A Sequence Transformation Algorithm for Supporting Cooperative Work on Mobile Devices

Bin Shao
Fudan University
Shanghai, China
binshao@fudan.edu.cn

Du Li
Nokia Research Center
Palo Alto, CA, USA
lidu008@gmail.com

Ning Gu
Fudan University
Shanghai, China
ninggu@fudan.edu.cn

ABSTRACT

Operational transformation (OT) is a promising technique for supporting collaboration using mobile devices because it allows users to work on local data replicas even in a disconnected mode. However, as work goes mobile, a large number of operations may accumulate, defying the capacity of current OT algorithms that are mostly designed for real-time group editing. Since their assumption is that operations are propagated frequently, they generally only address how to integrate one remote operation at a time. As a consequence, most algorithms take $O(|H|^2)$ to integrate one operation and thus $O(|H|^3)$ to integrate a long sequence, where H is the operation history. This paper proposes a novel algorithm that provides optimized transformation of long sequences, improving the time complexity to $O(|H|)$. Our experiments will show that it takes 59 minutes in a recent algorithm versus 1.5 seconds in this work to integrate two long sequences on a mobile device. The performance improvement is critical towards achieving desired responsiveness and group productivity in a class of mobile collaborative applications.

Author Keywords

Collaborative Applications, Data Consistency, Group Editing, Mobile Computing, Operational Transformation

ACM Classification Keywords

H.5.3 Information Systems: Group and Organization Interfaces—*Asynchronous Interaction, Computer Supported Cooperative Work, Collaborative Computing*

General Terms

Algorithms, Experimentation, Human Factors, Performance

1. INTRODUCTION

Mobile devices such as laptops, netbooks, PDAs, tablets and cell phones are becoming pervasive in recent years. They typically operate on batteries and wireless networks for mobility and their connections are usually intermittent. These characteristics pose significant challenges on the design of systems that support collaboration using mobile devices [8]. One of the major challenges is data consistency. For performance reasons, optimistic consistency control techniques [17] are widely used: The shared data is replicated so that the users are able to work on their local data replicas even when disconnected and sync with each other when reconnected.

Operational transformation (OT) [3, 22] is an optimistic consistency control technique that has been well-established in CSCW applications. It allows a distributed group of users to simultaneously edit shared documents, e.g., Web pages, software source code, and even music scores [1]. The technique lies in the heart of many recent products, including CoWord [24], ACE, Gobby, SubEthaEdit, and the newly-released Google Wave (which runs on mobile devices). [1]

However, most existing OT algorithms are designed for supporting real-time group editing in which operations are frequently propagated and integrated. As a result, they focus on how to transform one remote operation at a time. To the best of our knowledge, most algorithms take $O(|H|^2)$ to integrate one remote operation and hence $O(|H|^3)$ to integrate a long sequence, where H is the local operation history.

An algorithm with time complexity $O(|H|^3)$ would be suboptimal on mobile devices. In mobile collaboration, a large number of operations may accumulate during a disconnected mode. Hence the algorithm must be able to efficiently integrate long sequences of editing operations. Individual and group productivity would be very low if it had to take minutes or even hours every time a user presses a button to sync with her collaborators after an episode of disconnected work.

This paper presents a novel OT-based algorithm for supporting mobile collaboration. As a major contribution, the algorithm provides optimized handling of long sequences and improves the time complexity to $O(|H|)$. Our experiments show that, for example, it takes *59 minutes* in a recent algorithm versus *1.5 seconds* in this work on a mobile device to integrate two sequences each of 3,000 operations. This efficiency improvement is achieved primarily by maintaining operations in each of the involved sequences in the order of the so-called operation effects relation [14]. As a result, the key steps in integrating any two sequences can all be achieved in linear time. The algorithm is called admissibility-based sequence transformation or ABST be-

CSCW 2010, February 6–10, 2010, Savannah, Georgia, USA.

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

cause it is based on the admissibility theory [14, 11] that proposes alternative solutions to some well-known correctness problems in previous work.

The next section analyzes related works to establish the context of research. After that, we introduce the basic concepts and notations. The next two sections will present the ABST algorithm, which is followed by an example to illustrate how the algorithm works. Then we discuss its complexities and performance experiments. Finally, we summarize the contributions and limitations of this work and future directions.

2. RELATED WORK

Pessimistic protocols such as locking and floor control are counterproductive for human users because they generally have to wait for their turns [6]. Traditional optimistic techniques [17] may suffer from loss of work since only one write can eventually "win" among concurrent writes that conflict, i.e., target the same object [4]. Therefore, they are usually used in applications in which concurrent updates rarely conflict, e.g., distributed file systems [7]. Version control systems such as SVN and GIT are asynchronous and work at line granularity: two concurrent changes conflict if they target the same line in a textual document.

OT is an optimistic technique motivated specifically for concurrent editing of a shared document [3]. Among the most differentiating properties of OT, in general, two primitives, insert and delete, are used in place of write and all updates can be transformed to commute. This avoids losing interaction results and relying on the same total order of execution to achieve convergence. This property also makes it possible to use OT for supporting fine-grained updates and merging.

A plethora of OT algorithms have been proposed under the framework of [23]. Most of them are designed for real-time group editors which assume frequent propagation and integration of operations. Under this assumption, the operation history (H) is usually kept short enough to warrant efficiency [23]. Hence, even though operations are usually propagated in batches [13], they only address how to integrate one remote operation at a time. In general, they keep operations in the history H in their order of execution. Most of them (e.g., [9, 21, 22]) adopt a similar control procedure: when a remote operation o is integrated, they transpose the entire history such that those operations that happened before o precede those that are concurrent with o; then o is transformed with those concurrent operations.

According to analyses in [10], the above transposition step alone takes $O(|H|^2)$ time. Each time one operation in a remote sequence T is integrated, it is added to H, increasing its size by one. The time to integrate T is $O(\sum_{k=|H|}^{|H|+|T|-1} k^2)$, which is $O(|H|^3)$ when $|H|$ and $|T|$ are comparable. Although it may not cause performance problems in real-time group editors, the complexity would be too high on mobile devices because of their limited processing power and intermittent connectivity. The sequences could grow fast enough to render the algorithm too slow to achieve interactivity. Hence there is a room for improvements.

To the best of our knowledge, except for [12, 19], no other OT-based (e.g., [9, 21, 22, 25]) or non-OT (e.g., [5, 16]) algorithms proposed for collaborative editing systems have addressed how to integrate and transform sequences.

Shen and Sun [19] keep operations in sequences in their execution order. When transforming two sequences, it takes time quadratic in the size of the two sequences. They achieve convergence by making use of a central notification server. Nonetheless, we acknowledge that their proposal to compress the operation sequences is complementary to our work and could be leveraged in our future research.

