document is replicated at the two sites when the users discuss about it online. Suppose that the list is ini-

[1] http://www.waveprotocol.org/whitepapers/operational-transform

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8271
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8271

tially "Tom" and the first position of a string is zero. User A extends the list to "Karen,Tom" by operation o_A = insert(0, "Karen,"). At the same time, user B extends the list to "Tom,Sarah" by operation O_B = insert(3,",Sarah"). The two sites diverge before their results are merged.

When A receives o_B, if the operation were executed as-is, the wrong result"Kar,Sarahen,Tom" would yield in the list of A. The intuition of OT [1] is to transform remote operations to incorporate the effects of concurrent local operations that have been executed earlier. In this scenario, for example, A transforms o_B into a form o'_B such that o'_B can be correctly executed in current state "Karen,Tom" of site A. Considering the fact that A has inserted a string of six characters on the left side of the intended position of o_B, we must shift the position of o_B by six to the right, yielding o'_B = insert(9, "Sarah"). Execution of o'_B in state "Karen,Tom" results in the right list of "Karen,Tom,Sarah". On the other hand, when user B receives o_A, the operation can be executed as-is in current state of B because the target position of o_A is not affected by the execution of o_B. This results in list "Karen,Tom,Sarah". Now the two sites converge.

The philosophy of OT is to avoid operation overwriting so as not to lose user interaction results. To reduce chaos in the result as caused by concurrency, Sun et al [11] propose three conditions to constrain the output, namely, convergence, causality and intention preservation. Unfortunately, although intuitive and widely accepted, intention preservation is not a well-formalized condition. As a consequence, OT algorithms developed under their framework (e.g., [10, 11, 9]) are not completely proved and counterexamples are often reported [4, 7, 5, 8].

Our work has established formal, provable correctness conditions [4, 7, 6]. In particular, the Admissibility-Based Transformation (ABT) framework [6, 5] proposes an alternative constraint called admissibility, which conceptually requires that the execution of any operation not violate the character order established by previous executions in the system. However, they mainly serve theoretical purposes and only consider two characterwise primitives. The presented stringwise ABTS algorithm is a significant extension to its characterwise version ABT [6, 5]. Specifically, when transforming two stringwise operations, the algorithm is greatly complicated by the handling of position relations between the operation regions because operations may be split cascadingly during transformation.

In the literature, only the GOT algorithm [11] supports stringwise operations. In their follow-up work (e.g., [10, 12]), new algorithms are proposed to replace GOT. We believe that they have implemented stringwise operations. However, in their publications, they have not addressed how to prove the correctness of their algorithms and how to support stringwise operations in their new algorithms. Note that this is not to say that their algorithms are incorrect.

Nevertheless, GOT converges by maintaining a predefined total order of operations and using a do/undo/redo based control procedure integrated with OT. In order to utilize the do/undo/redo mechanism, GOT requires that all operations be reversible. To ensure reversibility, a "lost and found" mechanism is employed to save and restore the object relations between two operations in transformation. Hence its space complexity is $O(|H|^2)$, where H is the history buffer. By comparison, ABTS requires neither a total order of execution nor reversibility of operations; correctness is ensured without saving the object relation; its space complexity is $O(|H|)$. We prefer not to compare the time complexity of GOT before all its details are presented, e.g., their solutions to the counterexample identified in [4].

3. System Model and Notations

A **system** consists of a number of collaborating sites. The same version of the shared data is replicated at all sites when a session starts. For local responsiveness, each site submits operations only to its local replica which are executed immediately. In the background, local operations are propagated to remote sites.

The **shared data** is abstracted as a linear string of atomic characters. Objects are referred to by their positions in the string, starting from zero. For simplicity, we consider two **primitive operations**, namely, $insert(p, s)$ and $delete(p, s)$, which insert and delete a string s at position p in the shared data, respectively. Any operation o has the following attributes: $o.id$ is the unique id of the site that originally submits o; $o.type$ is the operation type which is either $insert$ or $delete$; $o.pos$ is the position in the shared data at which o is applied; $o.str$ is the target string which the operation inserts or deletes. We use established notations [2] happens-before ($\rightarrow$) and concurrent ($\|$) to denote the temporal relations between operations.

Note that, for any operation o, $o.pos$ is always defined relative to some specific state of the shared data. Following notations in [10], the **definition state** of o, denoted as $dst(o)$, is the state in which $o.pos$ is defined. Given any two operations o_1 and o_2, we say that they are **contextually equivalent**, denoted as $o_1 \sqcup o_2$, if dst(o_1) = dst(o_2); they are **contextually serialized**, denoted as $o_1 \mapsto o_2$, if o_2's position is defined in the resulting state of applying o_1 (but no other operation). For example, in the scenario given in Section 2, we have $o_A \parallel o_B$, $o_A \sqcup o_B$, and $o_A \mapsto o'_B$.

A **list** is an ordered collection of objects, denoted as $[e_1, \ldots, e_n]$. An empty list is denoted as []. For any list L, notation $|L|$ denotes the number of objects in L and $|[]|$ = 0. Borrowing notations from Prolog, if $|L| > 0$, $L.head$ refers to its first element $L[0]$, and $L.tail$ refers to remaining list $L[1..|L|-1]$. For example, if L=[a,b,c], then L.head = a and L.tail = [b,c]. Operator $\cdot$ concatenates two lists, or a

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8271
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8271

Notation	Brief Description
o.id	the id of site that originally generates o
o.type	the operation type of o, either ins or del
o.pos	the position of o relative to the data model
o.str	the string inserted or deleted by o
$o_1 \rightarrow o_2$	o_1 happens before o_2
$o_1 \parallel o_2$	o_1 and o_2 are concurrent
$o_1 \sqcup o_2$	o_1 and o_2 are contextual equivalent
$o_1 \mapsto o_2$	o_1 and o_2 are contextually serialized
$[o_1, o_2]$	an ordered list of two operations
$< o_1, o_2 >$	a 2-operation sequence in which $o_1 \mapsto o_2$
$\|L\|$	the number of objects in list/sequence L
$L_1 \cdot L_2$	a concatenated list/seq of two lists/seqs

Table 1. A summary of the main notations.

list and an object. For example, for $L = [a, b, c]$, we have $L = [a] \cdot [b, c] = [a, b] \cdot c$.

