How does Google Docs use Operational Transformation to provide real time collaboration?

I. Introduction

Today, individuals and businesses take real time online collaboration tools as a given. All of the major online document editing platforms, including Google Docs, Office 365, and iWork for iCloud, now enable people across the globe the unrestricted ability to contribute to the same document and see in real time the changes made by other collaborators.

The aim of real time collaborative systems is to provide a mechanism for distributed collaboration, without imposing pessimistic restrictions on what each user can edit to avoid race conditions, without the latency of client-server round trips to confirm that a user's changes do not conflict with concurrent changes made by another user and so can be committed to the document, and without the need for users to manually resolve conflicts when merging edits. With such a goal in mind, the local replicas of the shared document will necessarily diverge as each editor makes changes which are committed immediately locally for low latency, and yet take time to propagate to the local replicas of other editors. The challenge is for the system to be able to integrate the changes made by all editors in such a way that the original intention of each change is preserved, no data is lost unnecessarily, and once all changes have been propagated and applied to all local replicas of the document, all the versions of the document have converged to the same state.

The conception of the Operational Transformation (OT) technique to solve this problem originates from the 1989 paper *Concurrency control in Groupware systems* [1] with the proposed dOPT algorithm used by the GROVE groupware editor. Google's OT algorithm, which debuted with Google Wave in 2009 (now an open source Apache project) and has been the basis for collaboration in Google Docs since May 2010, has a lineage which traces back to the 1995 paper *High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System* [2], which itself is based on the dOPT algorithm in the original 1989 paper.

In this paper, we provide an introduction to the problems faced by real time collaborative editors and the ways that OT based algorithms can solve such problems; we explore in detail the Google OT algorithm which is the basis of real time collaboration in Google Docs today; we define the properties required to prove correctness of general OT based algorithms; and finally prove the correctness of the Google OT algorithm.

A. *Framework for discussing OT algorithms*

In a general real-time group editing session, there are multiple geographically distributed sites, each with an editor which maintains a local replica of a shared document. Editing operations can be generated by any editor at any time. When an operation is generated at a site, it is immediately applied to the local replica, and then propagated to the other sites. Upon arrival at another site, it is possibly transformed, and then applied to the local replica at that site.

To define a specific group editing application, one must specify the set of states $\mathcal{S}$ that each replica of the document can be in at any time, the set of operations $\mathcal{O}$, and a transition function $\Delta : \mathcal{S} \times \mathcal{O} \to \mathcal{S}$ which encodes how to apply each operation to each state to produce a new state. Throughout this paper, we often consider a simple text document group editing application. In this case, $\mathcal{S}$ is the set of all strings, which we think of as a zero-indexed lists of characters; and $\mathcal{O}$ is the set of all operations of the form `Insert(i, str)` or `Delete(i, len)` where `i` and `len` are integers and `str` is a string. Given a state $s \in \mathcal{S}$, we define $\Delta(s,$`Insert(i, str)`$)$ to be the state s' obtained by inserting the string `str` into s so that the first character of `str` ends up at index `i` in s'. Also, given a state $s \in \mathcal{S}$, we define $\Delta(s,$`Delete(i, len)`$)$ to be the state obtained by deleting `len` characters from s, starting at index `i`.

Throught this paper, we will use space-time diagrams, such as the one in Figure 1, to represent the progression of a real-time group editing session. Each vertical line corresponds to one of the editors. The sequence of states of that replica are in rectangles spaced out along the line in time increasing order from top to bottom. The operations which cause transition from one state to the next are in rounded rectangles and in between the states of the replica before and after the operation was applied. In the basic style of diagram, propagation of operations from one site to another is represented by a line connecting the operation from the original site to the operation where it was applied. In some of the later diagrams where we depict the Google algorithm, we instead show the exact contents of the messages which are sent from one site to another.

II. Consistency issues

The literature has identified three main inconsistency problems which could arise in a real time group editing session: (1) divergence, (2) causality violation, and (3) intention violation [1], [3], [5].

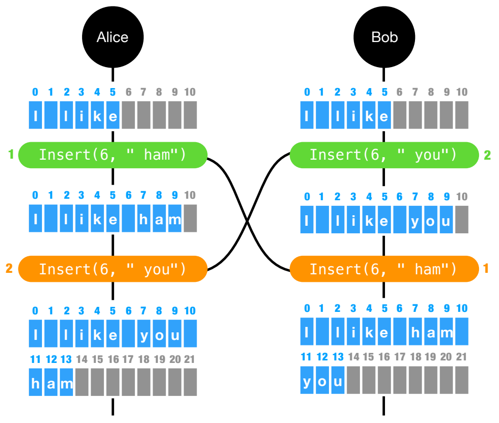


Fig. 1. Example of divergence. In this space-time diagram, Alice and Bob start with a document containing the text "I like". Alice appends the text " ham", and concurrently, Bob appends the text " you". Both of these operations are immediately committed to their local document replicas, and then propagated to the other collaborator. After applying each other's received operations, Alice and Bob arrive at the different document states "I like you ham" and "I like ham you" respectively.

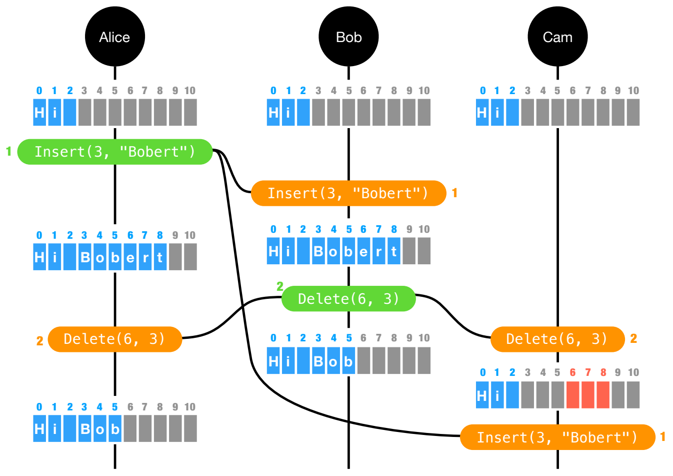


Fig. 2. Example of causality violation. In this example, there are three participants - Alice, Bob and Cam, who all start in sync with the document "Hi ". Alice appends the text "Bobert", and her change is received first by Bob, who subsequently deletes the ending "ert" to correct Alice's mistake. Unfortunately Cam receives Bob's delete operation before he receives Alice's insert operation, and so the delete operation isn't well defined on Cam's local replica of the document at that point in time.