Li and Lu [12] is also able to merge in linear time two operation sequences (edit scripts that are obtained by diffing). However, their focus is on transparent sharing of familiar single-user editors. Hence, their algorithm is simplified such that every time two sequences are transformed they must be defined in the same state. This means that a user cannot make new edits until synchronized with all other users, which is a strong assumption in mobile environments. The presented ABST algorithm could be used in the system of [12] for improved flexibility in synchronization.

Admissibility-Based Transformation (ABT) [14, 11] is our latest theoretical framework under which OT algorithms can be formally proved. Conceptually, admissibility requires that every operation be executed such that its effect does not contradict the object order established by previous executions. We acknowledge the existence of different understandings of the correctness of OT algorithms, e.g., as in [23, 25]. By our understanding [9, 15, 14, 11], however, ABT is the state-of-the-art because it is formally proved and fully presented; we prefer to compare the performance of ABST with ABT, by no means implying that the other works are incorrect.

To achieve admissibility, ABT maintains history H as the concatenation of an insertion subsequence (H_i) and a deletion subsequence (H_d) with operations in execution order in either subsequence. As a result, when a remote operation is integrated, it is transformed with insertions before deletions, avoiding the insertion-tie problems [9, 21, 23]. Because only subsequence H_i is transposed, it takes $O(|H_i|^2 + |H_d|)$ to integrate a remote operation, which outperforms other OT algorithms with time complexity $O(|H|^2)$ or higher. Nevertheless, the time complexity of ABT is still $O(|H|^3)$ when integrating a long sequence with size comparable to $|H|$.

ABST follows the same theory as ABT and hence its correctness can be formally proved [18]. The key concept in ABT, namely, the *effects relation*, is leveraged in this work to order operations by the relative position of their effects. The main differences between ABT and ABST are that (1) ABST propagates and integrates operations in units of transactional sequences, and (2) ABST keeps the histories and sequences ordered by the effects relation. As a result, by exploiting this ordering property, time complexities of all the key steps in ABST can be achieved in linear time. All the functions presented in this paper are new in ABST because they are designed for efficient handling of sequences.

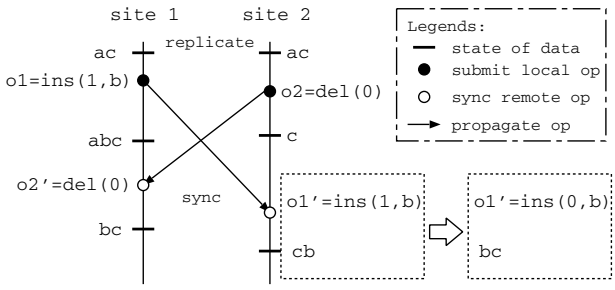

Figure 1. With operational transformation, concurrent operations are transformed to commute at sync time without overwriting each other.

3. BACKGROUND: CONCEPTS AND NOTATIONS

The basic idea of OT is illustrated as in Figure 1. Suppose that a file containing string "ac" is replicated at two sites. Site 1 performs operation o_1 = ins(1, b) which inserts 'b' at position 1 to change the content to "abc". In parallel, site 2 performs o_2 = del(0) to delete the object 'a' at position 0 to change the content to "c". These two operations are propagated to all sites. At sync time, site 1 needs to integrate remote operation o_2 with local operation o_1. By inspecting their positions, o_1 inserts on the right side of where o_2 deletes. Hence the position of o_2 is not affected and can be executed as-is in state "abc", which results in content "bc".

On the other hand, at site 2, if o_1 were executed as-is in current state "c", the wrong content "cb" would be yielded. This is because the execution of o_2 on the left of where o_1 inserts has invalidated the position of o_1. Hence o_1 needs to be transformed (i.e., by shifting its position) into o_1' = ins(0,b) such that it can be correctly executed in current state of site 2. As a result, site 2's content becomes "bc" and the two replicas converge despite different orders of execution.

In many traditional approaches, the same *total order of operations* is enforced at all sites such that the replicas converge in the same state. Different total orders usually lead to different results. In Figure 1, for instance, if the total order is that o_1 precedes o_2, site 2 must first undo o_2 and then execute o_1 and o_2 in order; if the total order is o_2 precedes o_1, on the other hand, site 1 must first undo o_1 and then execute o_2 followed by o_1. Although the former total order produces the right result "bc", the latter yields the wrong result "cb" that violates the intention of o_1 to insert 'b' before 'c' [23].

In general, traditional systems focus on convergence and do not address which *operation order* is "right". OT theories require that the execution of operations preserve operation intentions [23] or admissibility [14, 11] in addition to convergence. In particular, admissibility ensures that the execution of any operation does not violate the *order of objects* established by previous executions. As a result, when all operations have been executed, objects are consistently ordered at all sites and this final order is consistent with the original object order established when the operations are generated. In spirit, the idea of admissibility resembles that of intention preservation, except that it is a formal condition that can be proved. As long as the effects relation is always preserved (i.e., admissibility) and causality preserved, data replicas in the system eventually converge in the right state.

By the admissibility theory [14, 11], any two executed operations can be unambiguously ordered by their *effects relation* $\prec$, i.e., the relative position of the objects they insert or delete. For example, in the scenario of Figure 1, the operations can be ordered as $o_2 \prec o_1$ because the object o_2 deletes ('a') appears left to the object o_1 inserts ('b') in one of the legal states that ever appear in the system. When tie situations arise, e.g., two concurrent insertions target the same position, tie-breaking policies are used in the specific algorithm as long as they do not cause inconsistent ordering of objects. The resulting effects relation is deterministic.

In scenarios like the above, concurrent execution of operations easily leads to inconsistencies. Hence, when transforming two operations, we need to determine their temporal relationship, e.g., whether or not they are concurrent. As a convention [3], we use vector timestamps for determining the happens-before ($\rightarrow$) and concurrent ($\|$) relations. In the above scenario, we have relation $o_1 \parallel o_2$.

To simplify discussions, we model the shared data as a string and consider two primitive operations, ins(p, c) and del(p), which insert and delete one object at position p, respectively. For any operation o, attribute $o.v$ is the vector timestamp; $o.id$ is the id of the site that originally generates it; $o.type$ is either ins or del; $o.pos$ is the target position; and $o.c$ is the effect object being inserted or deleted.

Theoretically, an update operation that modifies an attribute of an object could be represented as a deletion of the original object followed by an insertion of the object with the new attribute. Note that $o.pos$ is an integer since the shared data is modeled as a linear string. According to [2], we can denote $o.pos$ as a vector of integers for a tree structure, with little impact on the algorithms. Further, according to [24], even operations in sophisticated office and design software can be mapped to a linear or tree-like data model. Hence our model does not lose generality although simplified.