A **sequence** is a special list in which all elements are operations that are contextually serialized. A sequence sq of n operations is denoted as $sq = <o_1, o_2, \ldots, o_n>$, where $o_1 \mapsto o_2 \mapsto \ldots \mapsto o_n$. An empty sequence is denoted as $<>$. The above list notations also apply to sequences, e.g., $|sq| = n$ and $sq = <o_1> \cdot <o_2, \ldots o_n>$.

4. The ABTS Algorithm

We first overview the ABTS algorithm and then explain the involved procedures in the following subsections.

4.1. Overview

A history buffer H is maintained at each site which logs operations that have been applied to the data replica at that site. For correctness reasons [6, 5], H is maintained as a concatenation of of two subsequences, H_i and H_d, which record the executed $insert$ and $delete$ operations in their order of execution, respectively. That is, $H = H_i \cdot H_d$. In addition, each site maintains RQ, a list of operations received from remote sites in their order of arrival. Each site j runs the following three concurrent threads:

Thread $\mathcal{L}$ each time receives a local operation o, applies it to the data replica, calls algorithm $updateHL$ to update H and compute o', a transformed version of o, and propagates the resulting o' to remote sites.

Thread $\mathcal{N}$ receives remote operations from the network and appends them to RQ in their order of arrival.

Thread $\mathcal{R}$ scans RQ for a remote operation o at a time that is causally-ready, i.e., all operations that happen before o have been executed at site j. Then algorithm $updateHR$ is called to update H and transform o into a version o' that can be correctly executed in current state of site j. After that, o' is executed on the data replica at site j.

4.2. Algorithm updateHL

By the way H is maintained, a new local insertion o_i must be appended to H_i and a new deletion o_d to H_d. Note that all operations that have been executed on the local data replica are included in H; the new local operation o (o_i or o_d) is defined in the current state of the shared data. That is, all operations in H happen before o (or $H \rightarrow o$); H and o are contextually serialized (or $H \mapsto o$). Hence, we can directly append o_d to H_d because $H_d \mapsto o_d$ also holds. However, we cannot directly append o_i to H_i because $H_i \mapsto o_i$ does not hold due to the existence of H_d.

We solve this problem by computing o_i', some version of o_i, such that $H_i \mapsto o_i'$. This is achieved by swapping H_d and o_i. Before the swapping, $H_d \mapsto o_i$. As a result of the swapping, they become H_d' and o_i', respectively, such that $o_i' \mapsto H_d'$. Then o_i' can be appended to H_i.

To explain swapping, consider a scenario with initial state "xy". First execute o_1 = delete(1, 'y') to reach state "x". Then execute o_2 = insert(0, 'z') to yield state "zx". The relation is $o_1 \mapsto o_2$. If we swap o_1 and o_2, yielding o_1' and o_2', respectively, such that $o_2' \mapsto o_1'$, it must be that o_2' = insert(0, 'z') and o_1' = delete(2, 'y'). As a result, o_2' is executed before o_1' yet they produce the same effects.

Function updateHL(o) not only appends o to the right subsequence but also its swapping process excludes the effects of H_d from o as if no deletions had been executed before o at current site. In either case of o (o_i or o_d), after swapping o and H_d, we obtain $o' \mapsto H_d'$. As a result, none of the deletion effects of H_d are included in the definition state of o' when it is propagated to remote sites.

Algorithm 1 *updateHL*(o) : o'

```
1: if o.type = ins then
2:     (o', H_d') ← swapDsqI(H_d, o)
3:     H ← H_i · o' · H_d'
4: else
5:     sq ← H_d
6:     (o', sq') ← swapDsqD(sq, o)
7:     H ← H_i · H_d · o
8: end if
9: return o'
```

Based on the above explanation, Algorithm 1 specifies function updateHL, which works as follows: If the new local operation o is an insertion, we swap it with H_d and append the resulting o' to H_i. Then we update the history to $H_i \cdot o' \cdot H_d'$. The resulting o' is returned (line 9) and will be propagated to remote sites. On the other hand, if o is a deletion, we directly append it to subsequence H_d (line 7) and update the history to $H_i \cdot H_d \cdot o$. Meanwhile we exclude

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8271
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8271

the effects of H_d from o by swapping o with a copy of H_d (lines 5-6). Then the resulting o' is returned and propagated to remote sites. The two swapping procedures, $swapDsqI$ and $swapDsqD$, will be presented later in Section 4.6.

4.3. Algorithm updateHR

Function updateHR(o) has two goals, where o is a causally-ready remote operation: First, it appends o to H_i or H_d depending on $o.type$. Secondly, it outputs o', a version of o, such that o' can be executed in current state.

When updateHR(o) is called in thread $\mathcal{R}$, operation o must be causally ready. That is, all operations that happen before o are already executed and included in the history $H = H_i \cdot H_d$. However, note that some operations that are concurrent with o may also have been executed at current site. That is, subsequences H_i and H_d include both operations that happen before o and those concurrent with o.

