A Transformation based New Algorithm for Transforming Deletions in String Wise Operations for Wide-Area Collaborative Applications

Santosh Kumawat
M.Tech Scholar
Poornima College of Engg.
Jaipur, Rajasthan, India

Ajay Khunteta
Asst Prof. Dept. of CS
Poornima College of Engg.
Jaipur, Rajasthan, India

ABSTRACT

Operational transformation (OT) is an established optimistic consistency control method in collaborative applications. This approach requires correct transformation functions. In general all OT algorithms only consider two character-based primitive operations and hardly two or three of them support string based two primitive operations, insert and delete. In this paper we have proposed a new algorithm MSITDD that consider transformation of two deletions and give right result in all possible cases satisfying user intentions and has removed the faults of previous ITDD[1]. In this paper a comparative study is done of the new proposed algorithm MSITDD with ITDD[1] taking an example and is proved that new proposed algorithm MSITDD is giving right output and ITDD[1] is giving wrong output. It also handles overlapping and splitting of operations when concurrent operations are transformed. These algorithms can be applied in a wide range of practical collaborative applications.

General Terms

Operational transformation (OT), optimistic consistency control method.

Keywords

Operational transformation, transformation functions, string operations, deletion transformation, collaborative applications.

1. INTRODUCTION

Operational Transformation (OT) [1] is an established optimistic consistency control method in collaborative applications network.

Operational Transformation (OT) was originally invented for consistency maintenance in plain-text group editors [15]. In over 20 years, OT has evolved to support an increasing number of applications, including group undo , group-awareness , operation notification and compression , spreadsheet and table-centric applications , HTML/XML and tree-structured document editing , word processing and slide creation , transparent and heterogenous application-sharing , and mobile replicated computing and database systems .To effectively and efficiently support existing and new applications, it must continue to improve the capability and quality of OT in solving both old and new problems. The soundness of the theoretical foundation for OT is crucial in this process. One theoretical underpinning of all existing OT algorithms is causality/ concurrency causally related operations must be executed in their causal order; concurrent operations must be transformed before their execution. However, the theory of causality is inadequate to capture essential OT conditions for correct transformation. Collaborative systems using OT typically adopt a replicated architecture for the storage of shared documents to ensure good responsiveness in high latency environments, such as the Internet. The shared documents are replicated at the local storage of each collaborating site, so editing operations can be performed at local sites immediately and then propagated to remote sites. Remote editing operations arriving at a local site are typically transformed and then executed. The transformation ensures that application-dependent consistency criteria are achieved across all sites. The lock-free, non blocking property of OT makes the local response time not sensitive to networking latencies. As a result, OT is particularly suitable for implementing collaboration features such as group editing in the Web/Internet context.

To address the challenge of transforming two deletions , this paper proposes a OT algorithm MSITDD . It is based on the ABT framework [13, 14] which formalizes two correctness condition, causality and admissibility preservation. Causality preservation needed whenever an operation o is executed at a site, all operations that happen before o must have been executed at that site. 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. In general the ABT framework algorithms can be formally proved. The new proposed algorithms is transforming two deletions removing the unfeasibility of earlier algorithms like ITDD[1] and handles overlapping and splitting of operations when concurrent operations are transformed . These algorithms can be applied in a wide range of practical collaborative applications . Moreover, the design of these algorithms 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.

1.1 OT Functions- Inclusion and Exclusion Transformation

OT functions used in different OT systems may be named differently, but they can be classified into two categories.

One is **Inclusion Transformation** (or Forward Transformation): $IT(O_a, O_b)$ or $T(op_1,op_2)$, which transforms operation O_a against another operation O_b in such a way that the impact of O_b is effectively included and the other is **Exclusion Transformation** (or Backward Transformation) : ET (O_a, O_b) or $T^{-1}(op_1,op_2)$,

which transforms operation O_a against another operation O_b in such a way that the impact of O_b is effectively excluded.