In addition, note that $o.pos$ is always relative to a state of the document. Borrowing established notations from [22], if two operations are defined in the same state, they are said to be contextually equivalent ($\sqcup$); if one is defined in the state resulted from executing another operation, they are said to be contextually serialized ($\mapsto$). In Figure 1, for instance, we have $o_1 \sqcup o_2$, $o_1 \mapsto o_2'$, and $o_2 \mapsto o_1'$.

An operation sequence $sq = [o_1, o_2, ..., o_n]$ is such that all operations in sq are contextually serialized. That is, $o_1 \mapsto o_2 ... \mapsto o_n$. The number of operations in sq is denoted as $|sq|$. Notation $sq[i]$ refers to the operation in position i of sq, where $0 \leq i < |sq|$. For example, $sq[0] = o_1$ and $sq[1] = o_2$. If two sequences sq_1 and sq_2 are contextual serialized, denoted as $sq_1 \mapsto sq_2$, we can concatenate them into a new sequence $sq = sq_1 \cdot sq_2$. Generalizing notations $\mapsto$ and $\cdot$, for an operation o and a sequence sq, if $sq \mapsto o$, we can append o to sq obtaining a new sequence sq' = $sq \cdot o$.

4. SEQUENCE TRANSFORMATION ALGORITHM

A history buffer H is maintained at each site which records all operations that have been executed at that site, including local and remote operations. ABST propagates and integrates a sequence of operations at a time as if it is a "transaction". In the sequences and the history, insertions and deletions are kept in separate subsequences to achieve correctness, and each subsequence is ordered by the effects relation to achieve efficiency. The time *complexity* is as low as $O(|H|)$ although the algorithm may look *complicated*.

In this section, we first introduce the concept of "transactional sequence"; then we give the control algorithm for integrating a remote transactional sequence; after that, we elaborate the functions that are called in the integration algorithm. In the next section, we will present how to obtain a local transactional sequence before it is propagated.

4.1. Transactional Sequence

From the user's perspective, a sequence T can be considered as a "transaction". We call it a *transactional sequence*, which conceptually has two aspects: On one hand, the user performs a batch of editing operations locally first and then propagates them to remote sites when she feels "ready" for sharing. During a local transaction, no remote operation is integrated so as not to distract the user from her own work. This way, a sequence T of operations is accumulated and propagated in its entirety to remote sites. On the other hand, at sync time, a number of sequences could have been received from remote sites. Those sequences are kept in a receiving queue, RQ. When the user feels "ready" for merging, each time a sequence T from RQ is integrated into H after transformation. The transformed sequence T' is then applied to the local data replica in its entirety.

A transactional sequence has the following properties: All operations in a given transactional sequence T have the same site id; their vector timestamps are equivalent in all corresponding elements except for the one representing that site. Hence we can use $T.id$ for the site id, and $T.type$ for the type if all operations in T are of the same type. It can be shown that, for any operation o that is not in T, if o is concurrent with any $T[i]$, then o must be concurrent with any other $T[j]$, where $0 \leq i, j < |T|$. Similarly, for any operation o' that is not in T, if it happens before any $T[i]$, it must also happen before any other $T[j]$. For convenience, we denote the relations as $o \parallel T$ and $o' \rightarrow T$, respectively.

4.2. Integrate One Remote Sequence

We maintain the history H at each site as $H_i \cdot H_d$, where H_i is a sequence of all insertions and H_d is a sequence of all deletions in H. Furthermore, operations in each of these two subsequences are ordered by the effects relation $\prec$. Similarly, a transactional sequence T is maintained as $T_i \cdot T_d$, where T_i and T_d are sequences of insertions and deletions that are ordered by the effects relation, respectively. History H is initially empty and grown incrementally as local and remote transactions are integrated. Sequences H and T are so ordered for correctness and efficiency reasons.

Algorithm 1 Integrate(T, H): (T', H')

1: $csq_i \leftarrow \text{getConcurrentSQ}(T_i, H_i)$
2: $T_i'' \leftarrow \text{ITSQ}(T_i, csq_i)$
3: $T_i' \leftarrow \text{ITSQ}(T_i'', H_d)$
4: $H_i' \leftarrow \text{mergeSQ}(H_i, T_i'')$
5: $H_d'' \leftarrow \text{ITSQ}(H_d, T_i'')$
6: $csq_d \leftarrow \text{getConcurrentSQ}(T_d, H_i')$
7: $T_d'' \leftarrow \text{ITSQ}(T_d, csq_d)$
8: $T_d' \leftarrow \text{ITSQ}(T_d'', H_d'')$
9: $H_d' \leftarrow \text{mergeSQ}(H_d'', T_d')$
10: return $(T_i' \cdot T_d',\ H_i' \cdot H_d')$

Let H^k be site k's history. When a sequence T is generated at site x, $H^x \rightarrow T$ must hold. When T is integrated at site y, it must be *causally-ready* at site y, i.e., all operations that happen before T have been executed at site y and included in H^y. Before $T = T_i \cdot T_d$ is propagated at site x, it must have been transformed with H^x such that $H_i^x \mapsto T \mapsto H_d^x$. That is, T does not include the effects of any deletions that happen before T.

Algorithm 1 shows how a remote sequence T is integrated with local history H. In the output, T' is the result of transforming T with H; H' already includes all operations in T; and the same ordering properties are maintained in H' and T'. After function Integrate returns, the resulting T' is ready to be executed in the current state of the local data replica.

The essential idea of integrating a remote operation o, is to incorporate the effects of operations that have been executed locally (already in H) but not yet included in o. The execution of those operations has led to current state of the data but may have invalidated the position of o relative to the current state. The process of incorporating effects of one operation into another is called inclusion transformation or IT [23].

The same idea applies when integrating a remote sequence T. When Algorithm 1 is called, T is causally ready, i.e., any operation in H must either happen before T or be concurrent with T. In addition, T has excluded the effects of all deletions that happen before T in its originating site. We know that H is maintained as $H_i \cdot H_d$, or an insertion sequence H_i concatenated by a deletion sequence H_d. Hence T includes effects of some of the operations in H_i (that happened before T) but none of the operations in H_d. By the spirit of OT, we need to incorporate the effects of all insert operations in H_i that are concurrent with T and all delete operations in H_d.

Accordingly, Algorithm 1 consists of two parts: lines 1-5 integrates T_i and lines 6-9 integrates T_d. We explain these two parts as follows: First, to integrate T_i, we need to extract all operations in H_i that are concurrent with T_i, resulting in sequence csq_i. This is achieved by function getConcurrentSQ. Next, we call function ITSQ to incorporate the effects of csq_i into T_i, resulting in T_i''. Then, we incorporate the effects of H_d into T_i'', yielding T_i'. After that, we call function mergeSQ to merge T_i'' into H_i, yielding H_i'. Now all insertions in T are integrated into H.