The intuition of OT [1] is to transform an operation o with those with effects not included in o to incorporate their effects. This process is called inclusion transformation or IT [10]. For example, in the scenario in Section 2, by step o'_B = IT(o_B, o_A), operation o_B is inclusively transformed with o_A to incorporate the effect of o_A. The relationship between the two operations is $o_B \sqcup o_A$ before the IT step and $o_A \mapsto o'_B$ after. Then the resulting o'_B can be correctly executed in current state of site A.

In fact, we do not need to transform o with all operations in H because the effects of some of operations in H_i that happen before o are already included in o. To distinguish, we must somehow transpose H_i into two contextually serialized subsequences sq_h and sq_c, such that $H_i = sq_h \cdot sq_c$, where sq_h contains all operations in H_i that happen before o and sq_c contains all operations in H_i that are concurrent with o. Then history H is equivalent to $sq_h \cdot sq_c \cdot H_d$.

According to Section 4.2, before o is propagated, it has excluded the effects of all deletions that happen before it; however, it includes the effects of all insertions that happen before it, which are exactly all the operations in subsequence sq_h. Hence o does not include any effects in subsequence $sq_c \cdot H_d$. Then $sq_h \mapsto o$ and $o \sqcup (sq_c \cdot H_d)$. If we get o' by inclusively transforming o with $sq_c \cdot H_d$, then o' can be executed in current state. After the transformation, the relationship will be $(sq_c \cdot H_d) \mapsto o'$ and $H \mapsto o'$.

Now consider the goal of how to add o to H. If o is a deletion, then o' can be directly appended to H_d. However, if o is an insertion, we only need to transform o with sq_c to get o'' and append o'' to H_i. Nevertheless, we cannot naively add o'' between H_i and H_d because, although $H_i \mapsto o''$, the relationship is $o'' \sqcup H_d$ rather than $o'' \mapsto H_d$. Therefore, we must first transform all operations in H_d to incorporate the effect of o'', yielding H'_d, and then update the history H to $H_i \cdot o'' \cdot H'_d$.

Algorithm 2 *updateHR*(o) : o'

```
1: (sq_h, sq_c) ← transposeHC(H_i, o)
2: o'' ← ITOSq(o, sq_c)
3: o' ← ITOSq(o'', H_d)
4: if o.type = ins then
5:    H'_d ← ITDsqI(H_d, o'')
6:    H ← H_i · o'' · H'_d
7: else
8:    H ← H_i · H_d · o'
9: end if
10: return o'
```

As in Algorithm 2, we specify function updateHR based on the above discussions. In line 1, it first transposes H_i into two contextually serialized subsequences, sq_h and sq_c, by calling algorithm transposeHC. In line 2, it calls algorithm ITOSq to get o'' by transforming o with sq_c. Then in line 3, it calls algorithm ITOSq again to get o' by transforming o'' with H_d. If o is an insertion, it transforms H_d to incorporate the effect of o''. After that, o'' is added between H_i and the resulting H'_d. If o is a deletion, it simply appends o' to H_d. Finally, o' is returned and executed by thread $\mathcal{R}$ in the current state of the shared data.

Algorithm transposeHC is already well-understood [4, 6, 9, 10] and here omitted. The two IT functions, ITOSq and ITDsqI, will be explained in Section 4.5.

4.4. Atomic and Composite Operations

To support stringwise transformation, we need to introduce a few more notations. Given any string s, notation $|s|$ is the number of characters in s. If $0 \leq i < j \leq |s|$, notation $s[i{:}j]$ returns a substring of s starting from position i to position j-1. If j is not specified, $s[i{:}]$ returns a substring from i to the end. For example, let s="abc", then $|s|$=3 and s[0:2]="ab" and s[1:]="bc".

A stringwise operation can be denoted as a list of sub-operations that achieves the same effects. We use notation $o.sol$ to denote the sub-operation list of operation o. Notation $o.sol[i]$ is simplified as $o[i]$, and $|o.sol|$ as $|o|$. For an atomic operation o, its sub-operation list only includes itself or $|o|$=1. A composite operation o has more than one sub-operation or $|o| > 1$. Operations in $o.sol$ have the same id and $type$ properties (as well as timestamps) but different positions. We extend operation relations such as $\rightarrow$, $\|$, $\sqcup$, and $\mapsto$ to composite operations without re-definition.

Here we are only interested in sub-operations of deletions. For example, given string "abc", delete(0, "abc") can be denoted as a list of two sub-operations, delete(0, "a") and delete(1, "bc"). As another example, if for some reason we need to delete 'a' and 'c' in conceptually one operation, we

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8271
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8271

may define a composite operation with two sub-operations delete(0,'a') and delete(2,'c'). Note that positions of all operations in $o.sol$ are defined relative to the same state, dst(o). That is, they are contextually equivalent with regard to dst(o). Hence, to achieve the same effects as o, they must be applied simultaneously to dst(o). It would be wrong to apply them one after another like a sequence.

As will be shown in Sections 4.5 and 4.6, when a deletion is transformed with another insert or delete operation, the result could be a composite deletion with two or more sub-operations. Hence, the (delete) operations being propagated or received could be composite operations.

Algorithm 3 specifies the function for executing (atomic and composite) operations. In particular, to execute a composite operation o, we need to do a special transformation to $o.sol$ before applying the sub-operations in $o.sol$ in tandem. Algorithm 4 specifies this special transformation, called *selfIT*, which only adjusts positions of sub-operations that belong to the same composite operation o. Assume that an atomic operation's sub-operation list only includes itself. The algorithm first initializes sol with the given $o.sol$; then for every sub-operation $sol[i]$, subtract the total length of substrings deleted by preceding operations ranging from $sol[0]$ to $sol[i\text{-}1]$. As a result, every $sol[i]$ has accounted for the effects of preceding operations in the list. Then, operations in the resulting sol list (actually now a sequence) can be executed one by one in dst(o).