2. Background and Related Work

Following the established conventions[1], it model the shared data as a linear string s. Let the position of the first character in any nonempty string be zero. Assume that every appearance of any character has a different object id. Note that this assumption only serves analysis purposes and it do not really need object ids for characters in actual implementation. In this paper, a "character" refers to the object that carries the character, whose ASCII code is possibly only one of its attributes. If c is the i^{th} character in any nonempty string s, it say s[c]= i and c =s[i]. For convenience, it also use s to denote the set of characters in s. For simplicity, it only consider two primitive operations: ins(p; c), which inserts a character c at position p, and del(p), which deletes the character at position p. Apparently, the position parameter p of any operation o is defined relative to some state s. If o is generated in s, then s is called its generation state. If o is executed in s, then s is called its execution state. Due to concurrency, an operation's execution state is not necessarily equivalent to its generation state if the operation is executed at a remote site [12].Given operation o, function pos(o) returns its position value, type(o) returns its operation type (ins or del), char(o) returns the effect character to be inserted or deleted, id(o) returns the id of the site that generates o, gst(o) denotes its generation state, and est(o) denotes its execution state. The fact that the execution of o in state s yields s' is denoted by exec(s; o) = s' . It use two long established relations without further definition: For any

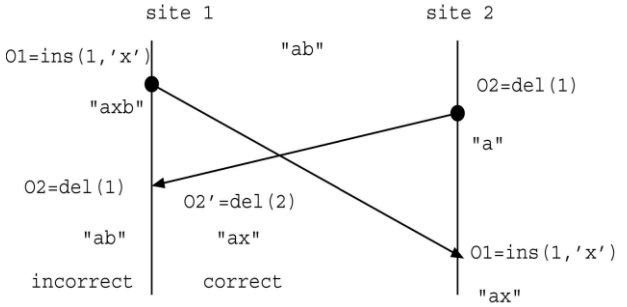

Fig. 1. OT transforms o_2 and then executes o'_2 [1].

two operations o_1 and o_2, o_1 ll o_2 iff o_1 is concurrent with o_2. Iff o_1 happened before o_2 and $o_1 \rightarrow o_2$. The basic idea of OT is to execute any local operation as soon as it is generated for high local responsiveness. Remote operations are transformed against concurrent operations that have been executed locally before its execution. A history buffer HB is maintained at each site to keep track of all executed operations in their order of execution.

As a simple example, consider the scenario in Fig.1. Suppose two sites start from the same initial state $s_1^0 = s_2^0 =$ ''ab:'' Site 1 performs o_1 = ins(1, 'x') to insert character 'x' before 'b', yielding s_1^1 exec(s_1^0 ; o_1) = ''axb; '' while site 2 concurrently performs o_2 = del(1) to delete character 'b', yielding s_2^1 = exec(s_2^0 ; o_2) = ''a:'' When o_2 = del(1) arrives at site 1, if it is executed as it is, then the wrong character 'x' will be deleted. This is because o_2 is generated in s_2^0 without the knowledge of o_1, but its execution state s_1^1 has been changed by the execution of o_1, which invalidates its position parameter. The intuition of OT is to shift the position of o_2 to incorporate the effect of o_1 such that the result o_2' can be correctly executed in state s_1^1 . This process is called inclusion transformation (IT).

Because a character has been inserted by o_1 on the left of its intended position, o_2 should delete the character currently at position 2 instead of 1, i.e., o_2^0 = IT(o_2; o_1) = del(2). The execution of o_2^0 in s_1^1 leads to the correct state s_1^2 = ''ax; '' which is identical to the final state at site 2 after o_2 and o_1 are executed in tandem. As a result, OT seems able to achieve convergence and preserve intentions of operations despite the different orders of execution at different sites.

Another type of transformation function is called exclusion transformation (ET). In the above example, given o_1 defined in s_1^0 and o_2^0 defined in s_1^1 = exec(s_1^0 ; o_1), ET(o_2^0; o_1) excludes the effect of o_1 from o_2^0 as if o_1 had not been executed in s_1^0 . The result o_2 = ET(o_2^0; o_1) = del(1) is exactly the execution form of o_2 as defined relative to s_1^0 .

