Overview of Operational Transformation

Zach Smith
University of Minnesota, Morris
smit4608@morris.umn.edu

ABSTRACT

The Operational Transformation algorithm is designed to facilitate seamless real-time cooperation and collaboration. Operational transformation can be used for collaboration where users need to remotely interact with a shared resource in real-time. In this paper we specifically cover cases in which operational transformation is used in text-documents that are collaboratively edited in a web-based environment. Operational transformation can, however, be used in many different contexts.

Categories and Subject Descriptors

H.5.3 [**Information Systems**]: Group and Organization Interfaces—*Web-Based Interaction*; C.2.4 [**Computer Systems Organization**]: Distributed Systems—*Client/Server*

General Terms

Algorithms, Theory

Keywords

Computer Supported Collaborative Work, Operational Transformation

1. INTRODUCTION

In recent years collaborative technologies have become more popular. Many current applications allow physically separated users to simultaneously edit a shared document, spreadsheet, presentation, or image. A well-known example of this type of application is Google Docs.

In order to facilitate this type of interaction, the software must seamlessly keep collaborators up-to-date with one another. Optimally the software would allow users to work as easily as if they were not using collaborative software.

Pessimistic concurrency technologies such as locking, where a user locks a document, preventing all other users from using it until they save and unlock the document, are ill equipped to solve the problem of Computer Supported Cooperative Work (CSCW), since only one user can interact with a document at time, and other users cannot see the changes made by other users in real-time. Other methods of collaboration, such as those used by source-code control software (e.g. subversion or git) can support multiple users editing the same document, but do not support real-time updates, and many conflicts must be handled manually by the user, making these solutions non-optimal for situations where real-time collaboration is a requirement.

For this discussion, we are interested in *optimistic* concurrency, which is real-time, distributed, and unconstrained [5]. Real-time, in this context, means that the latency between when a client performs an action and another client sees the action is determined by the users connection latency. If the delay is much longer than it takes for the changes to propagate through the network, the system is not real-time.

The term “distributed” means that clients can be on separate remote machines and still interact with one another. If users are constrained to being on the same machine, the algorithm is not distributed. The term “unconstrained” means the user can make modifications to any part of the document at the same time as other users, regardless of what any other users are doing.

Operational Transformation (OT) is an optimistic consistency control algorithm used in applications such as Google Docs [3]. OT is well-suited to collaborative applications, since it allows multiple people to collaborate on a single document simultaneously. OT allows client software to respond immediately to user interaction, regardless of network latency, and will merge updates from other clients seamlessly and without any interaction required from the user.

This means that, when a user types something, it appears immediately as they type it. Those changes are sent to the server, and any new changes are integrated into the user’s view of the document [1].

UMM CSci Senior Seminar Conference, April 2012 Morris, MN.

2. OPERATIONAL TRANSFORMATION

There are many implementations of operational transformation. The implementations we will be discussing, such as [4] and [6], are based on a client-server model. For web applications a client-server model is the only viable model due to difficulties in implementing peer-to-peer systems from a web browser. The remainder of this section will outline the fundamentals of operational transformation: operations, transform functions, and the constraints required to achieve consistency.

2.1 Operations

The basic unit of OT is an operation. In a basic OT algorithm, an operation is the insertion or deletion of a single character. From these two operations we can make any change to a document. A deletion followed by an insertion, for example, makes a substitution.

An important property of the operation is causality. We will use the following notation to describe causality: given two operations o_1 and o_2 the notation $o_1 \rightarrow o_2$ denotes that o_1 occurred before o_2. Causality will be discussed more in depth later.

2.2 Consistency Model

Previous OT work use a consistency model based on convergence, causality preservation, and intention preservation [5].

1. *Convergence*: Eventually all documents in the system will be in the same state. Documents diverge when a client makes an edit to a document. For an OT algorithm to be correct the algorithm must ensure that all documents in the system eventually converge on the same state.

2. *Causality preservation*: any two operations o_1 and o_2, such that $o_1 \rightarrow o_2$, if both executed, must be executed in that order on all clients.

3. *Intention Preservation*: the effect of any operation o must, on all clients, be the effect that the creator of the operation intended.

Convergence is the property that, even though clients may have differing states, once they have received and applied all of the updates from the server, all clients must have the same final state. If client 1 inserts an **x** and client 2 inserts a **y** after both clients have received and applied the operations they must both either have the state **xy** or **yx**.