Algorithm 3 *execute*(o)

```
1: sol ← selfIT(o)
2: for (i=0; i < |sol|; i++) do
3:    apply sol[i] in shared data
4: end for
```

Algorithm 4 *selfIT*(o): sol

```
1: sol ← o.sol
2: if o.type = del and |o| > 1 then
3:    Δ ← |sol[0].str|
4:    for (i=1; i < |sol|; i++) do
5:       sol[i].pos ← sol[i].pos − Δ
6:       Δ ← Δ + |sol[i].str|
7:    end for
8: end if
9: return sol
```

Based on *selfIT*, we could define the following two simple utility functions: *getSubOpList(sq)* collects a list of sub-operations of all operations in a given sequence sq, after applying *selfIT* on every $sq[i]$; and function *combineSubOpList(ol)* returns a sequence of composite operations by combining all their sub-operations in a given list ol. These two functions are inverse of each other. For space reasons, we leave out their specifications in this paper.

4.5. IT Algorithms

In this subsection, we first discuss the most basic IT functions and then discuss advanced IT functions that involve at least one list (sequence) of primitive operations.

4.5.1 Basic IT Functions

In the most basic form, function IT(o_1, o_2) transforms a primitive operation o_1 with another primitive operation o_2 and outputs result o_1'. As will be shown shortly, the output result is sometimes a composite operation. By the types of the two involved operations, insert (I) and delete (D), we define four functions, ITII, ITID, ITDI, and ITDD, as in Algorithms 5–8, respectively. According to [10], the precondition of IT(o_1,o_2) is $o_1 \sqcup o_2$ and the postcondition is $o_2 \mapsto o_1'$. Intuitively, the positions of two operations must be defined in the same state so as to be compared in transformation. We will discuss the precondition further in Section 5.

Algorithm 5 *ITII*$(o_1, o_2) : o_1'$

```
1: o1' ← o1
2: if o2.pos < o1.pos then
3:    o1'.pos ← o1'.pos + |o2.str|
4: else if o2.pos = o1.pos and o2.id < o1.id then
5:    o1'.pos ← o1'.pos + |o2.str|
6: end if
7: return o1'
```

Algorithm 6 *ITID*$(o_1, o_2) : o_1'$

```
1: o1' ← o1
2: if o1.pos > o2.pos then
3:    if o1.pos ≥ o2.pos + |o2.str| then
4:       o1'.pos ← o1'.pos − |o2.str|
5:    else
6:       o1'.pos ← o2.pos
7:    end if
8: end if
9: return o1'
```

Algorithm 5 transforms insertion o_1 with another insertion o_2 to incorporate the effects of o_2. As shown in the condition of line 2, if $o_2.pos$ is on the left of $o_1.pos$, meaning $o_1.str$ is to be inserted after $o_2.str$ is inserted, then $o_1.pos$ should be shifted to the right by $|o_2.str|$ characters.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8271
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8271

If $o_1.pos$ and $o_2.pos$ tie, however, we use a priority scheme, e.g., by comparing their site ids, to break the tie: the two strings are ordered by their site ids in the result. That is, if $o_1.id$ is greater, $o_1.pos$ is shifted to the right. An intuitive scenario is that two users concurrently insert two strings at the same position in the same state. Using site ids to break the tie is a reasonable resort in concurrency control.

In Algorithm 6, an insertion o_1 is transformed with a deletion o_2. Let s be their common definition state. Since o_2 deletes a substring that is already in s and o_1 is to insert a new string into s, the deletion affects $o_1.pos$ only when $o_1.pos > o_2.pos$, as shown in the condition of line 2. There are two cases: $o_1.pos$ may fall out the right border of $o_2.str$ or within it. The condition in line 3 handles the former case, in which $o_1.pos$ is shifted to the left by $|o_2.str|$ characters. Otherwise, o_1 is to insert within a substring that is deleted by o_2. Note that, in this case, either policy is reasonable: keep $o_1.str$ or remove $o_1.str$. In this paper, we choose to keep $o_1.str$ in the result and hence reset its new position to be the same as that of o_2, as shown in lines 5-6.

Algorithm 7 $ITDI(o_1, o_2) : o_1'$

```
1:  o1' ← o1
2:  if o2.pos ≤ o1.pos then
3:      o1'.pos ← o1.pos + |o2.str|
4:  else if o1.pos < o2.pos < o1.pos + |o1.str| then
5:      oL ← oR ← o1
6:      oL.str ← o1.str[0 : o2.pos − o1.pos]
7:      oR.pos ← o2.pos + |o2.str|
8:      oR.str ← o1.str[o2.pos − o1.pos :]
9:      o1'.sol ← [oL, oR]
10: end if
11: return o1'
```

Algorithm 7 specifies how to transform a deletion o_1 with an insertion o_2. Let s be their common definition state. Since $o_1.str$ is already in s and o_2 inserts new content into s, we can use $o_1.str$ as the reference. As in the condition of line 2, if o_2 inserts on the left of $o_1.pos$, we need to shift $o_1.pos$ by $|o_2.str|$ characters to the right. However, if o_2 inserts within the substring that o_1 intends to delete, as in the condition of line 4, we need to split $o_1.str$ into two parts that are separated by $o_2.str$. As shown in lines 5-9, the transformation result is a composite operation with two sub-operations: o_L deletes the substring up to the position pointed to by $o_2.pos$ (exclusively), starting at position $o_1.pos$ in s; and o_R deletes the substring inclusively from after $o_2.pos$, starting at the position in s right after $o_2.str$, that is, $o_2.pos + |o_2.str|$.