It has been generally accepted that each OT algorithm consists of two parts: a set of transformation functions (such as IT and ET) that determine how one operation is transformed against another and a control procedure that determines how an operation is transformed against a given operation sequence (e.g., the history buffer). The control procedure is also responsible for generating and propagating local operations as well as executing remote operations.

System Model and Notations

A number of collaborating sites is there in a system. The shared data is replicated at all sites when a session starts. Local operations are executed immediately and for local responsiveness, each site submits operations only to its local replica. In the background, local operations are propagated to remote sites. The shared data is like a linear string of atomic characters. Objects are referred to by their positions in the string, starting from zero . It consider two only 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 attributes like 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. For a operation o, o.pos is always defined relative to some specific state of the shared data.

In the following table1[1] general notations of operation are summarized.

To support string wise transformation, we need to introduce a few more notations. Given any string s, notation |s| is the number of characters in s. If 0 <= i<j <= |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".

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.

2.2 Literature Survey

MOODS[2], a synchronous real-time cooperative editor for music scores. Its architecture includes mechanisms for troubleshooting conflicts in real-time, managing histories of commands and versioning, and performing selective undo. The system also includes specific solutions in order to control the editing of permission profiles.

An integrating approach to concurrency control and group undo that is based on the dOPT algorithm is given by Ellis and Gibbs. It proved the correctness of our adOPTed-algorithm [6] by finding necessary and sufficient preconditions to be satisfied for producing identical application states in replicated groupware architecture

The GRACE editor [3] pioneered the technique of creating multiple versions of objects to accommodate conflicting, concurrent changes. It are looking at an extension of the vanilla GOTO algorithm which uses multiple versioning as an option in cases where transformation is insufficient to preserve operation intentions. It will investigate the utility of this extended GOTO, and other techniques [5] [4] in the context of our grove work.

The difficulty of building correct transformation Functions[8] get demonstrated . Even on a simple string object, all existing transformation functions are incorrect or over- specified. The difficulty stems from the complexity of correctness proof for transformations functions.

A set of transformation functions [9] for structural operations on a grove, that used with the GOTO operational transformation control algorithm [10] and a set of transformations for mutation operations such as [11] will enable synchronous collaborative editing of any meta data rich hierarchical content. In addition, it contribute a new operational transformation control algorithm SLOT for concurrency control, which is significantly simpler and more efficient than existing algorithms. Furthermore, it is free of state vectors, free of ET transformation functions, and free of the TP2 transformation condition.
In addition, it contribute a new operational transformation control algorithm SLOT for concurrency control, which is significantly simpler and more efficient than existing algorithms.

It have contributed the theory of operation context and the COT (Context-based OT) algorithm. The theory of operation context is capable of capturing essential relationships and conditions for all types of operation in an OT system; it provides a new foundation for better understanding and resolving OT problems.

To ensure the convergence of the copies while respecting the user intention, it have proposed two new algorithms, called SOCT3 and SOCT4.

A novel state difference based transformation (SDT) approach which ensures convergence in the presence of arbitrary transformation paths.

It proposes an alternative framework, called admissibility-based transformation (ABT), that is theoretically based on formalized, provable correctness criteria and practically no longer requires transformation functions to work under all conditions. Compared to previous approaches, ABT simplifies the design and proofs of OT algorithms.
Next it is having ABTS for string handling. 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.

3. Algorithms

In this section we are considering the algorithm ITDD[1] and a newly proposed algorithm MSITDD. Taking an example we are analyzing the output of both algorithms in a particular situation and proving that ITDD[1] fails in case when R_2 is included in R_1 where two target regions, R_1= s [b_1 : e_1] for operation o_1 and R_2 = s[b_2 : e_2] for operation o_2.