Causality is simply the order in which operations occur. Any two operations are causally related if o_1 was executed before o_2 at the same location. If o_1 occurs before o_2 but they are at different locations, i.e. two separate clients, and neither client has received notification of either of the operations, the operations are not causally related.

If convergence and causality preservation were the only rules for maintaining consistency, an algorithm could simply always return a sequence of operations to delete all of the characters. With this algorithm the documents would always converge on the same state, an empty string.

To solve this, [5] suggests intention preservation. For example, assume we have a document that is shared between two clients. The document consists of the following string:

There will be word here

Client 1 inserts an **s**, changing their document to:

There will be words here

At the same time, client 2 inserts the article **a** changing their document to:

There will be a word here

Each client's intention was to correct the grammar of the sentence, however, determining that this was their intention is a very difficult problem. A naive approach to intention preservation, which is what is used in most OT implementations, will simply converge upon the following string:

There will be a words here

The problem that becomes obvious in this situation is that it preserves syntactic intention but not semantic intention. However, the design of an algorithm that also preserves semantic intention would present a difficult problem [5].

2.3 Transformation

The transformation step of OT is responsible for bringing clients into convergence. The transform function *transform(seq1, seq2)* takes in two sequences of operations and returns a sequence seq_2'. When the sequence seq_2' is applied to a client who has already applied seq_1, the client will converge with another client who has already applied seq_2 and then applies seq_1*'*, which is the result of *transform(seq2, seq1)*.

Applying seq_1 followed by seq_2' results in the same end state as applying seq_2 followed by seq_1'. This is very useful in a collaborative environment, since each client will immediately apply any operations it receives from the user. The server needs to be able to compute which operations need to be performed on the client in order to bring it into convergence.

Assume we have a string "ca" which is the current state of the document on all clients and the server. Client 1 performs an operation o_1 = *ins(2, n)* and at the same time client 2 performs operation o_1 = *ins(2, t)*. Each client has applied the operation to their local copy of the document, so client 1's state is "can" and client 2's state is "cat", when the operations reach the server. If we were to simply send client 1's operation to client 2 and client 2's operation to client 1 the two states would not converge. Client 1 would apply client 2's operation, resulting in "catn" and client 2 would apply client 1's operation resulting in "cant". This violates convergence, since all operations have been applied on all clients and the documents are not in the same state.

If both operations arrive at the same time, the server can apply either operation first, as long as it applies sequences atomically. The server first applies o_1, bringing its state to "can" It then receives o_1, and since o_2 is based on the state "ca" it applies *transform(*o_1*,* o_2*)* which gives us o_2' = *ins(2, n)* and *transform(*o_2*,* o_1*)* which gives us o_2' = *ins(3, t)*. Then, sequence o_1' is sent to client 2 and o_2' is sent to client 1. After the operations are received and applied, both clients and the server converge on the final state "cant". Note that if the server had integrated client 2's operation first, the final state would be "catn", which would be completely valid.

This is the simplest case of operational transformation. However, multiple sequences of operations can be sent from a client a, before client b receives these changes. This means that b's operations have not taken into account any of client a's updates, so multiple transform steps will have to be performed to bring the a and b documents back into convergence. In this case, the transform function cannot simply be applied to the operations, because the second client's operations are not based on the same state first client's operations.

Therefore, the last state that both clients shared must be found, and transform must be run multiple times to find the set of operations to send to to bring the clients into convergence. One issue that this creates is that it ends up taking $O(|h|^2)$ memory to maintain a history of operations in the worst case. However, methods exist to garbage collect unneeded history, so that the history does not end up being the number of operations that have been performed. Otherwise, If a client has been editing a document for hours, the number of operations for each client could easily be in the thousands, making OT slow and memory intensive. In the next two sections, we will discuss implementations of OT which decrease this space requirement.

2.4 Google Wave

Google Wave is a communication tool which uses Operational Transformation to allow users to edit shared documents collaboratively. Google Wave supports operations for inserting opening and closing XML tags, annotating ranges of the text, and retaining characters, in addition to inserting and deleting characters [6].

A sequence of operations in Google Wave is a sequence which spans the entire length of the document. Instead of each operation having a position as above, a number of characters are retained in between each ins and delete. The sequence to add an "e" to the end of "wav" would look like "retain(3)ins(e)". It would retain the first three characters, and then it would insert an "e" after the third character.

In Google Wave's OT algorithm two sequences of operations can be composed. Given two sequences of operations, the composition of those two sequences would be a new sequence, which would have the same effect as applying the first sequence followed by the second sequence.