As shown in Algorithm 8, transforming two deletions is more complicated. Both o_1 and o_2 are to delete an existing

Algorithm 8 $ITDD(o_1, o_2) : o_1'$

```
1:  o1' ← o1
2:  b1 ← o1.pos;  e1 ← o1.pos + |o1.str|
3:  b2 ← o2.pos;  e2 ← o2.pos + |o2.str|
4:  if b2 ≥ e1 then
5:      return o1'
6:  else if e2 ≤ b1 then
7:      o1'.pos ← b1 − |o2.str|
8:  else if b1 ≥ b2 and e1 ≤ e2 then
9:      return φ
10: else if b1 ≥ b2 and e1 > e2 then
11:     o1'.pos ← b2
12:     o1'.str ← o1.str[e2 − b1 :]
13: else if b1 < b2 and e1 ≤ e2 then
14:     o1'.str ← o1.str[0 : b2 − b1]
15: else if b1 < b2 and e2 < e1 then
16:     oL ← oR ← o1
17:     oL.str ← o1.str[0 : b2 − b1]
18:     oR.pos ← b2
19:     oL.str ← o1.str[e2 − b1 :]
20:     o1'.sol ← [oL, oR]
21: end if
22: return o1'
```

substring in their definition state s. We need to consider the following six cases regarding the relations between the two target regions, $R_1 = s[b_1 : e_1]$ and $R_2 = s[b_2 : e_2]$.

1. (line 4) R_2 is completely on the right of R_1. Deletion of R_2 does not affect o_1. Hence o_1 is returned as-is.
2. (line 6) R_1 is on the right of R_2. After R_2 is deleted, we shift $o_1.pos$ by $|o_2.str|$ characters to the left.
3. (line 8) R_1 is included in R_2. Hence after o_2 is executed, R_1 is already deleted. There is no longer need to execute o_1. We return an empty operation ϕ.
4. (line 10) R_2 partially overlaps with R_1 around the left border of R_1. After o_2 is executed, the left part of R_1 is already deleted. Hence, we need to reset $o_1.pos$ so that it will start from b_2. And $o_1.str$ only needs to include the right part that is not deleted by o_2, starting from $e_2 - b_1$ in the original $o_1.str$.
5. (line 13) R_2 partially overlaps with R_1 around the righ border of R_1. This case similar to case (4). After o_2 is executed, o_1 only needs to delete the left part that is not deleted by o_2.
6. (line 15) R_2 is included in R_1. This case is similar to the case of lines 4-10 in Algorithm 7. The deletion of R_2 within R_1 divides R_1 into three parts, among which the middle overlapping part is already deleted by o_2. Hence o_1 must be split into two sub-operations that delete the two remaining substrings, respectively.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8271
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8271

4.5.2 Sequence-Related IT Functions

Algorithm 9 *ITOSq*$(o, sq) : o'$

1: $o' \leftarrow o$
2: $ol \leftarrow getSubOpList(sq)$
3: $o'.sol \leftarrow ITLL(o'.sol, ol)$
4: return o'

Algorithm 10 *ITLL*$(ol_1, ol_2) : ol$

1: **if** $ol_1 = []$ or $ol_2 = []$ **then**
2: $ol \leftarrow ol_1$
3: **else**
4: $ol_H \leftarrow ITOL(ol_1.head, ol_2)$
5: $ol_T \leftarrow ITLL(ol_1.tail, ol_2)$
6: $ol \leftarrow ol_H \cdot ol_T$
7: **end if**
8: return ol

Algorithm 11 *ITOL*$(o, ol_2) : ol$

1: **if** $ol_2 = []$ **then**
2: $ol \leftarrow [o]$
3: **else**
4: $o' \leftarrow IT(o, ol_2.head)$
5: $ol \leftarrow ITLL(o'.sol, ol_2.tail)$
6: **end if**
7: return ol

Now we specify the two functions, *ITOSq* and *ITDsqI*, that are used in function *updateHR* (Algorithm 2). As shown in Algorithm 9, function ITOSq(*o*,*sq*) transforms an operation o with a sequence sq. To do that, we need to transform o one by one with every operation $sq[i]$. However, both o and $sq[i]$ could be composite. To simplify processing, we first collect all sub-operations of sq in list ol by calling function *getSubOpList(sq)*. Then, we use another algorithm to transform one list $o.sol$ with another list ol.

The algorithm to transform two lists is implemented by a double recursion of two functions, ITLL and ITOL. As shown in Algorithm 10, ITLL(ol_1, ol_2) transforms one list ol_1 with another list ol_2. In the simplest case, if ol_1 or ol_2 is empty, just return ol_1. Otherwise, we divide ol_1 into two parts, its first element $ol_1.head$ and the rest of the list $ol_1.tail$. Then we call function ITOL to transform operation $ol_1.head$ with list ol_2, yielding ol_H, and call function ITLL to transform list $ol_1.tail$ with list ol_2, yielding ol_T. Finally, we concatenate these two partial results, ol_H and ol_T, and return $ol_H \cdot ol_T$ as the result of ITLL(ol_1, ol_2).

In Algorithm 11, function ITOL(o, ol_2) transforms an operation with a list ol_2. Given a non-empty list ol_2, we need to transform o with operations in ol_2 one by one. Depending on types of the two involved operations, we first call some IT(o, $ol_2.head$) to transform o with the first operation in ol_2, yielding intermediate result o'; then we call ITLL($o'.sol$, $ol_2.tail$) to transform $o'.sol$ with the rest of list ol_2. Note that the intermediate list $o'.sol$ may be a singleton if the result is an atomic operation.