Algorithm 2 getConcurrentSQ(T, isq) : csq

```
1: csq ← [ ]
2: for (j ← 0; j < |isq|; j++) do
3:    if (isq[j] ∥ T) then
4:       csq = csq · isq[j]
5:    end if
6: end for
7: return csq
```

Note, however, that the last step to merge T_i'' into H_i invalidates the definition state of H_d. This is because H_d is defined in the state obtained by executing H_i, i.e., $H_i \mapsto H_d$. After the merging, property $H_i' \mapsto H_d$ no longer holds due to the effects of T_i''. Hence we need to transform H_d by calling function ITSQ to incorporate the effects of T_i'', yielding H_d''. Now $H_i' \mapsto H_d''$ holds.

Second, to integrate T_d, we first extract all operations in H_i' that are concurrent with T_d, resulting in csq_d. Note that csq_d and csq_i have exactly the same set of operations. However, as a result of merging T_i in step 4, operations in csq_d already include the effects of T_i and hence the two sequences, T_d and csq_d, are relative to the same definition state, that is, $T_d \sqcup csq_d$. Now calling function ITSQ we first incorporate effects of csq_d into T_d, yielding T_d'', and then incorporate effects of H_d'' into T_d'', yielding T_d'. After that, we call mergeSQ to merge T_d' into H_d'', yielding H_d'.

When Algorithm 1 returns, we have fulfilled the first goal of transforming T with H such that $T' = T_i' \cdot T_d'$ can be executed in current state, and the second goal of merging T such that H is updated to $H' = H_i' \cdot H_d'$. In the rest of this section, we will explain the three functions, namely, getConcurrentSQ, ITSQ, and mergeSQ.

4.3. Extract Concurrent Subsequence

Algorithm 2 extracts all operations that are concurrent with sequence T from sequence isq. Input T is a transactional sequence and input isq is an insertion sequence that is ordered by the effects relation. The output csq is a sequence in which all operations are concurrent with T and ordered by the effects relation.

Any operation in the input sequence isq either happens before T or is concurrent with T. As an example, let isq be $[h_1, c_1, h_2, c_2, h_3]$, where h_1, h_2, h_3 happen before T and c_1, c_2 are concurrent with T. Conceptually, we have to transpose these operations such that the result isq' = [h_1', h_2', h_3', c_1', c_2']. That is, $[h_1', h_2', h_3'] \mapsto [c_1', c_2']$, meaning that all those that happen before T are contextually serialized before all those that are concurrent with T. To achieve so, we need to "swap" every h_i with every c_j on its left. As has been well understood [21, 22], a transposition function could be devised as follows: let csq hold the partial results; scan isq from left to right, for every $isq[j]$, if $isq[j] \parallel T$, append it to csq; or if $isq[j] \rightarrow T$, swap it with every operation in csq right to left. The time complexity is $O(|isq|^2)$.

Algorithm 3 ITSQii(sq_1, sq_2) : sq_1'

```
1: i ← 0;   j ← 0;   sq1' ← sq1
2: while (i < |sq2|) and (j < |sq1|) do
3:    ipos ← sq2[i].pos − i
4:    jpos ← sq1[j].pos − j
5:    if ipos < jpos then
6:       i ← i + 1
7:    else if (ipos=jpos) and (sq2[i].id < sq1[j].id) then
8:       i ← i + 1
9:    else
10:      sq1'[j].pos ← sq1'[j].pos + i
11:      j ← j + 1
12:   end if
13: end while
14: for (; j < |sq1|; j++) do
15:   sq1'[j].pos ← sq1'[j].pos + i
16: end for
17: return sq1'
```

However, this process can be greatly simplified if we consider that all operations in the input isq are insertions and ordered by the effects relation. By this property, for any o_i = $isq[i]$, we have the effects relation $o_i \prec o_{i+1}$, where $0 \leq i < i+1 < |isq|$. Swapping o_i and o_{i+1} means that, in an alternative execution order, o_{i+1} is executed before o_i. Swapping execution order of two operations should not change the effects relation. For example, consider state "abc". After executing sequence [ins(1,x), ins(3,y)], the state becomes "axbyc". Transposing these two operations results in sequence [ins(2,y), ins(1,x)], which applied on "abc" yields the same state "axbyc". More generally, let $[o_{i+1}', o_i']$ be the result of swapping $[o_i, o_{i+1}]$. The relation between their positions must be o_i'.pos = o_i.pos and o_{i+1}'.pos = o_{i+1}.pos − 1. That is, the swapping changes o_{i+1}.pos but not o_i.pos.

This property leads to the following simplification of algorithm: each time an operation $isq[j] \rightarrow T$ is swapped with the partial result csq, the swapping does not affect the positions of operations in csq. Also note that we do not really need those operations in isq that happen before T. Therefore, we just need to directly pick up those in isq that are concurrent with T. This is exactly how Algorithm 2 is designed. Obviously its time complexity is $O(|isq|)$.

4.4. Transform Two Sequences

Function ITSQ(sq_1, sq_2) inclusively transforms sequence sq_1 with sq_2, i.e., to incorporate the effects of sq_2 into sq_1. The precondition is $sq_1 \sqcup sq_2$, meaning that the two input sequences are defined in the same state. The postcondition is $sq_2 \mapsto sq_1'$, meaning that the output sequence sq_1' is as if it were to be executed after sq_2.

By operation types of the two input sequences, we can define four ITSQ functions. Here we only define function ITSQii for transforming two insertion sequences, as in Algorithm 3. The other three functions, ITSQid, ITSQdi, and ITSQdd, can be defined similarly. In principle, we need to transform every

Algorithm 4 mergeISQ(sq_1, sq_2): sq

```
1: sq ← [ ];   i ← 0;   j ← 0
2: while (i < |sq2|) and (j < |sq1|) do
3:    if (sq2[i].pos − i ≤ sq1[j].pos) then
4:       sq ← sq · sq2[i]
5:       i ← i + 1
6:    else
7:       sq1[j].pos ← sq1[j].pos + i
8:       sq ← sq · sq1[j]
9:       j ← j + 1
10:   end if
11: end while
12: for (; i < |sq2|; i++) do
13:    sq ← sq · sq2[i]
14: end for
15: for (; j < |sq1|; j++) do
16:    sq1[j].pos ← sq1[j].pos + i
17:    sq ← sq · sq1[j]
18: end for
19: return sq
```

$sq_1[j]$ with every $sq_2[i]$. However, some of the transformations can be saved due to their special ordering.