A. Divergence

Consider the group editing session for a simple text document depicted in Figure 1. Notice that the final states of Alice and Bob's local copy of the document are different. This is called divergence. Divergence occurs when non-commutative operations are applied in different orders to the local replicas at different sites.

B. Causality violation

Causality encompasses the idea that the generation of operations at a site is dependent on the history of operations which have already been applied to the replica at that site. In the literature [3], [5], [6], the following definitions of causal ordering of operations, and concurrent operations are popular.

Definition 1 (Causal ordering relation): Let O_a and O_b be operations generated at sites i and j respectively. Then O_a is said to causally precede O_b, denoted by $O_a \to O_b$, if and only if either

1) $i = j$ and the generation of O_a happened before the generation of O_b, or
2) $i \neq j$ and the execution of O_a at site j happened before the generation of O_b, or
3) there is an operation O_x such that $O_a \to O_x$ and $O_x \to O_b$

Definition 2 (Concurrency relation): Let O_1 and O_2 be generated operations. Then O_1 and O_2 are concurrent, denoted by $O_1 \parallel O_2$, if and only if neither $O_1 \to O_2$ nor $O_2 \to O_1$.

We always assume that when a sequence of messages is sent from one editor to another, the messages arrive in the order they are sent. This is a reasonable assumption which can usually be guaranteed by a communication channel. However, when more than two editors are involved in the propagation of operations, there is still the potential that an editor could receive operations out of causal order. We call such a situation a causal violation. In the best case, a causal violation will confuse a user by the appearance of an effect before its cause in their local document replica. In the worst case, a causal violation could result in an editor trying to apply a received operation onto a state for which that operation isn't well defined.

As an example, consider the group editing session for a simple text document depicted in Figure 2. Cam received operation 2 before operation 1, even though operation 1 was generated before operation 2. Here, the causal violation occurred because the communication channels between Alice and Bob, and between Bob and Cam, were much faster than the communication channel between Alice and Cam. When Cam tries to apply operation 2 to his local document replica, he finds that he can't, because the characters which operation 2 is trying to delete don't yet exist in this replica of the document. The delete operation isn't well defined on Cam's local document state.

C. Intention violation

The intention of an operation is the editing effect that operation has when applied onto the document state from which it was generated. For particular applications, the precise nature of the intention of each operation must be specified in order to formally reason about whether the intention of an operation is preserved when it is applied in a context different to that from which it was generated.

In our simple text editing application, the intention of the operation `Insert(i, str)` when applied to a state s, a string of length n, is that in the resulting document state, `str` comes after characters 0 to $i-1$ of s, and before characters i to $n-1$ of s. Similarly, the intention of the operation `Delete(i, len)` when applied to a state s, is to delete the characters `i` to `i + len` in s.

Consider now the group editing session for a simple text document depicted in Figure 3. Operation 1 is generated by

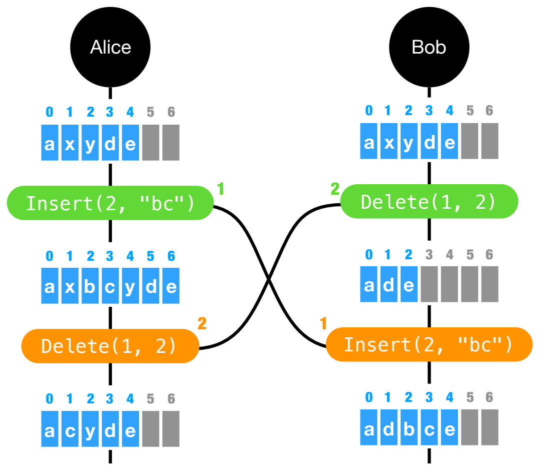


Fig. 3. Example of intention violation.

Alice on the state “axyde”, so its intention is that “bc” is inserted after the characters “ax” and before the characters “yde”. For its intention to be preserved when it is applied by Bob to the state “ade”, “bc” should be inserted after the character “a” and before the characters “de”. Here we have an intention violation because “bc” is actually inserted after the “d” to result in the state “adbce”. Similarly, the intention of operation 2 is that the characters “xy” are deleted, but when operation 2 is applied to the state “axbcyde”, the characters “xb” are deleted instead, and so this is another intention violation.

Intention violation is subtly distinct from divergence. It would be possible to solve divergence with a protocol which ensures all operations are executed at all sites in the same order (if such a protocol could exist). Nonetheless, such a protocol cannot always solve intention violation. For instance, in the above example, if serialisation could guarantee that operation 1 and 2 would always be executed in the same order, then both Alice and Bob’s local replicas would converge to the same document state — “acyde” if operation 1 is always done before operation 2, or “adbce” if operation 2 is always done before operation 1. However, we have already seen that the intention of operation 2 is violated in the first case, and that of operation 1 in the second.

D. Consistency

A real-time group editing application is considered to be consistent if it satisfies the following three properties

1) Convergence property. All replicas of the shared document are identical after all generated operations have been executed at all sites.
2) Precedence property. If operation O_a causally precedes operation O_b, then O_a is executed before O_b at each site.
3) Intention preservation property. The effect of all generated operations when applied to local replicas of the document matches the intention of those operations when they were generated.