Algorithm 12 *ITDsqI*$(sq, o) : sq'$

1: $o_1 \leftarrow o$
2: $ol_1 \leftarrow getSubOpList(sq)$
3: $ol_2 \leftarrow []$
4: **for** (i=0; $i < |ol_1|$; i++) **do**
5: $o_2 \leftarrow o_1$
6: $o_1 \leftarrow ITID(o_1, ol_1[i])$
7: $o_d \leftarrow ITDI(ol_1[i], o_2)$
8: $ol_2 \leftarrow ol_2 \cdot (o_d.sol)$
9: **end for**
10: $sq' \leftarrow combineSubOpList(ol_2)$
11: return sq'

Next, we specify function *ITDsqI(sq, o)* for transforming a sequence sq with an operation o to incorporate the effects of o into every operation in sq, as shown in Algorithm 12. Since *ITDsqI* is only called in Algorithm 2, we know that its input sq is deletion-only and o is but an insertion. A pitfall in implementing *ITDsqI(sq,o)* is to naively transform every operation in sq with o one by one. The input precondition is $sq \sqcup o$, or $sq[0] \sqcup o$. Hence it makes sense to do IT($sq[0], o$). However, for the next operation, $sq[1]$, the relation is not $sq[1] \sqcup o$. Hence it makes no sense to do IT($sq[1], o$). The fix is to first transform o with $sq[0]$, yielding o', and then transform $sq[1]$ with o'. Following this idea, we first collect all sub-operations of sq into list ol_1; for every operation in ol_1, we transform o with $ol_1[i]$, and then transform $ol_1[i]$ with o, yielding o_d. All sub-operations in o_d are collected in list ol_2. Finally, we combine the sub-operations and return a sequence of composite operations sq'.

4.6. SWAP Algorithms

We first present the basic swap functions for swapping two primitive operations, and then discuss advanced swap functions that involve sequences of operations.

4.6.1 Basic swap Functions

Given two operations o_1 and o_2, where $o_1 \mapsto o_2$, function *swap(o_1, o_2)* transposes them into o_1' and o_2' such that $o_2' \mapsto o_1'$. Depending on their types, insert (I) and delete

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8271
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8271

Algorithm 13 *swapDI*$(o_1, o_2) : (o_2', o_1')$

1: $o_1' \leftarrow o_1$; $o_2' \leftarrow o_2$
2: **if** $o_2.pos \geq o_1.pos$ **then**
3: $o_2'.pos \leftarrow o_2'.pos + |o_1.str|$
4: **else**
5: $o_1'.pos \leftarrow o_1'.pos + |o_2.str|$
6: **end if**
7: return (o_2', o_1')

Algorithm 14 *swapDD*$(o_1, o_2) : (o_2', o_1')$

1: $o_1' \leftarrow o_1$; $o_2' \leftarrow o_2$
2: **if** $o_2.pos \geq o_1.pos$ **then**
3: $o_2'.pos \leftarrow o_2.pos + |o_1.str|$
4: **else if** $o_2.pos + |o_2.str| \leq o_1.pos$ **then**
5: $o_1'.pos \leftarrow o_1'.pos - |o_2.str|$
6: **else**
7: $o_{2L} \leftarrow o_{2R} \leftarrow o_2$
8: $o_{2L}.str \leftarrow o_2.str[0 : o_1.pos - o_2.pos]$
9: $o_{2R}.pos \leftarrow o_1.pos + |o_1.str|$
10: $o_{2R}.str \leftarrow o_2.str[o_1.pos - o_2.pos :]$
11: $o_2'.sol \leftarrow [o_{2L}, o_{2R}]$
12: $o_1'.pos \leftarrow o_2.pos$
13: **end if**
14: return (o_2', o_1')

(D), we specify two basic swapping functions as in Algorithms 13 and 14, respectively. Function *swapII* is used in function *transposeHC* (called in Algorithm 2) and omitted altogether for space reasons. Function *swapID* is irrelevant because it is not used in our ABTS algorithm at all.

Algorithm 13 swaps a deletion o_1 and an insertion o_2. The case of $o_2.pos \geq o_1.pos$ means that, after o_1 deletes $o_1.str$, o_2 inserts $o_2.str$ on the right side of the region $o_1.str$ originally occupies. Hence, if they are swapped, meaning that o_2 inserts $o_2.str$ before o_1 deletes $o_1.str$, then $o_2.pos$ should be shifted by $|o_1.str|$ characters to the right because $o_1.str$ is not deleted yet. The case of $o_2.pos < o_1.pos$ means that, after o_1 deletes $o_1.str$, o_2 inserts $o_2.str$ on the strict left of $o_1.pos$ and there are at least one character between $o_2.pos$ and $o_1.pos$. Hence swapping the execution order of o_1 and o_2 entails that $o_1.pos$ be shifted by $|o_2.str|$ characters to the right to account for $o_2.str$.

As specified in Algorithm 14, function *swapDD*(o_1, o_2) transposes two deletions o_1 and o_2. There are three cases to consider: First, if $o_2.pos \geq o_1.pos$, it means that o_2 is to delete a substring on the right side of the substring $o_1.str$ deleted by o_1. Hence, if we execute o_2 before o_1 instead, then $o_2.pos$ should account for $o_1.str$ because it has not been deleted yet. Secondly, if $o_2.pos + |o_2.str| \leq o_1.pos$, it means that $o_2.str$ is completely on the left side of $o_1.pos$. Hence, if we execute o_2 before o_1 instead, $o_1.pos$ should be shifted to the left because $o_2.str$ has been deleted. Thirdly, as in lines 6-12, $o_1.str$ is completely covered by $o_2.str$. Then, if we execute o_2 before o_1 instead, $o_2.str$ is divided into three parts, among which the middle part is to be deleted by o_1. The remaining left and right parts, as divided by position $o_1.pos$, are deleted by two sub-operations o_{2L} and o_{2R}, respectively. Finally, $o_1.pos$ should be set to $o_2.pos$ due to the deletion of $o_{2L}.str$.