This algorithm allows both the server and the client to store operations and then compose new operations onto the old, before sending them to the server. This can reduce bandwidth, because the server does not have to send each operation or small set of operations individually. Since these sequences are linear and ordered based on the document they are modifying, two sequences of operations can be composed efficiently in linear time.

To resolve the problem of the server having to store $O(|h|^2)$ history in the worst case. Google Wave does not allow clients to rapid-fire changes to the server. The client will compose all new operations into a buffer until the server sends a message requesting the changes the client has made. It will send this message for to each client that is connected. The client will then send the stream of operations to the server. The server can then parse all of the streams from all of the clients at once.

Because the server decides when the client sends all of its operations, it knows exactly what state the client operations are based on, namely the last one it sent. This reduces the complexity of the history from $O(|h|^2)$ to $O(|h|)$.

One downside of this solution is that changes are not propagated as often, which may lead to a reduced user experience [6]. Another is that more work is required of the client, which may drain battery life on mobile devices [3]. The upside of more work being performed on the client is that it frees up memory and resources on the server, since the server only needs to store a copy of the document, rather than the transform operation history of each client, making the server much more reliable and scalable [6].

3. ADMISSIBILITY BASED TRANSFORM

Admissibility Based Transform (ABT) is a recent model of Operational Transformation where intention preservation is not used as a correctness property, instead, the admissibility property is used. Admissibility ensures the order of characters is consistent across all clients at all times [2].

ABT has many similarities to the Google Wave protocol discussed earlier. ABT prevents clients from getting more than one step out of state, relative to the other clients, by limiting how often clients can send new operations, and has clients buffer operations until the server is ready for them.

As an example of admissibility, if a character **x** is inserted before another character **y** then at any point, on all clients and the on server, the character **x** must always precede the character **y**, this must hold true for the clients as well as the server. If **x** were to appear after the character **y** the effects relation would be violated and the algorithm would not satisfy the admissibility criterion.

The admissibility property has been formally proven correct. The theoretical proof of admissibility based OT, a global effects relation graph is created. The effects relation property is the order relative to the position that they effect. This graph observes the entire system, where the nodes are operations, and the edges are the effects relation between the two operations. Admissibility is preserved when the execution of an operation does not violate the effects relation in the global graph, meaning that there are no cycles between nodes. If there were a cycle, that would mean that an operation o_1 preceded an operation o_2 at one point, and at another point o_2 preceded o_1 [2].

As an example of the effects relation order, when given the sequence of operations *[ins(2, x), del(1, b), ins(4, y), del(2, c)]*, the string "abcd" will be changed to "axdy". When reordered into effects relation order, this sequence becomes *[ins(1, b), ins(1, x), del(2, c), ins(3, y)]*. These two sequences, when applied to a document, will have the same effect [4].

3.1 Use in Web Applications

The OT framework presented in [4] uses an Admissibility-Based Transformation (ABT) algorithm. The framework can be used to implement Web 2.0 CSCW over HTTP with a request-response architecture, or if available with a server push architecture.

A request-response architecture restricts the ways in which the server can communicate with the client. The server cannot initiate a connection to the client, and must wait until the client contacts it and asks for information to send data. This is an issue for a simple implementation of operational transformation, since the server needs to send updates out to clients as fast as possible once another collaborator has made changes.

The way around this is to have the client poll at a certain interval. For example, rather than send an update message every time any client changes something, the server must remember the results of all of the transformations it ran against messages from the client, and then send them to the client when the client polls. The issue with polling is that there is a great deal of overhead which comes from creating a new http connection every 100-200 milliseconds. Many times polls are unproductive, since the server has no information for the client and the response from the server is simply that there is no new data for the client.

The request-response architecture can also create problems with detecting disconnect, since there is no persistent full-duplex connection, a connection where information can be sent or received by either party at any time. The server the server has no way of knowing when a client has disconnected.

When using a request-response architecture the algorithm on the client handles all user input and merges it into a local buffer of operations. This is very similar to the method used in the Google Wave Protocol, in that all changes are queued in a buffer and then composed or merged as new operations are created. Since the server cannot send changes to the client at any time, it has a similar buffering procedure before it sends the operations to each client.

At an interval the client checks for any updates from the server. If the client receives any changes it must transform them against its local operation buffer, this will become the new working copy for the user. It must then perform the reverse transformation in order to find the set of operations to send to the server [4].

In a server push architecture, a full-duplex connection to the server is kept open. This allows the server to request information from the client at any point. This vastly simplifies the design of an ABT implementation. We will therefore discuss an implementation of ABT with a server push architecture in the following section.