III. Operational Transformation

The basic idea that OT algorithms use to achieve convergence, and to eliminate intention violation, is that when an editor receives an operation that was generated at another site, the editor transforms that operation against all operations which weren’t present in the replica of the document for which the operation was generated, so as to account for the effects of these operations. The editor applies the transformed operation to its local replica rather than the original. An OT algorithm consists of two pieces — a transformation function $T : \mathcal{O} \times \mathcal{O} \to \mathcal{O}$ where $T(O_1, O_2)$ is the result of transforming operation O_1 so that it accounts for the effect of operation O_2, and a control algorithm which decides when messages between editors are sent and received, what information they contain, and when and how operations are transformed against each other We also define an additional transformation function $\mathcal{X} : \mathcal{O} \times \mathcal{O} \to \mathcal{O} \times \mathcal{O}$ (also known in the literature as `xform` [2]) where $\mathcal{X}(O_1, O_2) = (O_1', O_2')$ where $O_1' = T(O_1, O_2)$ and $O_2' = T(O_2, O_1)$.

We introduce the concept of a *state-space diagram* to better understand convergence and how operational transformation can help us achieve convergence. We call the sequence of states exhibited by the local replica of an editor as it applies a sequence of operations a trajectory in state space. Consider now the trajectories of a pair of editors X and Y which start in the same state. We assume the precedence property holds, and so in particular, the sequence of operations generated by X are always applied at all sites in the order they are generated, and the same holds for the sequence of operations generated by Y. It follows that we can represent the trajectories of X and Y through state space as paths in an integer lattice which start at the origin and only move in the positive X or Y directions by increments of a unit. The vertices (x, y) correspond to states where x of the operations generated by X and y of the operations generated by Y have been applied to the starting state, and edges from (x, y) to $(x+1, y)$ represent applying the $(x + 1)$-th operation generated by X (after possibly transforming it), and edges from (x, y) to $(x, y + 1)$ represent applying the $(y + 1)$-th operation generated by Y (after possibly transforming it). For example, if the sequence of operations generated by X is X_1, X_2, and generated by Y is Y_1, Y_2, then the blue trajectory in Figure 4 corresponds to applying the sequence of operations X_1, X_2, Y_1, Y_2 to the starting state, and the green trajectory corresponds to applying the sequence of operations Y_1, X_1, Y_2, X_2 to the starting state. Given two trajectories in state-space which terminate at the same point (x, y), such as the blue and green paths in Figure 4, we say the state-space diagram *commutes* if two editors which start in the same state s arrive at the same state after one follows one of the trajectories and the other follows the other trajectory. Notice that if we can define a way to transform operations so that we can make any state-space diagram commute, then we can guarantee convergence. Here we have defined a state-space diagram for the trajectories taken in a system with a pair of editors, but the idea can be generalised to n editors by adding an extra dimension for each additional editor.

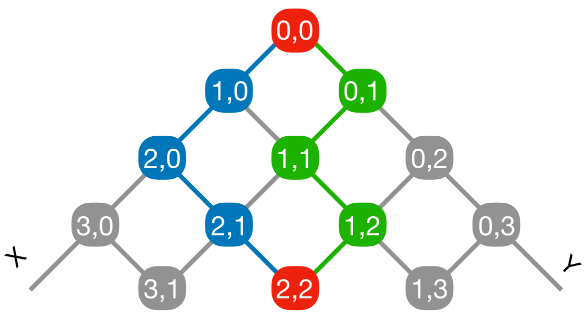


Fig. 4. A state space diagram.

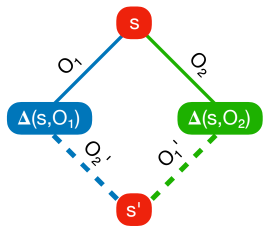


Fig. 5. State space diagram for TP1.

It has been shown in the literature that the following conditions TP1 and TP2 (also known as CP1 and CP2 in some papers) are sufficient conditions on the transformation function T to ensure convergence.

(TP1) For all states $s \in \mathcal{S}$ and all operations $O_1, O_2 \in \mathcal{O}$, if $(O_1', O_2') = \mathcal{X}(O_1, O_2)$, then

$$\Delta(\Delta(s, O_1), O_2') = \Delta(\Delta(s, O_2), O_1')$$

(TP2) For any three operations $O, O_1, O_2 \in \mathcal{O}$,

$$T(T(O, O_1), T(O_2, O_1)) = T(T(O, O_2), T(O_1, O_2))$$

We can represent (TP1) by the state-space diagram in Figure 5. Here, editor X generated operation O_1 and editor Y generated operation O_2. The blue path is the trajectory of X and the green path is the trajectory of editor Y. Operation O_1 is generated by X and applied immediately to X's local replica to produce state $\Delta(s, O_1)$. Concurrently, operation O_2 is generated by Y and applied immediately to Y's local replica to produce state $\Delta(s, O_2)$. Both operations are now broadcast and received by the other editor. Editor X transforms the received operation O_2 against the operation O_1 that it has already applied to s to get operation $O_2' = T(O_2, O_1)$, and then applies O_2' to its current state $\Delta(s, O_1)$, to produce state $\Delta(\Delta(s, O_1), O_2')$. Editor Y does a similar thing with the operation O_1 that it receives to produce operation $O_1' = T(O_1, O_2)$ and end up in state $\Delta(\Delta(s, O_2), O_1')$. Both editors have now applied both operations to their local replicas, and so for convergence to hold, they must be in the same state s'. This is equivalent to saying the state space diagram must commute. The requirement that $\Delta(\Delta(s, O_1), O_2') = \Delta(\Delta(s, O_2), O_1')$ in (TP1) ensures that convergence holds in this case, or equivalently, that the diagram commutes.