4.6.2 Sequence-Related swap Functions

Algorithm 15 *swapDsqI*$(sq, o) : (o', sq')$

1: $o' \leftarrow o$
2: $ol \leftarrow getSubOpList(sq)$
3: **for** $(i = |ol| - 1; i \geq 0; i - -)$ **do**
4: $(o', ol[i]) \leftarrow swapDI(ol[i], o')$
5: **end for**
6: $sq' \leftarrow combineSubOpList(ol)$
7: return (o', sq')

Algorithm 16 *swapDsqD*$(sq, o) : (o', sq')$

1: $o' \leftarrow o$
2: $ol \leftarrow getSubOpList(sq)$
3: $(o'.sol, ol) \leftarrow swapLL(ol, o'.sol, |ol| - 1)$
4: $sq' \leftarrow combineSubOpList(ol)$
5: return (o', sq')

As in Algorithm 15, function *swapDsqI(sq, o)* transposes a deletion sequence sq with an insertion o, where $sq \mapsto o$, into sq' and o' such that $o' \mapsto sq'$. We first flatten sq by collecting all sub-operations of sq in list ol. Then we call the specified function *swapDI* to transpose every operation in ol with o from right to left. Finally we merge all sub-operations in ol and return the resulting sequence as sq'.

As in Algorithm 16, function *swapDsqD(sq,o)* transposes a deletion sequence sq and a deletion o. The process is more complicated because swapping two deletions may result in composite operations. Similarly to Algorithm 14, we first flatten sq into list ol, then call function *swapLL* to transpose list ol and list $o.sol$, and finally combine sub-operations in the resulting list. When calling swapLL, the third parameter is the index indicating from which operation in list ol we start the actual swapping.

The algorithm for transposing two lists is implemented by a double recursion of two functions, *swapLL* and *swapOL*, as specified in Algorithm 17 and 18, respectively. Function *swapLL*(ol_1, ol_2, p) transposes two given list ol_1 and ol_2, where $ol_1 \mapsto ol_2$, into ol_1' and ol_2' such that

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8271
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8271

Algorithm 17 $swapLL(ol_1, ol_2, p) : (ol'_2, ol'_1)$

1: **if** $ol_1 = [\,]$ or $ol_2 = [\,]$ **then**
2: return (ol_2, ol_1)
3: **else**
4: $ol'_1 \leftarrow ol_1$
5: $(ol_{2H}, ol'_1) \leftarrow swapLO(ol'_1, ol_2.head, p)$
6: $(ol_{2T}, ol'_1) \leftarrow swapLL(ol'_1, ol_2.tail, p)$
7: $ol'_2 \leftarrow ol_{2H} \cdot ol_{2T}$
8: return (ol'_2, ol'_1)
9: **end if**

Algorithm 18 $swapLO(ol_1, o, p) : (ol'_2, ol'_1)$

1: **if** $p = 0$ **then**
2: $(o', ol'_1[0]) \leftarrow swapDD(ol_1[0], o)$
3: return $(o'.sol, ol'_1)$
4: **else**
5: $ol'_1 \leftarrow ol_1$
6: $(o', ol'_1[p]) \leftarrow swapDD(ol'_1[p], o)$
7: $(ol'_2, ol'_1) \leftarrow swapLL(ol'_1, o'.sol, p-1)$
8: return (ol'_2, ol'_1)
9: **end if**

$ol'_2 \mapsto ol'_1$. Due to Algorithm 16, the input ol_2 is a sub-operation list. That is, operations in ol_2 are defined relative to the same state. Hence the ordering of operations in ol_2 is not important. Function swapLL works as follows: First, we call swapLO(ol'_1, $ol_2.head$, p) to transpose ol'_1 with the first operation in ol_2, where ol'_1 is a copy of ol_1. While ol'_1 is transformed in place, operation $ol_2.head$ is transformed into list ol_{2H}. Then, we recursively call function swapLL(ol'_1, $ol_2.tail$, p) to transpose ol'_1 with the remaining operations in ol_2. Again, while ol'_1 is transformed in place, list $ol_2.tail$ is transformed into list ol_{2T}. Finally, we concatenate partial results ol_{2H} with ol_{2T} into ol'_2 and return tuple (ol'_2, ol'_1). Because operations in ol_2 are all contextually equivalent, operations in the resulting ol'_2 are also contextually equivalent.

Function swapLO(ol_1, o, p) transposes list ol_1 and operation o, where $ol_1 \mapsto o$, into list ol'_1 and list ol'_2, respectively, such that $ol'_2 \mapsto ol'_1$. Parameter p points to the current operation in ol_1 to be transposed with o. If p is zero, we call swapDD to transpose $ol_1[0]$ with o, yielding $ol'_1[0]$ and o', and return the resulting $o'.sol$ and ol'_1. Otherwise, we transpose every operation in ol_1 with o from right to left, as in lines 5-8. First, we call swapDD($ol'_1[p], o$) to transpose the last operation in ol'_1 and o, transforming o into o'. Note that ol'_1 is a copy of ol_1 and all its operations are transformed in place. As a result of swapDD, o' could be a composite operation. Hence we call swapLL($ol'_1, o'.sol, p-1$) to transpose ol'_1 with $o'.sol$, which transforms $o'.sol$ into ol'_2.

5. Analysis of Correctness

ABTS is based on a well-proved theoretical framework called admissibility-based transformation (ABT) [6, 5], which establishes that an OT algorithm is correct if the following two formal conditions always hold:

(1) **Causality preservation:** whenever an operation o is executed at a site, all operations that happen before o must have been executed at that site.
(2) **Admissibility preservation:** the execution of every operation is admissible, i.e., it does not introduce inconsistent ordering of objects at different sites.