For any two given operations $sq_1[j]$ and $sq_2[i]$, we can directly transform $sq_1[j]$ with $sq_2[i]$ only when they are defined in the same state and their positions can be compared. Consider that these two sequences are defined in the same state and that operations in these two sequence are contextually serialized. To bring $sq_1[j]$ and $sq_2[i]$ to the same state, we only need to swap $sq_1[j]$ to the front of sq_1 and also swap $sq_2[i]$ to the front of sq_2. By how the operations are ordered in sq_1, we know that all operations ordered on the left of $sq_1[j]$ have effect objects that precede the effect object of $sq_1[j]$. So the swapping results in operation $sq_1[j]'$ such that $sq_1[j]'$.pos = $sq_1[j]$.pos - j to account for the j objects on its left. For the same reason swapping $sq_2[i]$ results in operation $sq_2[i]'$ such that $sq_2[i]'$.pos = $sq_2[i]$.pos $- i$.

Let $ipos$ = $sq_2[i]'$.pos and $jpos$ = $sq_1[j]'$.pos. With every $sq_1[j]$, as in lines 5-8, we fix cursor j on sq_1 and move cursor i on sq_2 until we have identified all operations in sq_2 whose effect objects precede that of $sq_1[j]$. Then, as in line 10, we shift $sq_1[j]$.pos to account for those i operations. There are two cases in which $sq_2[i] \prec sq_1[j]$: (1) $ipos < jpos$, since now they are defined in the same state and can be compared; and (2) $ipos = jpos$ and $sq_2[i]$.id $< sq_1[j]$.id, meaning there is a tie and we break the tie by comparing their site ids. It is worth noting that the tie cannot always be broken this way. The correctness conditions are discussed in [14, 11].

4.5. Merge Two Sequences

Function mergeSQ(sq_1, sq_2) merges sequences sq_1 and sq_2 into a new sequence sq. The two input sequences are ordered by the effects relation and $sq_1 \mapsto sq_2$. The output sequence sq is *effect-equivalent* with $sq_1 \cdot sq_2$. That is, if sq_1 is defined in state s and the state becomes s' after execution of $sq_1 \cdot sq_2$, then the execution of sq in s also yields s'. Here we only specify function mergeISQ as in Algorithm 4, which merges two insertion sequences. The mergeDSQ function for merging two deletion sequences can be specified similarly.

The structure of Algorithm 4 is similar to that of the classic 2-way merge algorithm, since the two input sequences are ordered. However, some transformation is required so as to make comparisons between any two given operations $sq_1[j]$ and $sq_2[i]$. This is achieved as follows: First, we swap $sq_1[j]$ to the end of sq_1. In this process, $sq_1[j]$.pos remains as-is. Second, we swap $sq_2[i]$ to the front of sq_2. In this process, $sq_2[i]$.pos is shifted to account for the i operations on its left, resulting in $sq_2[i]'$.pos = $sq_2[i]$.pos - i. The reasons are similar to those for getConcurrentSQ and ITSQ.

After the above transformation, $sq_1[j] \mapsto sq_2[i]'$, the two insertions are defined in two adjacent states, respectively, and we can compare their positions. Because they are insertions that are executed in tandem, if $sq_2[i]'$.pos $\leq sq_1[j]$.pos, the effect object of $sq_2[i]'$ should precede that of $sq_1[j]$. Hence, in the resulting sequence, $sq_2[i]$ should precede $sq_1[j]$. On the other hand, if $sq_2[i]'$.pos $> sq_1[j]$.pos, the effect object of $sq_2[i]'$ should go after that of $sq_1[j]$ and hence $sq_2[i]$ should be ordered after $sq_1[j]$ in the resulting sequence. However, to account for the i operations in sq_2 that have been placed on its left, we need to adjust $sq_1[j]$.pos by i, as in line 7.

5. OBTAINING TRANSACTIONAL SEQUENCES

Remote transactions are not integrated until the current local transaction is submitted. In this section, we present the endTransaction algorithm for submitting a local transaction and the swapSQ algorithm for swapping two sequences. Some work could be done at the UI level (e.g., as in [12]) to help the user make informed decisions as to when to submit the local transaction and integrate remote transactions.

We record local operations in buffer T that is maintained as $T_i \cdot T_d$. Every time a new local operation o is performed, property $(T_i \cdot T_d) \mapsto o$ must hold; If o.type = del, we swap o with operations in T_d right to left until it can be added in the right position in T_d; if o.type = ins, we first swap o with T_d and then with T_i right to left until it can be added in the right position of T_i. When a transaction is ended, function endTransaction(T, H) is invoked to integrate T into the local history H and meanwhile transform T with H. After that, the resulting sequence T' is propagated to remote sites.

5.1. Submit a Local Transaction

Algorithm 5 endTransaction(T, H): (T', H')

```
1: (T_i', H_d'') ← swapSQdi(H_d, T_i)
2: (T_d', H_d''') ← swapSQdd(H_d'', T_d)
3: H_i' ← mergeISQ(H_i, T_i')
4: H_d' ← mergeDSQ(H_d'', T_d)
5: return (T_i' · T_d',   H_i' · H_d')
```

Function *endTransaction* is specified in Algorithm 5. The two input sequences are $T = T_i \cdot T_d$ and $H = H_i \cdot H_d$, each

Algorithm 6 swapSQdi$(sq_2, sq_1) : (sq_1', sq_2')$

```
1: i ← 0;   j ← 0;   sq1' ← sq1;   sq2' ← sq2
2: while (i < |sq2|) and (j < |sq1|) do
3:    if (sq2[i].pos < sq1[j].pos − j) then
4:       sq2'[i].pos ← sq2'[i].pos + j
5:       i ← i + 1
6:    else
7:       sq1'[j].pos ← sq1'[j].pos + i
8:       j ← j + 1
9:    end if
10: end while
11: for (; j < |sq1|; j++) do
12:    sq1'[j].pos ← sq1'[j].pos + i
13: end for
14: for (; i < |sq2|; i++) do
15:    sq2'[i].pos ← sq2'[i].pos + j
16: end for
17: return (sq1', sq2')
```

of the four subsequences being ordered by the effects relation. Since no new remote transaction has been integrated before this function is called, the history H does not include operations concurrent with T. The precondition is $H \mapsto T$. When the function returns, T is merged into H', $T' = T_i' \cdot T_d'$ and $H' = H_i' \cdot H_d'$. The postconditions are that H' is effect-equivalent with $(H \cdot T)$ and that the effects of H_d are excluded from T'. Note that the latter satisfies the precondition of Algorithm 1. The resulting T' is ready for propagation.