Consider now the more complicated scenario where editor X generates and applies operations X_1, X_2 in sequence, con-

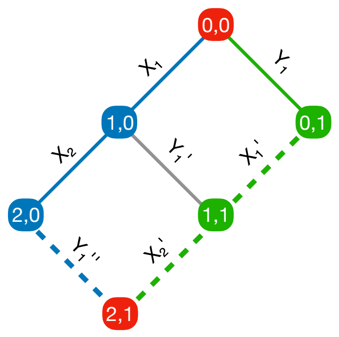


Fig. 6. Using TP1 to make a state space diagram commute.

currently with editor Y which generates and applies operation Y_1. Editor Y would like to transform the operations X_1 and X_2 which it receives to produce operations X_1' and X_2', and editor X would like to transform the operation Y_1 which it receives to produce the operation Y_1'', to make the state-space diagram in Figure 6 commute, where the trajectory of X is in blue and the trajectory of Y is in green. Note that if we assume the transformation function satisfies (TP1), then we can define $(X_1', Y_1') = \mathcal{X}(X_1, Y_1)$ and $(X_2', Y_1'') = \mathcal{X}(X_2, Y_1')$ so that the two smaller squares commute. It follows then that the whole diagram commutes, and so the local replicas of X and Y converge to the same state.

This trick of creating phantom operations, such as operation Y', in order to complete a commutative diagram and ensure two editors converge to the same state, can be extended to any arbitrary scenario. It is the responsibility of the control algorithm to remember sufficient information about the operations which have already been applied to each replica, and to ensure operations are processed in a particular order that allows it to use this trick to produce the transformed operations it needs to ensure each replica converges to the same final state.

As explained in [6], a sufficient condition for an OT algorithm to avoid the need for the transformation function to satisfy property (TP2) in order to have convergence, is that the control algorithm avoids transforming the same pair of operations in different state contexts. In [6], it is shown that Google's OT algorithm, which we describe in the next section, successfully avoids the need for its transformation function to satisfy (TP2).

As a final note, constructing a transformation function that satisfies even only (TP1) is difficult. Suppose we want to define the value of $T($`Insert(i, s)`, `Insert(j, t)`$)$. If `i<j`, the transformed operation should obviously be `Insert(i, s)` , and if `i>j`, it should be `Insert(i + length(t), s)`. What isn't clear is how to define the transformed operation when `i=j`, using only the information `i`, `j`, `s`, `t`. In order to satisfy (TP1), the function must choose which insert operation — its first argument, or its second — is performed first, in such a way that if its arguments are swapped, so is the order the inserts which are performed. Hence we can't naively choose to always apply the first insert first and the second one second in this case. One solution to this problem is to introduce a total ordering on all generated operations, such as the time that the operations are generated (and assume that it is extremely unlikely that two operations are generated at exactly the same time), and use this total ordering in the transformation function to decide which operation is applied first.

IV. Google's OT algorithm

Google's OT algorithm guarantees consistency using a combination of operational transformation and its control algorithm.

Unlike the general case where each editor sends all operations it generates to all the other editors, Google's OT algorithm has a single central server and multiple client editors where the clients only send and receive operations to and from the server.

Google's OT algorithm makes use of a transformation function T, named `transform` in code, which must satisfy property (TP1) for the algorithm to be correct. As already mentioned, it was shown in [6] that T doesn't need to satisfy (TP2). The cross transformation function $\mathcal{X}$, which we defined in terms of T, will be named `xform` in code.

The client editors track three pieces of information in addition to the state of its local replica of the document:

1) The number of the most recently received document revision from the server.
2) A queue of all locally generated operations which haven't yet been sent to the server.
3) A queue of all locally generated operations which have been sent to the server, but which haven't yet been acknowledged by the server.

The server tracks two pieces of information in addition to the state of its local replica of the document:

1) A queue of all operations it has received from clients, but hasn't yet processed. Here, it remembers also which client sent each operation and the number of the document revision that operation was generated on.
2) A list of all operations it has ever processed in the order that the server processed them, called the revision log. Each operation in the log corresponds to a new revision of the document and is associated with a revision number.

A. *Control algorithm*

The control algorithm is event driven. Here we describe the algorithm's response to different events, at both client and server sites.

1) When an operation is generated at a client site, it is immediately applied to that client's local replica and added to the end of that client's pending queue.
2) As soon as there are no operations in a client's sent queue, the client sends a message to the server which contains the number of the revision the client has most recently received from the server, as well as the first operation in its pending queue. It then moves that operation from the start of the pending queue to the end of the sent queue.
3) When a message is received by the server, the operation and revision number in the message and the name of the client who sent it, are added as an item to the end of the server's pending queue.
4) Whenever a server's pending queue isn't empty, it takes the first operation `c` off its pending queue. It transforms `c` against all operations in the log which have revision numbers greater than the revision number of the revision which `c` was defined on, to produce the operation `c'`. More precisely, using Haskell style syntax, if `log` is the list of operations from the log which `c` must be transformed against, then

   ```
   c' = foldR transform c log
   ```

   It then applies `c'` to its document replica and adds it to the end of the its revision log as a new revision. It then generates and sends a message to the client who generated the operation acknowledging that it was received, and a message to each other client containing the transformed operation `c'`.
5) When an acknowledgment message is received by a client, the first operation on its sent queue is removed and the client's last received version number is incremented.
6) When a message containing an operation `s` is received by a client, `s` is transformed against all of the operations in the client's sent and pending queues in the order they were generated, to produce `s'`. At the same time, all of the operations in the client's sent and pending queues are transformed against the operation `s` and the results are used to update both queues in place. More precisely, using Haskell style syntax, if `sent` and `pend` are lists of the operations in the client's sent and pending queues, then