3.2 Implementation

A protocol for implementing ABT is the Transformation and Time Interval Based Protocol for Synchronization (TIPS) from [4]. TIPS is designed for Web 2.0 applications, specifically, the client-side portion can be implemented in a web browser. In [4] the authors describe both a request-response protocol for TIPS, as well as an outline for implementing TIPS as with a Server Push architecture. We will discuss the server push architecture for the remainder of this section. An outline of the algorithm can be seen in Figure 1.

Firstly, all clients must be assigned a unique id. For this discussion we'll assign each client a monotonically increasing integer id. In the TIPS protocol, each operation o_i consists of the following four parts:

1. The type of the operation (e.g. insert or delete).
2. The specific character to insert or delete (e.g. 'w').
3. The index at which the character should be inserted at or deleted.
4. The id of the client where the character was inserted, or, if the operation type is delete, the ids of all clients where this character was deleted.

Note, if the operation was a deletion, we must store the ids of all clients at which this character has been deleted. This ensures the same character is not deleted multiple times, since it only exists once in any copy of the document.

At initialization, each client starts with a copy of the document, this document is the same as the copy of the document on the server. The document is presented to the user and they are able to modify the document. Any modifications they make are turned into operations.

Each operation a user performs is merged into a sequence of performed operations *Buffer*, which is stored on the client. The *merge* algorithm used merges the operation into *Buffer*, and maintains the effects relation order. This is similar to the composition of operations in Google Wave OT. This algorithm will be explained in greater detail later.

At an interval I_s the server sends a *SYNC* message to each client. When a client receives this they send all of the operations in *Buffer* to the server.

Once the server has received the operations from each connected client, the *nWayMerge* algorithm is used. *nWayMerge* repeatedly merges two sequences into each other until there is only one sequence left, this will be explained in greater detail later. The result of the *nWayMerge* is then applied to the server's master copy of the data and is sent to each client. Before the result of *nWayMerge* can be sent back to the client, we must remove all operations that originated from the client receiving the sequence. Recall that, one property of an operation in ABT is the originating client's id, so we remove all operations whose client id is the same as the id the operations are being sent to.

Once that is complete, the operations are sent to the client, shown as the *UPDATE* message in Figure 1. Once received by the client, the operations are transformed with any new client operations via *transformSequence* and then applied to the local working copy, which allows the user to see changes made by their collaborators. We will now discuss the component algorithms in greater detail.

3.3 merge

The *merge* algorithm allows the client to store sequences of operations without sending them to the server immediately. As operations are performed, they are applied to the working copy and displayed immediately. Then *merge* merges the operations into the local history, *Buffer*. This allows users to make changes without the changes being sent to the server until the server requests them. The running time of this algorithm is $O(n)$ where n is the length of sequence.

The *merge* algorithm takes in a single operation and a sequence of operations and inserts the single operation into the sequence in the correct effects relation order. It works by looping backwards over *Buffer* shifting each operation's position, until the correct position is found and then the operation is inserted into *Buffer*.

In order to discuss *merge*, we define a utility method *precedes*(o_1, o_2). The *precedes* function takes in two operations and returns *true* if the first operation comes before the second in effects relation order, and *false* otherwise. More specifically, it returns *true* if $o_1.position < o_2.position$ or $o_1.position = o_2.position$ and $o_1.type = delete$, and returns *false* otherwise. Below, $Buffer_i$ denotes the ith operation in $Buffer$.

The *merge* algorithm can be defined more precisely as follows:

```
1. Initialize:
   $insertPosition \leftarrow length(Buffer)$
   $\Delta \leftarrow 1$ if $o.type = insert$
   $\Delta \leftarrow -1$ if $o.type = delete$
2. For each position $i$ in Buffer starting from the end:
   (a) If $precedes(Buffer_i, o) = true$, break out of the loop.
   (b) Else, $Buffer_i.position \leftarrow Buffer_i.position + \Delta$
       $insertPosition \leftarrow i$.
3. Insert operation $o$ into Buffer at insertPosition.
```