In Algorithm 5, we first swap H_d and T_i to obtain T_i' and H_d''. Since $H_i \mapsto H_d \mapsto T_i \mapsto T_d$, we have $H_i \mapsto T_i' \mapsto H_d'' \mapsto T_d$. Then we swap H_d'' with T_d to obtain T_d' and H_d''', by which we have $H_i \mapsto T_i' \mapsto T_d' \mapsto H_d'''$. Now we have achieved the first goal to obtain $T' = T_i' \cdot T_d'$ in which the effects of all deletions that happen before T are excluded. Next, we merge T_i' into H_i, yielding H_i', and merge T_d into H_d'', yielding H_d'. As discussed in Section , the preconditions of these two merging steps are $H_i \mapsto T_i'$ and $H_d'' \mapsto T_d$, respectively, which are satisfied. As a result, we fulfill the second goal to merge T into H.

5.2. Swap Two Sequences

Now we specify the two swap functions called in Algorithm 5. Here we only give function *swapSQdi* for swapping a deletion sequence and an insertion sequence. Function *swapSQdd* for swapping two deletion sequences can be defined similarly. As in Algorithm 6, function swapSQdi(sq_2, sq_1) swaps the execution order of the two input sequences. The precondition is $sq_2 \mapsto sq_1$, where operations in each sequence are of the same type and ordered by the effects relation. The postcondition is $sq_1' \mapsto sq_2'$.

Conceptually, we would have to swap every $sq_1[j]$ with the entire sq_2 right to left. However, this can be simplified because the two sequences are ordered by the effects relation. For any given $sq_1[j]$ and $sq_2[i]$, we do the following two steps: First, swap $sq_2[i]$ to the end of sq_2. Because its effect object precedes those of the operations on its right, its position remains as-is, i.e., $sq_2[i]'$.pos = $sq_2[i]$.pos. Second, swap $sq_1[j]$ to the front of sq_1. Because its effect object follows those of the operations on its left, its position must be shifted by j to account for those j operations, i.e., $sq_1[j]'$.pos = $sq_1[j]$.pos - j. As a result, $sq_2[i]' \mapsto sq_1[j]'$ and their positions can be compared.

Now we swap these two operations. Let o_i = $sq_2[i]'$ and o_j = $sq_1[j]'$. The conditions are o_i.type = del, o_j.type = ins, and $o_i \mapsto o_j$. We need to consider three cases:

1. If o_i.pos $< o_j$.pos, it means that after o_i.c is deleted, o_j.c is inserted on the right side of position o_i.pos. That is, $o_i \prec o_j$. Hence, if o_j is executed earlier, the position of o_j should be shifted because o_i.c is not deleted yet, whereas o_i.pos should remain as-is because the earlier execution of o_j on the right does not change the position of o_i.c.

2. If o_i.pos $> o_j$.pos, it means $o_j \prec o_i$. Hence, if o_j is executed earlier, o_j.pos should remain because the deferred execution of o_i has no impact, whereas o_i.pos should be shifted because o_j.c has been inserted on the left of o_i.c.

3. If o_i.pos = o_j.pos, it means that, after o_i.c is deleted, o_j.c is inserted at the same position. Since $o_i \rightarrow o_j$, according to proofs in [14], we can safely mandate a tie-breaking policy such that the inserted object precedes the deleted object, that is, $o_j \prec o_i$. As a result, if we swap the execution order and o_j is executed earlier, o_j.pos should remain as-is whereas o_i.pos is shifted.

Therefore, in Algorithm 6, the condition in line 3 corresponds to case (1), $o_i \prec o_j$, by which o_j.pos should be shifted; the condition in line 6 corresponds to cases (2) and (3), $o_j \prec o_i$, by which o_i.pos should be shifted.

We explain the loop in lines 2-10 as follows: On one hand, in lines 6-8, for a given o_i in sq_2, scan sq_1 left to right until all operations that precede o_i in the effects relation are found; for each o_j: $o_j \prec o_i$, we do not need to shift o_j.pos with regard to o_i, however, we need to account for the i operations in sq_2 that have already been in place in sq_2' and whose effects precede o_j; hence we still need to shift o_j.pos by i positions (as in line 7). On the other hand, similarly in lines 3-5, for a given o_j in sq_1, scan sq_2 left to right until all operations that precede o_j in the effects relation are found and, for each o_i such that $o_i \prec o_j$, shift o_i.pos by j to account for those j operations that have already been in place (line 4).

6. AN EXAMPLE

Consider a scenario in which two users, Alice and Bob, are coauthoring a document. Suppose that the initial state is a string "*abcd*". As shown in Figure 2, Alice submitted one transaction T_1 and Bob submitted two transactions T_2 and T_3. We study how the sequences are processed.

6.1. Processing Local Transactional Sequences

Site A: Alice changes the document from "*abcd*" to "*pabq*" by performing a sequence of operations: *[ins(0, p), del(4),*

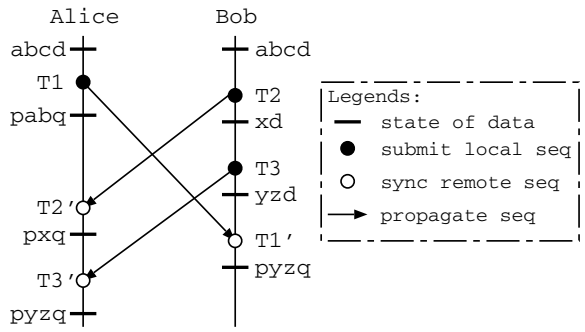

Figure 2. Alice submitted sequence T_1 and Bob submitted two sequences T_2 and T_3.

del(3), ins(3, q)]. Meanwhile, they are reordered into a transactional sequence $T_1 = [ins(0, p), ins(3, q), del(4), del(4)]$. When T_1 is submitted, the local history is empty. Hence, H^A = T_1 and T_1 is propagated as-is.

Site B: Bob first changes "*abcd*" into "*xd*" by performing *[del(1), del(1), ins(1, x), del(0)]*. It is reordered into a transactional sequence $T_2 = [ins(1, x), del(0), del(1), del(1)]$. When T_2 is submitted, the local history is empty. Hence, we get H^B = T_2 and propagate T_2 as-is.

Later, Bob changes "*xd*" into "*yzd*" by performing operations *[ins(1, z), del(0), ins(0, y)]*. They are reordered into a transactional sequence $T_3 = [ins(0, y), ins(2, z), del(1)]$.