Condition (1) is satisfied by using vector timestamps. To satisfy condition (2), our approach is to first establish sufficient conditions of the basic IT and swap functions and then design a control procedure that satisfies those sufficient conditions while integrating local and remote operations. These two conditions together imply convergence [6, 5].

According to [11], the precondition of IT(o_1, o_2) is $o_1 \sqcup o_2$, and the precondition of swap(o_1, o_2) is $o_1 \mapsto o_2$. In [6, 5], they are extended with the following proved sufficient conditions for the basic IT and swap functions to produce the correct results, i.e., admissible operations, assuming that o_1 and o_2 are admissible:

(a) IT(o_1, o_2) is admissible if $o_1 \sqcup o_2$ and, in the case they are both insertions and their positions tie, $o_1 \parallel o_2$ and neither o_1 nor o_2 includes effects of any deletions.
(b) swap(o_1, o_2) is admissible if $o_1 \mapsto o_2$ and, in the case o_1 is a deletion and o_2 is an insertion and their positions tie, $o_1 \rightarrow o_2$ and o_2 is generated in state dst(o_2).

The presented ABTS algorithm has mainly two parts, updateHL and updateHR. The correctness of updateHL(o) for integrating a local operation o (Algorithm 1) is ensured as follows: when o is a deletion, it is only swapped with deletions in H_d. At every step (Algorithms 16- 18), as long as $o_1 \mapsto o_2$ is guaranteed for every swapDD(o_1, o_2), the result is correct. By discussions in Section 4.6, this is ensured. On the other hand, when o is an insertion, it is swapped with deletions in H_d. However, since o is a local operation that happens after H_d and generated in its definition state, the above condition (b) holds for every swapDI function called. Hence, updateHL(o) is correct.

The correctness of updateHR(o) for integrating a remote operation (Algorithm 2) is ensured as follows: In line 1, function transposeHC [10, 6] transposes an insertion-only sequence H_i, which ultimately calls basic function swapII. By the above condition (b), the result is correct as long as $o_1 \mapsto o_2$ is ensured every time swapII(o_1, o_2) is called. In line 2, the result of ITOSq is correct by the above condition (a) because, although IT happens between two concurrent insertions, neither of them includes effects of any deletions as a consequence of updateHL. In line 3, the result is correct as long as two operations involved in every IT are contextu-

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8271
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8271

ally equivalent. Similarly the result of line 5 is also correct. Hence, updateHR(o) is also correct.

6. Analysis of Complexities

Note that the double-recursion presentation of Algorithms 9 and 16 is only for the sake of conceptual clarity. In the actual system, we rewrite them in a more efficient way to avoid the runtime overheads of recursions.

The space complexity of the presented ABTS algorithm is trivially $O(|H|)$. The time complexity of ABTS is in the same order of magnitude as that of its characterwise version ABT [6, 3, 5], which is not counterintuitive.

For space reasons, here we only give the results: The time complexity to integrate a local operation is $o(|ol_d| \cdot |o.str|)$, where ol_d is the corresponding sub-operation list of H_d. When $m = |o.str|$ and the average length of deleted strings in the history, $c = |ol_d|/|H_d|$ can be considered as small constants, the complexity is linear in the number of deletions in the history, i.e., $O(|H_d|)$. The execution of a remote operation o takes time $O(|H_i|^2 + (|sq_c| + |ol_d|) \cdot |o.str|)$, where sq_c is the operations in H_i that are concurrent with o and ol_d is the corresponding sub-operation list of H_d. When $m = |o.str|$ and $c = |ol_d|/|H_d|$ can be considered as small constants, the complexity is $O(|H_i|^2 + |H_d|)$, roughly quadratic in the number of insertions in the history.

7. Conclusions

This paper presents a novel transformation based consistency control algorithm called ABTS that supports string-based primitive operations. The presented algorithm is the first of its kind with stringwise operations and correctness formally proved. For space reasons, we only sketched the correctness proofs and the complexity analyses in this paper. Since operations are stored in their execution order in the history H, the time complexity to integrate a remote operation is roughly $O(|H|^2)$, which is in the same order of magnitude as its character-based precursor, ABT [6, 5]. Although ABTS is extended from ABT, the extension is theoretically significant due to the complications in handling operation region overlapping and splitting. Moreover, the extension makes it possible to apply OT techniques to a wider range of practical collaborative applications.

In future research, we plan to extend this work specifically for application domains such as collaborative software development and study its usability. With support of string operations as the new starting point, it will be interesting to study techniques for conflicts detection and resolution in the context of specific application domains [11]. Another interesting direction is to optimize the algorithm to reduce the time complexity for it to work more efficiently for both real-time and asynchronous collaborative applications [3].

Acknowledgments

The work is supported in part by the National Natural Science Foundation of China (NSFC) under Grant No. 60736020 and No. 60803118, the Shanghai Science & Technology Committee Key Fundamental Research Project under Grant No. 08JC1402700 and the Shanghai Leading Academic Discipline Project under Grant No. B114.

References

[1] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD'89 Conference on Management of Data*, pages 399–407, Portland Oregon, 1989.

[2] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

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

[5] D. Li and R. Li. An admissibility-based operational transformation framework for collaborative editing systems. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, Aug. 2009. Accepted.

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

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

[8] G. Oster, P. Urso, P. Molli, and A. Imine. Proving correctness of transformation functions in collaborative editing systems. Technical Report 5795, INRIA, Dec. 2005.

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

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

[11] 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 Transactions on Computer-Human Interaction*, 5(1):63–108, Mar. 1998.

[12] D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, 2009.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8271
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8271