Algorithm ITDD

Algorithm ITDD[1] is for transforming two deletions. Both o_1 and o_2 are to delete an existing substring in their definition state s. We need to consider the following cases regarding the relations between the two target regions, R_1= s [b_1 : e_1] and R_2= s [b_2 : e_2]
In this algorithm in line-15 when R_2 is included in R_1, it fails and gives unexpected and unfeasible output which cannot be accepted. So this algorithm fails in transforming two deletions when o_2 is substring of o_1. We explain it in more details using the following example.

3.2 Example 1

In this example we are considering the case when R_2 is included in R_1. Let s be common definition state of o_1 and o_2:
s.str= RamBhaktHanumanKiJayHoSansarMae
o_1.str = BhaktHanumanKiJayHo
o_2.str=Hanuman
$b_2 = o_2.pos = 8$
$b_1 = o_1.pos = 3$
$lo_2.strl = 7$
$lo_1.strl = 19$
According to algorithm ITDD
1)From step1 $o_1' \leftarrow o_1$
2) Step 2 and step 3 get executed

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

1: $o_1' \leftarrow o_1$
2: $b_1 \leftarrow o_1.pos;\ \ e_1 \leftarrow o_1.pos + |o_1.str|$
3: $b_2 \leftarrow o_2.pos;\ \ e_2 \leftarrow o_2.pos + |o_2.str|$
4: **if** $b_2 \geq e_1$ **then**
5: return o_1'
6: **else if** $e_2 \leq b_1$ **then**
7: $o_1'.pos \leftarrow b_1 - |o_2.str|$
8: **else if** $b_1 \geq b_2$ and $e_1 \leq e_2$ **then**
9: return ϕ
10: **else if** $b_1 \geq b_2$ and $e_1 > e_2$ **then**
11: $o_1'.pos \leftarrow b_2$
12: $o_1'.str \leftarrow o_1.str[e_2 - b_1 :]$
13: **else if** $b_1 < b_2$ and $e_1 \leq e_2$ **then**
14: $o_1'.str \leftarrow o_1.str[0 : b_2 - b_1]$
15: **else if** $b_1 < b_2$ and $e_2 < e_1$ **then**
16: $o_L \leftarrow o_R \leftarrow o_1$
17: $o_L.str \leftarrow o_1.str[0 : b_2 - b_1]$
18: $o_R.pos \leftarrow b_2$
19: $o_L.str \leftarrow o_1.str[e_2 - b_1 :]$
20: $o_1'.sol \leftarrow [o_L, o_R]$
21: **end if**
22: return o_1'

3) Conditions at step 4, 6, 8, 10, 13 is false for given o_1 and o_2 so switch to line-15 step by step .
4) Condition at line-15 is true so enter in if-block
5) From step16 we have $o_L = o_1$ and $o_R = o_1$
6) From step17 o_L.str← o_1.str[0:5]=Bhakt
7) From step16 o_L. pos = o_1.pos = 3
8) From step18 o_R. pos = o_2.pos = 8
9)From step19 o_L.str←o_1.str[14-3:] = o_1.str[11:] = KiJayHo
10)From step19 and step16 o_L.str = KiJayHo and o_L. pos =3 . Also o_R. pos =8 from step18 and o_R= o_1 from step16.
11) o_1'.sol =[o_L , o_R] from step 20

Note here o_1'.sol =[o_L , o_R] is not possible because o_L. pos =3 and o_L.str = KiJayHo , so lo_L.strl=7 and o_L. pos+ lo_L.strl=10 , so o_L.str is ending at position 9 and o_R. pos = o_2.pos = 8 from step 8 of example. So there is overlapping between o_L and o_R at position 8 and 9 ,so o_1'.sol =[o_L , o_R] not possible because according to traditional notations from table1 [o_L , o_R] means concatenation of two strings operations in a sequence or order, so overlapping of o_L and o_R is infeasible and unaccepted.
Also in string s when we substitute o_1'.sol at place of o_1 then lo_1'.strl= lo_L.strl + lo_R.strl =7+19=26 and lo_1.strl=19 so again there is overlapping between o_1'.sol and given string s, so again the resulting string is totally unaccepted.