Then Algorithm 5 is called to integrate T_3 into local history H^B and produce a transformed transactional sequence that is to be propagated to site A. The effects of deletions in the local history are excluded from T_3 and T_3 is merged into the history. The resulting T_3 and $H^B = H_i^B \cdot H_d^B$ are as follows:
T_3 = *[ins(0, y), ins(3, z), del(2)]*
H_i^B = *[ins(0, y), ins(2, x), ins(3, z)]*
H_d^B = *[del(1), del(1), del(2), del(2)]*

6.2. Processing Remote Transactional Sequences

Site A: Algorithm 1 is called to integrate T_2, which transforms it into an executable form T_2' and merges it with H^A yielding $H^{A'} = H_i^{A'} \cdot H_d^{A'}$, as follows.
T_2' = *[ins(2, x), del(1), del(2)]*
$H_i^{A'}$ = *[ins(0, p), ins(2, x), ins(4, q)]*
$H_d^{A'}$ = *[del(1), del(2), del(3), del(3)]*

After executing T_2' on Alice's document, it becomes "*pxq*". Later, when T_3 is integrated, the results T_3' and $H^{A''} = H_i^{A''} \cdot H_d^{A''}$ are as follows:
T_3' = *[ins(1, y), ins(3, z), del(2)]*
$H_i^{A''}$ = *[ins(0, p), ins(1, y), ins(3, x), ins(4, z), ins(6, q)]*
$H_d^{A''}$ = *[del(2), del(2), del(3), del(4), del(4)]*
Executing T_3' on Alice's document yields "*pyzq*".

Site B: When T_1 is integrated, the results T_1' and history $H^{B'} = H_i^{B'} \cdot H_d^{B'}$ are as follows:
T_1' = *[ins(0, p), ins(3, q), del(4)]*
$H_i^{B'}$ = *[ins(0, p), ins(1, y), ins(3, x), ins(4, z), ins(6, q)]*
$H_d^{B'}$ = *[del(2), del(2), del(3), del(4), del(4)]*
Executing T_1' changes Bob's document to "*pyzq*", which is identical with the final state of Alice.

7. COMPLEXITIES AND PERFORMANCE

The space complexity of the presented ABST algorithm is trivially $O(|H| + |T|)$. From the algorithm specs and explanations, it is easy to see that all the functions, getConcurrentSQ, ITSQ, mergeSQ, and swapSQ each only need to scan their input sequences exactly once. Hence they all have time complexity linear in the size of input. Consequently, the time complexity of integrating either a local transaction or a remote transaction is $O(|H|+|T|)$. Note that there is no extra step for sorting the sequences because they are always maintained in the effects relation order.

As surveyed earlier, most existing OT algorithms are designed to integrate remote operations one by one. In general, they need to transpose the entire history H, which implies $O(|H|^2)$ to integrate one remote operation and $O(|H|^3)$ to integrate a long remote sequence T. Although the time complexity of ABT is in the same order of magnitude, it is faster by some factor because it only needs to transpose H_i which is a subsequence of H. Therefore, using ABT as the baseline in our experiments does not lose generality with regard to other algorithms, e.g., [9, 15, 16, 21, 22].

To obtain a more concrete sense of what the performance implications are in real applications, we implemented ABT and ABST and conducted experiments on a Nokia N810 Internet tablet. The tablet has a 400 MHz ARM v61 CPU (TI OMAP 2420) and 128 MB DDR RAM running Maemo Linux with kernel 2.6.21-omap1. The algorithms are programmed in C# and compiled using Maemo Mono C#. Additionally, the tablet has a built-in slideout keyboard and can be connected to a bluetooth keyboard and an external display, which makes a reasonable mobile computer.

The experiments are designed to study how long it takes to execute Integrate(T, H). In the experiments, we generate two sequences, H and T, such that $H \parallel T$, and call Integrate(T, H) in ABT and ABST, respectively. For every generated T and H, Integrate(T, H) is executed five times and the average time is recorded. For the purposes of this paper, it would suffice to use simulated workloads.

The positions of operations in each sequence are uniformly distributed over the same initial document. In ABT and ABST, operation positions have little impact on the integration time since they do not incur extra costs in breaking insertion ties as in some of the early algorithms [10]. The sizes of the two sequences are varied from 0 to 3,000 with step 300. The percentage of insertions in each sequence is 80% and accordingly the deletion ratio is 20%. The ratio is reasonable for typical editing tasks in which the content increases faster than it decreases. As the insertion ratio decreases, the execution time becomes shorter by some small factor in ABT but it remains almost the same in ABST.

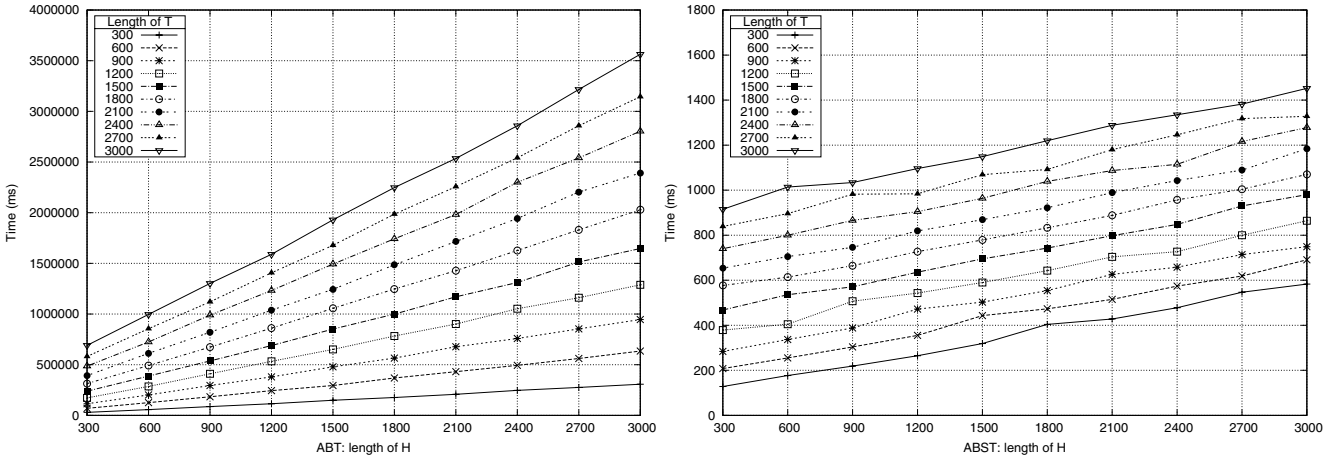

Figure 3. Execution time of integrating two sequences on an Nokia N810 tablet using (left) ABT and (right) ABST.

The experimental results are shown in Figure 3. The data clearly shows that ABST is faster than ABT by several orders of magnitude. For example, when $|T|$=3,000, $|H|$=3,000, and the insertion ratio is 80%, it takes 3,563,556 ms or over 59 minutes in ABT versus 1,452 ms or less than 1.5 seconds in ABST to integrate T with H. The number of 3,000 is not excessive for asynchronous work. For example, if a user performs 20 operations per minute on average, it will take her about 2.5 hours to generate 3,000 operations. Despite specific numbers, our data clearly indicates that ABST is much faster and more suitable for mobile collaboration.