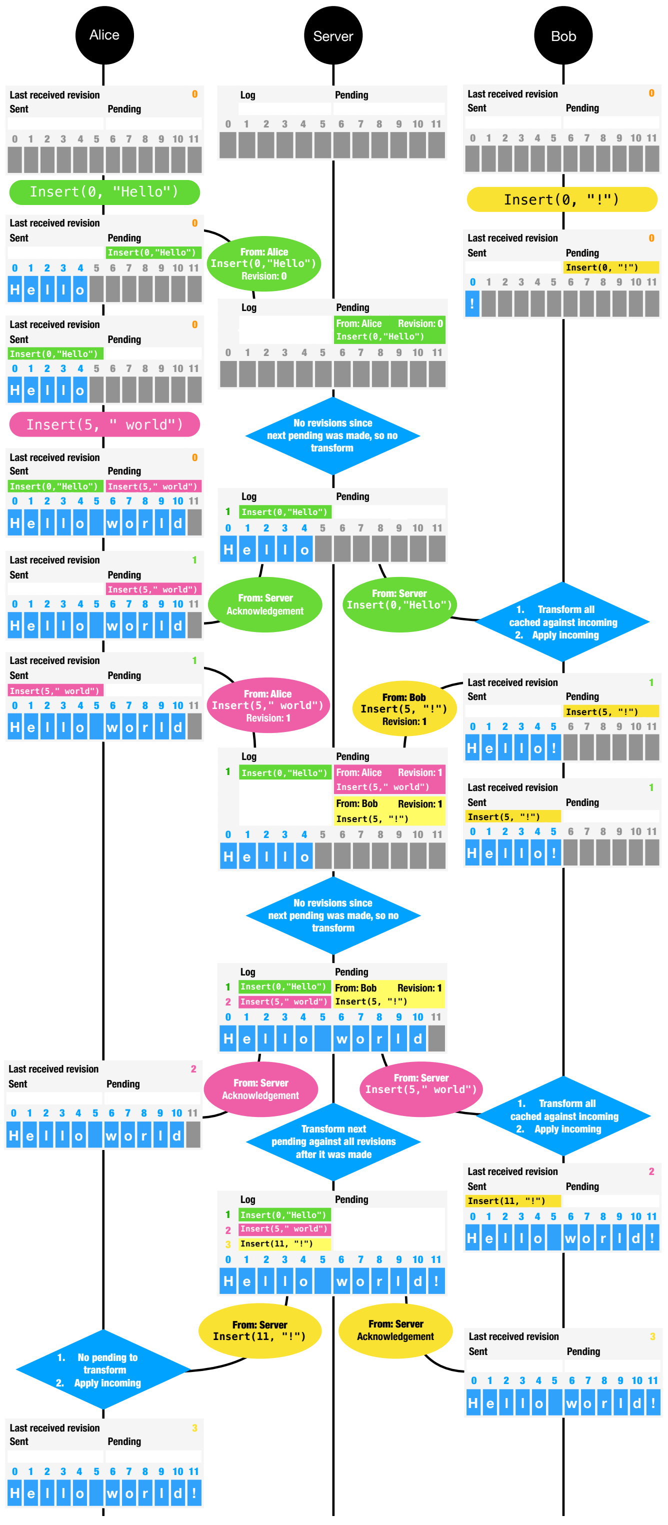
   ```
   (s'', sent') = mapAccumR xform s sent
   (s', pend') = mapAccumR xform s'' pend
   ```

   where `xform` is an implementation of the cross transform function X.

Refer to the space-time diagram in Figure 7 for an example of how the Google control algorithm achieves convergence and intention preservation in a particular editing session involving the editors Alice and Bob in addition to the server.

B. *Convergence*

We would like to see how operational transformation and the control algorithm work together to achieve convergence in general. To do so, we can restrict ourselves to considering state trajectories of a single client-server pair. In this pair, we distinguish between operations which are generated by the client, and operations generated by the server. Here, operations generated by the server are really all the operations the server receives from the other clients that we are ignoring. One can show that the control algorithm guarantees that both client and server process all generated operations in causal order. Nonetheless, it should be clear, at least, that all operations generated by the client are always executed in the order they are generated, and all operations generated by the server are always executed in the order they are generated. It follows that



Alice
Server
Bob
Last received revision
Sent
Pending
Log
Insert(0, "Hello")
Insert(0, "!")
Insert(5, " world")
From: Alice Insert(0,"Hello") Revision: 0
No revisions since next pending was made, so no transform
From: Server Acknowledgement
From: Server Insert(0,"Hello")
1. Transform all cached against incoming 2. Apply incoming
From: Alice Insert(5," world") Revision: 1
From: Bob Insert(5, "!") Revision: 1
From: Server Insert(5," world")
Transform next pending against all revisions after it was made
From: Server Insert(11, "!")
1. No pending to transform 2. Apply incoming

Fig. 7. Real time collaboration session using Google's OT algorithm, based on [9].

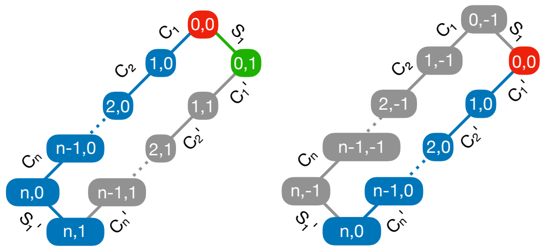


Fig. 8. Situation 1.

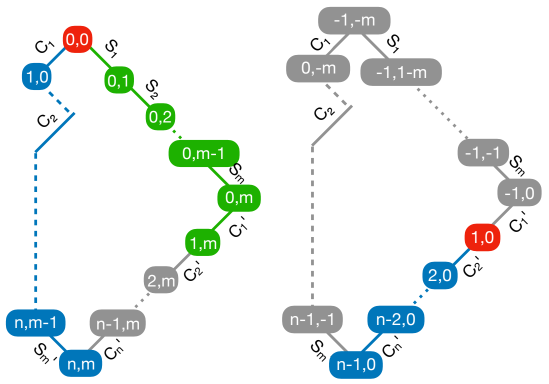


Fig. 9. Situation 2.

we can think about this algorithm in terms of the state-space diagram for a pair of editors [10], defined already in Section 4.