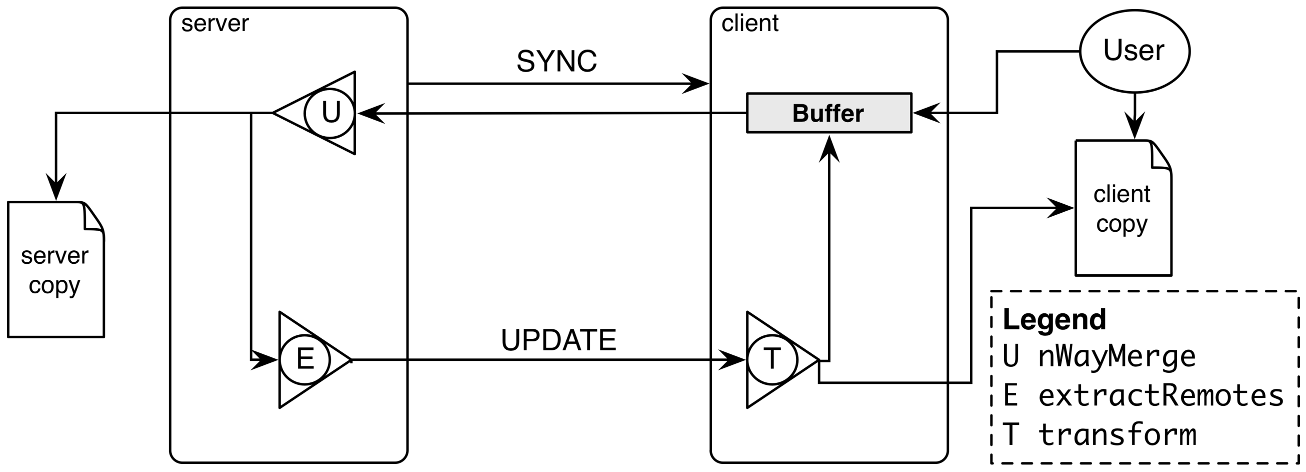

Figure 1: An overview of TIPS with server push

3.4 nWayMerge

Once the server has sent the *SYNC* message to the clients, and has received a sequence of operations from each client, it must merge the operations into a sequence of all of the changes that have happened since the last *SYNC*.

It does this with the *nWayMerge* algorithm, which takes in a sequence of operations from each connected client and merges them into a single sequence of operations in effects relation order.

The *nWayMerge* algorithm takes in *list*, which is a list of sequences of operations, which will all be from different clients. The sequences in list will, for our purposes, always be the sequences received from the *SYNC* command.

A more rigorous explanation follows:

```
1. While $length(list) > 1$:
   (a) $list_{first} \leftarrow mergeSequence(list_{first},\ list_{last})$
   (b) remove $list_{last}$ from $list$.
2. return $list_{first}$
```

3.5 mergeSequence

The *mergeSequence* algorithm is called by *nWayMerge* on all of the sequences received from clients. It takes in two sequences of operations seq_1 and seq_2, which may be from different clients and merges them together, returning the merged result.

The algorithm goes through all operations in each sequence, merges them into a single new sequence and updates the new position of each operation, maintaining effects relation order.

We'll use an expanded version of the *precedes* function *multiClientPrecedes*, with an additional case, if $o_1.position = o_2.position$ and $o_1.type$ and $o_2.type$ are *insert*, return true if $o_1.id < o_2.id$. This last clause is a tie breaker, for two clients committing an insert operation at the same time and at the same position. The tie needs to be broken in a consistent manner on the server as well as on all clients, so the operation with a lower client id number precedes the same operation from another client. Inside the while loop refer to $seq_{1,i}$ as o_1 and $seq_{2,j}$ as o_2. The Δ_1 and Δ_2 variables are used to shift the offsets to include the effects of the merged operations.

```
1. Initialize:
   $seq \leftarrow$ empty sequence
   $i \leftarrow 0$
   $j \leftarrow 0$
   $\Delta_1 \leftarrow 0$
   $\Delta_2 \leftarrow 0$
2. While $i < length(seq_1)$ and $j < length(seq_2)$:
   (a) $o_1.position \leftarrow o_1.position + \Delta_1$
       $o_2.position \leftarrow o_2.position + \Delta_2$
   (b) if $o_1$ and $o_2$ are delete operations, and have the same position and have the same character
       then $o_1.position \leftarrow o_1.position + \Delta_2$
       $o_1.ids \leftarrow o_2.ids \cup o_1.ids$
       Append $o_1$ to $seq$
   (c) Else if $multiClientPrecedes(o_1,\ o_2)$
       then $o_1.position \leftarrow o_1.position + \Delta_2$
       $o_2.position \leftarrow o_2.position + \Delta_1$
       if $o_2.type = insert$ then increment $\Delta_1$,
       else decrement $\Delta_1$
       Append $o_1$ to $seq$
       Increment $j$
   (d) Else $o_2.position \leftarrow o_2.position + \Delta_1$
       if $o_2.type = insert$ then increment $\Delta_2$,
       else decrement $\Delta_2$
       Append $o_2$ to $seq$
       Increment $i$
3. For each remaining operation in $seq_2$ append it to seq
   and add $\Delta_1$ to the operation's position.
4. For each remaining operation in $seq_1$ append it to seq
   and add $\Delta_2$ to the operation's position.
```