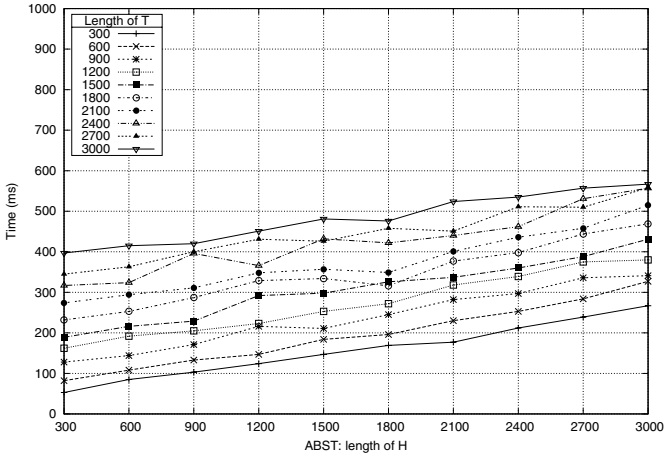

Figure 4. Execution time of endTransaction(T, H) on the N810 tablet.

Further, we measured the time to add a new local operation o to the transaction buffer T and the time to integrate a submitted T with the local history H. Our data shows that the time to add o is negligible: It takes less than 4 milliseconds even when T has as many as 5,000 operations, which means that local responsiveness is not an issue when a user performs local operations. The measured execution times of function endTransaction(T, H) under different values of $|T|$ and $|H|$ are plotted as in Figure 4. Not surprisingly, the time to integrate a local transaction is shorter than the time to integrate a remote transaction by some constant factor.

Integrating local and remote transactions is a rare event, which occurs only when a user is "ready" to sync. Regardless of how sync is initiated, explicitly or implicitly, the user is prepared to expect some delays in a mobile environment. Hence, even though it may take 1-2 seconds when the sequences are long, it will unlikely cause problems [20].

8. CONCLUSIONS

This paper presents ABST, a novel transformation-based consistency control algorithm for integrating sequences of updates to a shared document. Theoretically, it improves time complexity of the state-of-the-art from $O(|H|^3)$ to $O(|H|)$, where H is the operation history. Practically, it can take minutes to hours in previous work versus milliseconds to seconds in this work to integrate a long sequence on mobile devices. As shown in this paper, providing optimized sequence transformation is key to applying OT for mobile collaboration. The improvement is critical because mobile devices are resource-constrained and a user can easily accumulate a large number of operations in a disconnected mode.

For space reasons, we will report the following progress in separate publications. First, ABST is also efficient for real-time collaboration. Intuitively, an algorithm optimized for long sequences works for shorter sequences as well. Secondly, we have built several applications, for distributed version management and group editing, based on ABST with extended support for string operations. Thirdly, correctness proofs [18] and details such as history garbage collection are also left out. Due to the linear time/space complexities in our work, garbage collection is no longer as pressing an issue as in previous work with higher complexities.

Although this work enables a class of mobile applications that are not well-addressed in previous work, it is more of an enabling technology than a complete CSCW solution. Complementary directions include leveraging familiar single-user interfaces (e.g., [12, 24]), helping users make sense of the merged content, and providing selective undo and conflict resolution mechanisms (e.g., [25]). We will extend this work to address related UI and usability issues.

ACKNOWLEDGMENTS

The authors thank the expert referees for their insightful and constructive reviews. The work is supported in part by the National Natural Science Foundation of China (NSFC) under Grant No. 60736020 and No. 60803118, the National Grand Fundamental Research 973 Program of China under Grant No. 2005CB321905, 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. P. Bellini, P. Nesi, and M. B. Spinu. Cooperative visual manipulation of music notation. *ACM Transactions on Computer-Human Interaction*, 9(3):194–237, 2002.
2. Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. Generalizing operational transformation to the standard general markup language. In *ACM CSCW'02*, pages 58–67, November 2002.
3. Clarence A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *ACM SIGMOD'89*, pages 399–407, 1989.
4. Saul Greenberg and David Marwood. Real-time groupware as a distributed system: Concurrency control and its effect on the interface. In *ACM CSCW'94*, pages 207–217, October 1994.
5. Ning Gu, Jiangming Yang, and Qiwei Zhang. Consistency maintenance based on the mark & retrace technique in groupware systems. In *ACM GROUP'05*, pages 264–273, November 2005.
6. Charles M. Hymes and Gary M. Olson. Unblocking brainstorming through the use of simple group editor. In *ACM CSCW'92*, pages 99–106, November 1992.
7. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.
8. Du Li and Manish Anand. MaJaB: Improving resource management for web-based applications on mobile devices. In *ACM MobiSys'09*, pages 95–108, June 2009.
9. Du Li and Rui Li. Preserving operation effects relation in group editors. In *ACM CSCW'04*, pages 457–466, November 2004.
10. Du Li and Rui Li. A performance study of group editing algorithms. In *IEEE ICPADS'06*, pages 300–307, July 2006.
11. Du Li and Rui Li. An admissibility-based operational transformation framework for collaborative editing systems. *Computer Supported Cooperative Work: The Journal of Collaborative Computing (JCSCW)*, August 2009. Accepted.
12. Du Li and Jiajun Lu. A lightweight approach to sharing heterogeneous single-user editors. In *ACM CSCW'06*, pages 139–148, November 2006.
13. Du Li, Chengzheng Sun, Limin Zhou, and Richard R. Muntz. Operation propagation in real-time group editors. *IEEE Multimedia Special Issue on Computer Supported Cooperative Work*, 7(4):55–61, 2000.
14. Rui Li and Du Li. Commutativity-based concurrency control in groupware. In *IEEE CollaborateCom'05*, San Jose, CA, December 2005.
15. Rui Li and Du Li. A new operational transformation framework for real-time group editors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 18(3):307–319, March 2007.
16. Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *ACM CSCW'06*, pages 259–268, November 2006.
17. Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Survey*, 37(1):42–81, March 2005.
18. Bin Shao, Du Li, and Ning Gu. A fast operational transformation algorithm for mobile and asynchronous collaboration. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, September 2009. Accepted.
19. Haifeng Shen and Chengzheng Sun. Flexible merging for asynchronous collaborative systems. In *CoopIS'02*, pages 304–321, October 2002.
20. Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, third edition, 1998.
21. Maher Suleiman, Michèle Cart, and Jean Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *IEEE ICDE'98*, pages 36–45, February 1998.
22. Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *ACM CSCW'98*, pages 59–68, December 1998.
23. Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, March 1998.
24. Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Transactions on Computer-Human Interaction*, 13(4):531–582, December 2006.
25. David Sun and Chengzheng Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, 2009.