3.3 Algorithm MSITDD(o_1, o_2): o_1'

1. o_1'← o_1
2. if o_1.pos< o_2.pos and (lo_2.strl + o_2.pos) < (lo_1.strl+ o_1.pos) then
3. o_A ← o_1 and o_B← o_1
4. o_A.str← o_1.str[0: (o_2.pos- o_1.pos)]
5. o_B.pos← o_2.pos
6. o_B.str← o_1.str[{(o_2.pos+ lo_2.strl)- o_1.pos)}:]
7. o_1'.sol ←[o_A , o_B]
8. elseif (o_2.pos+ lo_2.strl)<= o_1.pos then
9. o_1'.pos← o_1.pos- lo_2.strl
10. elseif o_1.pos>= o_2.pos
11. if (lo_1.strl+ o_1.pos)<= (o_2.pos+ lo_2.strl) then
12. o_1'← ¢
13. endif
14. elseif o_1.pos< o_2.pos
15. if (lo_1.strl+ o_1.pos) <=(o_2.pos+ lo_2.strl) then
16. o_1'.str← o_1.str[0: (o_2.pos- o_1.pos)]
17. endif
18. elseif o_1.pos>= o_2.pos and (lo_1.strl+ o_1.pos) >(o_2.pos+ lo_2.strl) then
19. o_1'.pos← o_2.pos
    o_1'.str← o_1.str[{(o_2.pos+ lo_2.strl)-o_1.pos) }:]
20. endif
21. return o_1'

We need to consider the following cases regarding the relations between the two target regions, R_1 = s [o_1.pos: (lo_1.strl+ o_1.pos)] and R_2 = s [o_2.pos: (o_2.pos+ lo_2.strl)]
1. When 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-8) R_1 is on the right of R_2. After R_2 is deleted, we shift o_1'.pos by 1 o_2.strl characters to the left.
3. (line10-13) 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 ¢.
4. (line-18) 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 (o_2.pos). And o_1.*str* only needs to include the right part that is not deleted by o_2, starting from (o_2.pos+ lo_2.strl) - o_1.pos in the original o_1.*st*.
5. (line-14) R_2 partially overlaps with R_1 around the right border of R_1. This case is 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-2) R_2 is included in R_1. 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.

3.4 Example 2

In this example we are considering the case when R_2 is included in R_1. In this situation algorithm ITDD[1] fails what we have proved by example 1. Now we are considering the same situation using our newly proposed algorithm MSITDD and proving that it is giving right output in this condition also. Due to space reasons we are not considering other cases but our new proposed algorithm work well not only in this condition but also in all other situations whatever is possible in case of transforming two deletions.

In this example all parameters like s.str, o_1.str, o_2.str are taken like example1. so that comparison in ITDD and MSITDD becomes more clear.
Let s be common definition state of o_1 and o_2
s.str= RamBhaktHanumanKiJayHoSansarMae
o_1.str = BhaktHanumanKiJayHo
o_2.str=Hanuman
b_2 = o_2.pos = 8
b_1 = o_1.pos = 3
lo_2.strl = 7
lo_1.strl = 19

According to algorithm MSITDD
1) From step1 o_1'← o_1

2) Condition at line-2 is true so enter in if-block
3) From step3 we have $o_A = o_1$ and $o_B = o_1$
4) From step4 o_A.str← o_1.str[0:5]=Bhakt
5) From step3 o_A . pos = o_1.pos = 3
6) From step 5 o_B . pos = o_2.pos = 8
7) From step6 o_B.str←o_1.str[14-3:] = o_1.str[11:] = KiJayHo
8) o_1'.sol =[o_A , o_B] from step 7
From step 4 and step5 of example we have o_A.str = Bhakt and o_A . pos =3 . Also o_B . pos =8 from step 6 and o_B.str = KiJayHo from step7 of example. So o_1'.sol =[o_A , o_B] is totally feasible here and o_1'.str= BhaktKiJayHo and there is no overlapping between o_A and o_B because o_A . pos+l o_A.str l is less than or equal to o_B . pos. Also no overlapping with given string s of o_1' because o_1'. pos = o_1. pos and l o_1'.str l is less than lo_1.strl . So the output of MSITDD is totally feasible in the case when R_2 is included in R_1 but in this case ITDD[1] totally fails.
Also MSITDD satisfy all other possible cases also in case of transforming two deletions. Also MSITDD cover all cases what get covered by ITDD.