There are two main state-space situations which the algorithm allows the client and server to enter, which are depicted in Figures 8 and 9 below. Using the arguments below and proof by induction, one can show that once no more new operations are generated, both client and server will eventually converge to the same state.

In the first situation, depicted in Figure 8, the client and server started in sync at state $(0,0)$, but the client has since applied and sent operation C_1, which hasn't yet been acknowledged, and also has applied operations $C_2, C_3, ...C_n$ which are currently stored in its pending queue, so that its trajectory so far is the solid blue path to $(n,0)$. Also suppose that from state $(0,0)$, the server has received and applied an operation S_1 from another client, so its trajectory is the solid green path to $(0,1)$. At this point the server sends the client a message containing the operation S_1. The client uses S_1 and its cached list $C_1, ..., C_n$ to produce S_1' and $C_1', ..., C_n'$ as already described. It applies S_1' to its local replica, to arrive at state $(n,1)$, and updates its cached operations so that the sent queue contains C_1' and the pending queue contains $C_2', ..., C_n'$. If we now translate the coordinate system so that $(0,1)$ becomes $(0,0)$, we are now in a situation equivalent to one where the server is in the newly labeled state $(0,0)$, and the client started in sync with the server at $(0,0)$, but has since reached state $(0,n)$ by applying the sequence of operations $C_1', \cdots, C_n'$.

In the second situation, depicted in Figure 9, the client and server also started in sync at state $(0,0)$, but the server has since received and processed the server operations $S_1, ..., S_m$ (that is, operations which originated from other clients), so that its trajectory is the green path to $(0, m)$. Also, concurrently, the client has applied and sent operation C_1 to the server, which now is at the front of the server's pending queue, and then has followed a trajectory involving the other client generated operations $C_2, ..., C_n$, as well as the transformed server operations $S'_1, ..., S'_m$, so that its trajectory is the blue path to (n, m) (the dotted lines indicate 0 or more state transitions). As per the algorithm, the client's sent queue currently contains the operation C'_1, and its pending queue the operations $C'_2, ..., C'_n$, a result of transforming these operations against all the incoming server operations. These are the operations which the server would need to apply in order to converge with the client's state at (n, m). The server now takes C_1 off its pending queue, and transforms it against all the operations $S_1, S_2, ..., S_m$ which have occurred since the state that C_1 was generated on, in order to produce the new operation C'_1. Note that this C'_1 is the same operation as the C'_1 that the client currently has in their sent queue, because both client and server have applied the same transformations to C_1 to produce this operation. The server applies C'_1 to arrive at state $(0, m+1)$, adds C'_1 to the end of its log, and then sends an acknowledgment to the client, so that the client can remove C'_1 from its sent queue. Because its sent queue is now empty, the client moves the first operation C'_2 in its pending queue to its sent queue and sends this operation to the server. If we now translate the coordinate system so that $(1, m)$ becomes $(0, 0)$, the ending situation is equivalent to one where the server is in the newly labeled state $(0, 0)$, and the client started in sync with the server at $(0, 0)$, but has since applied the $n-1$ operations $C'_2, \cdots, C'_n$ to reach state $(n-1, 0)$, with C'_2 in its sent queue and $C'_3, ..., C'_n$ in its pending queue.

C. Stream operations

Google Wave (now Apache Wave) used the Google OT algorithm to support real time collaborative editing of XML documents [11]. It treated XML documents as a sequence of *items*, where the gaps between items are called *positions*. Each 16-bit Unicode character, start tag, and end tag constitutes an item. Position 0 comes before the first item. For example, we can break the XML document

`"<body><p>Hi</p></body>"`

into the list of items

`["<body>","<p>","H","i","</p>","</body>"]`

and the gap before `"<body>"` is position 0, the gap between `"H"` and `"i"` is position 3, and the gap after `</body>` is position 6.

To support efficient transformation and application of many operations, Wave introduced the idea of streaming operations. A streaming operation is a sequence of ordered document mutations. The main mutations which wave supports

- `Skip n`. Skip stream pointer ahead `n` items in the document before applying the next mutation.
- `InsertString str` inserts a new character item for each element in the string at the current stream pointer position. Doesn't advance the stream pointer position.
- `InsertStartTag name attributes`. For example, `InsertStartTag "img" ["src", "hello.jpg"]` would insert the tag `<img src="hello.jpg">`. Doesn't advance the stream pointer position.
- `InsertEndTag name`. For example, `InsertEndTag img` would insert the tag `</img>`. Doesn't advance the stream pointer position.
- `Delete n`. Deletes the next `n` items after the stream pointer's current position. Advances the stream pointer position by `n` items.
- `SetTagAttributes attributes`. Doesn't advance the stream pointer position.
- `UpdateTagAttributes attributes`. Doesn't advance the stream pointer position.

Each mutation applies to the current position of the stream pointer in the original document, and then advances the stream pointer as described. Hence one can process and apply the mutations in order as one scans the document linearly. Google designed these mutations specifically so that streaming operations can be composed together and transformed against each other efficiently, using stream-based algorithms which process their arguments linearly from start to end.

Let $N([m_1, m_2, \ldots, m_k])$ denote the number of items from the original state that the sequence of mutations $[m_1, m_2, \ldots, m_k])$ applies to and $M([m_1, m_2, \ldots, m_k])$ denote the number of items in the final state the sequence of mutations $[m_1, m_2, \ldots, m_k]$ relates to. We also define $N(m) = N([m])$ and $M(m) = M([m])$ for more convenient notation. We have $N($`[Delete n]`$) = n$, $N($`[InsertString str]`$) = 0$, and $N($`Skip n`$) = n$. Let We have $M($`[Delete n]`$) = n$, $M($`[InsertString str]`$)$ `=length str`, and $M($`Skip n`$) = n$.