It is important to note that within the loop, if both operations are deleting the same character at the same position, we only add the first operation to *seq* because a character can only be deleted once, since once it has been deleted it is gone and cannot be deleted again. We therefore add the ids in o_2 to the ids in o_1 and append o_1 to *seq*.

3.6 transformSequence

The *transformSequence* algorithm is responsible for transforming the sequence received from the server, the input sequence will be the result of *nWayMerge* received from the server. Before operations can be transformed, however, the *extractRemotes* algorithm must remove any operations from *seq* where the $o.id = id$ in the case of an insertion or $id \in o.ids$ in the case of a deletion. This prevents a clients own operations from being reapplied on their working copy of the document [4].

The *transformSequence*(seq_1, seq_2) algorithm takes in two sequences seq_1 and seq_2 and returns a sequence seq'_1 which, when applied to a document after seq_2 has been applied, will converge with a document where seq_1 and result of *transformSequence*(seq_2, seq_1) have been applied.

```
1. Initialize:
   $seq'_1 \leftarrow seq_1$
   $i \leftarrow 0$
   $j \leftarrow 0$
   $\Delta_1 \leftarrow 0$
   $\Delta_2 \leftarrow 0$

2. While $i < length(seq_1)$ and $j < length(seq_2)$:
   (a) $o_1.position \leftarrow o_1.position + \Delta_1$
       $o_2.position \leftarrow o_2.position + \Delta_2$
   (b) If both operations are delete operations and have
       the same position and the same character then
       replace the operation at $seq'_{1,j}$ with an identity
       operation.
       Increment $i$ and $j$
   (c) Else if $multiClientPrecedes(o_1, o_2)$, $seq'_{1,j}.position$
       $\leftarrow seq'_{1,j}.position + \Delta_2$
       Increment $j$
       if $o_2.type = insert$ then increment $\Delta_1$,
       else decrement $\Delta_1$.
   (d) increment $i$
       if $o_2.type = insert$ then increment $\Delta_2$,
       else decrement $\Delta_2$.

3. For each remaining operation in $seq'_1$ add $\Delta_2$ to its
   position.
```

After this step is completed, the cycle can repeat. The user can perform operations and they will be transformed and propagated out to all other collaborators.

4. CONCLUSIONS

This paper discusses Operational Transformation from a high level, covering the problems it tries to solve and the properties of the algorithm. These properties being convergence, causality preservation, and intention preservation in the general case. It then presents admissibility as a more rigorous, and provably correct, alternative to intention preservation.

Further research into the possibility of dynamically moving the transform step of ABT to the server in situations where a client has low processing power or battery life, such as a mobile phone, would be interesting. It is an open problem to implement ABT with undo and redo ability, it is currently not possible for a user to undo or redo operations in the TIPS framework we discussed. The TIPS framework also does not support operations on XML elements.

Operational Transformation is a very useful algorithm which can solve a wide range of CSCW problems. This algorithm, and others like it, will only become more important in the future, as remote collaboration becomes more popular.

5. ACKNOWLEDGEMENTS

I would like to thank Elena Machkasova for advising, proofreading, and technical advice. I would also like to thank Chad Seibert and Eugene Butler, for proofreading and suggestions as well.

6. REFERENCES

[1] D. Li and R. Li. An analysis of intention preservation in group editors. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 348–349, New York, NY, USA, 2007. ACM.

[2] R. Li and D. Li. Commutativity-based concurrency control in groupware. In *Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on*, page 10 pp., 0-0 2005.

[3] B. Shao, D. Li, and N. Gu. A sequence transformation algorithm for supporting cooperative work on mobile devices. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, CSCW '10, pages 159–168, New York, NY, USA, 2010. ACM.

[4] B. Shao, D. Li, T. Lu, and N. Gu. An operational transformation based synchronization protocol for web 2.0 applications. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, CSCW '11, pages 563–572, New York, NY, USA, 2011. ACM.

[5] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5:63–108, March 1998.

[6] D. Wang, A. Mah, and S. Lassen. Google wave operational transformation. July 2010.