4. CONCLUSION

In this paper we have also proposed a new algorithm called MSITDD for transformation of two deletions in all possible cases. Also taking an example have explained that MSITDD work well in all possible cases but ITDD[1] fails in some particular cases. So MSITDD has removed the faults what was earlier in ITDD[1].

To address the challenge of transforming two deletions, this paper proposes a OT algorithm MSITDD. It is based on the ABT framework [13, 14] which formalizes two correctness condition, causality and admissibility preservation. These algorithms can be applied in a wide range of practical collaborative applications that require string operations. In general the ABT framework algorithms can be formally proved. The new proposed algorithms is transforming two deletions removing the unfeasibility of earlier algorithms like ITDD[1] and handles overlapping and splitting of operations when concurrent operations are transformed. Moreover, the design of these algorithms 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.

4.1 Future Work

There is a lot of efforts needed to preserve intention preservation and also to preserve semantic consistency and syntactic consistency. There is still scope to extend the support to other composite operations of string handling and char handling. Also it can support other better data structures also. A lot of work is done to reduce space complexity and time complexity. Still there is a scope to reduce space complexity and time complexity.

5. REFERENCES

[1] ABTS: A Transformation-Based Consistency Control Algorithm for Wide-Area Collaborative Applications Bin Shao , Du Li , Ning Gu . IEEE Paper published in 2009

[2] P. Bellini, P. Nesi, and M.B. Spinu, "Cooperative Visual Manipulation of Music Notation," ACM Trans. Computer-Human Interaction, vol. 9, no. 3, pp. 194-237, Sept. 2002.

[3 Chengzheng Sun and David Chen. Consistency maintenance in real-time collaborative graphics editing systems. ACM Transactions on Computer-Human Interaction, 9(1):1–41, March 2002.

[4] Chengzheng Sun. Optional and responsive fine-grain locking in internet-based collaborative systems. IEEE Transactions on Parallel and Distributed Systems, 28(9):994–1008, September 2002.

[5] Chengzheng Sun. Undo as concurrent inverse in group editors. ACM Transactions on Computer-Human Interaction, 10(1), March 2003. (to appear).

[6] Ressel, D. Nitsche-Ruhland, and R. Gunzenha¨user, "An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors," Proc. ACM Conf. Computer- Supported Cooperative Work (CSCW '96), pp. 288-297, Nov. 1996.

[7] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In ACM CSCW'98, pages 59–68, Dec. 1998.

[8] 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.

[9] A.H. Davis, C. Sun, and J. Lu, "Generalizing Operational Transformation to the Standard General Markup Language," Proc. ACM Conf. Computer-Supported Cooperative Work (CSCW '02), pp. 58-67, Nov. 2002.

[10]. Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In Proceedings of ACM Conference on Computer Supported Cooperative Work, pages 59–68. ACM, May 1998.

[11] Chengzheng Sun, Xiaohua Jia, Yanchung 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.

[12] C. Sun and C. Ellis, "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements," Proc. ACM Conf. Computer-Supported Cooperative Work (CSCW '98), pp. 59-68, Dec. 1998.

[13] 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.

[14] 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.

[15] D. Sun, S. Xia, C. Sun, and D. Chen, "Operational Transformation for Collaborative Word Processing," Proc. ACM Conf. Computer- Supported Cooperative Work (CSCW '04), pp. 162-171, Nov. 2004.