If x is a stream mutation such that $N(x) > 0$, then for any integer partition $[n_1, n_2, \ldots, n_k]$ of $N(x)$, there is an operation $[x_1, x_2, \ldots, x_k]$ which is equivalent to x, where all the mutations $x_1, x_2, \ldots, x_k$ have the same type as x, and $N(x_j) = n_j$. For example, if x `=InsertString "Hello"`, $N(x) = 5 > 0$. The partition $[1, 3, 1]$ of 5 gives the equivalent operation `[InsertString "H", InsertString "ell", InsertString "o"]`. Similarly, if x is a stream operation such that $M(x) > 0$, then for any integer partition $[n_1, n_2, \ldots, n_k]$ of $M(x)$, there is an operation $[x_1, x_2, \ldots, x_k]$ which is equivalent to x, where all the mutations $x_1, x_2, \ldots, x_k$ have the same type as x, and $M(x_j) = n_j$. For example, if x `=Delete 5`, $M(x) = 5 > 0$. The partition $[1, 3, 1]$ of 5 gives the equivalent operation `[Delete 1, Delete 3, Delete 1]`. For the purposes of algorithmic analysis, we will assume that we have functions which can split a mutation x into an operation $[x_1, x_2]$ for any partition $[n_1, n_2]$ of $N(x)$, and for any partition $[n_1, n_2]$ of $M(x)$, as described above, in constant time. This is not strictly true. For example, to split a string of length k into two strings, one of length $j < k$ and the other of length $k - j$, requires $\mathcal{O}(k)$ steps. Finally, if x_1 and x_2 are two operations with all mutations of the same type, then we can merge them into a single operation x such that $N(x_1) + N(x_2) = N(x)$

and $M(x_1) + M(x_2) = M(x)$.

D. Stream operation transformation

We would now like to specify the algorithm to transform stream operation $o_2 = [y_1, y_2, \cdots, y_m]$ against $o_1 = [x_1, x_2, \cdots, x_n]$ to produce o_2'.

If o_2 is empty, then we are done. If o_1 is empty, then we append o_2 to o_2' and then we are done. Otherwise neither o_1, nor o_2 are empty.

We keep taking mutations off o_1 and o_2 until we find the largest i and j such that $N(y_1) = N(y_2) = \ldots = N(y_i) = 0$ and $N(x_1) = N(x_2) = \ldots = N(x_j) = 0$. Let $y = [y_1, \ldots, y_i]$ and $x = [x_1, \ldots, x_j]$. If $i > 0$ and $j = 0$, then we append the sequence of mutations y to o_2 and repeat the whole procedure. If $i = 0$ and $j > 0$, then we insert the mutation `Skip` $M(x)$ at the end of o_2, and then repeat the whole procedure. Otherwise, both i and j are positive. As was already discussed in section 4, in order to ensure that (TP1) holds, we must use a total ordering on all generated operations, such as the time they were generated, to decide whether the sequence of mutations y or x should be done first. For example, if o_2 was generated before o_1, and both o_1 and o_2 try to insert text at position 0, then we might decide that o_2's insert mutation should occur before o_1's insert mutation. If we decide that y should be done first, then we append the sequence of mutations $[y_1, \ldots, y_i,$`Skip` $M(x)]$ to o_2, and then repeat the whole procedure. Otherwise we append the sequence of mutations [`Skip` $M(x), y_1, \ldots, y_i]$ to o_2, and repeat the whole procedure.

Otherwise, $i = j = 0$, and this means that $N(x_1) > 0$ and $N(y_1) > 0$. We take the first mutation y_1 off the front of o_2, and then keep taking mutations off the front of o_1, until we find the first k such that $N([x_1, x_2, \ldots, x_k]) \geq N(y_1)$. If $N([x_1, x_2, \ldots, x_k]) > N(y_1)$, we split the mutation x_k into the equivalent sequence of mutations $[a, b]$ such that $N(b) = N([x_1, x_2, \ldots, x_k]) - N(y_1)$, and insert b at the front of o_1. Let o denote the operation $[x_1, x_2, \ldots, x_{k-1}, a]$ if we did the split, and $[x_1, x_2, \ldots, x_k]$ otherwise. We now do an operational transform of the mutation y_1 against the sequence of mutations o, to produce another sequence of mutations y_1'. For example, if $y_1 =$`Delete 5`, then for any operation o, the transformed mutation is $y_1' =$`Delete j`. If $y_1 =$`Skip 2` and $o =$`[Delete 1, Skip 1]`, then $y_1' =$`[Skip 1]`. The algorithm used here to transform y_1 against o will depend on the specific mutation y_1. We will assume, for the purpose of algorithmic analysis that this transformation can be done in $O(j)$ time where j is the number of mutations in o. Finally, we append the sequence of mutations y' to o_2', and then repeat the procedure.

Either at the end, or as the output stream o_2' is being constructed, we merge adjacent mutations of the same type. The whole process for transforming an operation with m mutations against an operation with n mutations can be done in $\mathcal{O}(m+n)$ time because the algorithm processes each mutation in o_1 and o_2 exactly once.

As an example of how the stream-based operational transformation algorithm works, in Figure 10 we see the transformation of operation

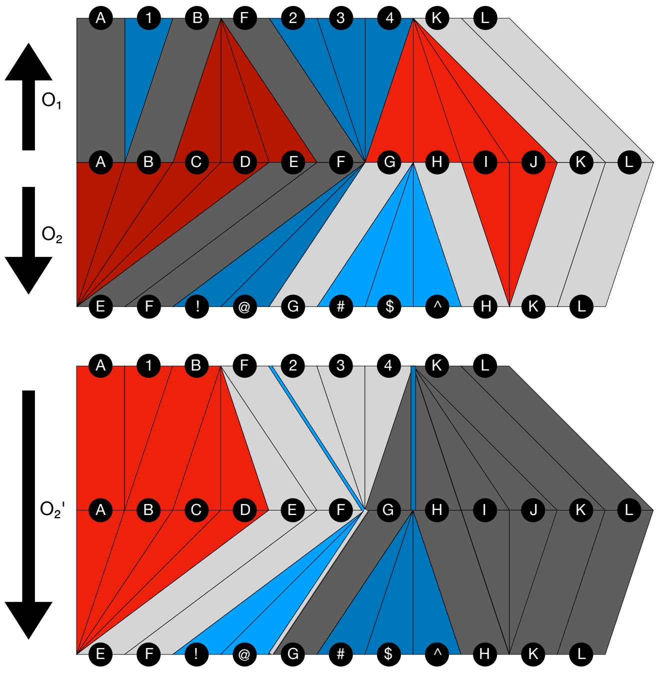


Fig. 10. An example of the stream transformation algorithm. Here, the middle line represents the original document state, the top one the document state after the execution of operation O_1 and the bottom after the execution of O_2. Red indicates a deletion, blue represents an insert, and grey represents a skip mutation. In the top sub-diagram, we have used darker colour to indicate the mutations which have already been processed, and lighter colour to indicate those yet to be processed. In the bottom sub-diagram, the darker colours indicate mutations which are yet to be produced whilst lighter colours indicate mutations which have already been produced.

```
o1 = [Skip 1, InsertString "1", Skip 1,
      Delete 3, Skip 1,
      InsertString "234",
      Delete 4, Skip 2]
```

against operation

```
o2 = [Delete 4, Skip 2, InsertString "!@",
      Skip 1, InsertString "#$^",
      Skip 1, Delete 2, Skip 2]
```

At this point in its execution, the algorithm has already processed the first three mutations of O_2 to produce the first three mutations of O_2'. As the algorithm progresses, the boundary between the light and dark regions, corresponding to the mutation in O_2 which is currently being processed, will move to the right. The final transformed operation will be

```
o2' = T(o2 o1) =  [Delete 3, Skip 1,
      InsertString "!@", Skip 3,
      InsertString "#$^", Skip 2]
```

E. Stream operation composition

Let $o_1 \circ o_2$ denote the operation obtained by composing operation o_1 with o_2, that is, an operation with the effect of first doing o_2 and then doing o_1. We require that composition satisfies the following properties:

(C1) For all $s \in \mathcal{S}$ and all $o_1, o_2 \in \mathcal{O}$,

$$\Delta(s, o_1 \circ o_2) = \Delta(\Delta(s, o_2), o_1)$$

(C2) If $\mathcal{X}(a_1, b) = (a_1', b')$ and $\mathcal{X}(a_2, b') = (a_2', b'')$, then $\mathcal{X}(a_2 \circ a_1, b) = (a_2' \circ a_1', b'')$.

(C3) If $\mathcal{X}(b, a_1) = (b', a_1')$ and $\mathcal{X}(b', a_2) = (b'', a_2')$, then $\mathcal{X}(b, a_2 \circ a_1) = (b'', a_2' \circ a_1')$.

(C4) For all operations $o_1, o_2, o_3 \in \mathcal{O}$, $(o_1 \circ o_2) \circ o_3 = o_1 \circ (o_2 \circ o_3)$. That is operation composition is associative.

(C5) There is an operation $1 \in \mathcal{O}$ such that for all $o \in \mathcal{O}$, $o \circ 1 = 1 \circ o = o$.

We would now like to specify the algorithm to compose stream operation $o_2 = [y_1, y_2, \cdots, y_m]$ with $o_1 = [x_1, x_2, \cdots, x_n]$ to produce o_3, where $o_3 = o_2 \circ o_1$. We assume that $M(o_1) = N(o_2)$ so that the composition of the operations is well defined. In practise, we can pad either operation with a sufficiently long skip as its last mutation in order to satisfy this requirement. If o_1 is empty, then we append o_2 to o_3 and then we are done. Similarly, if o_2 is empty, then we append o_1 to o_3 and then we are done. Otherwise neither o_1 nor o_2 is empty.

If $M(x_1) = 0$, we take x_1 off the front of o_1 and add it to the end of o_3, and then repeat the procedure. Else, if $N(y_1) = 0$, we take y_1 off the front of o_2 and add it to the end of o_3 and repeat the procedure. Notice that if $M(x_1) = 0$ and $N(y_1) = 0$, then it doesn't matter which order these operations are added to o_3 as both orders will have the same effect on any state. Otherwise, $M(x_1) \neq 0$ and $N(y_1) \neq 0$.

Notice that to compose a mutation x with a mutation y to produce a mutation z, we require $M(y) = N(x) > 0$. If $M(x_1) > N(y_1)$, we split x_1 into the equivalent operation $[x, x_1']$ so that $M(x_1') = M(x_1) - N(y_1)$, add x_1' to the front of o_1 and compose $y = y_1$ with x to produce z. Else if $M(x_1) < N(y_1)$, we split y_1 into the equivalent operation $[y, y_1']$ so that $N(y_1') = N(y_1) - M(x_1)$, add y_1' to the front of o_2 and compose y with $x = x_1$ to produce z. Else $M(x_1) = N(y_1)$, and we compose $y = y_1$ with $x = x_1$ to produce z. In all cases, we add z to the end of o_3, and then repeat the procedure.

Assuming that the composition of any mutation x with any mutation y can be done in constant time, an operation with m mutations and an operation with n mutations can be composed in $\mathcal{O}(m + n)$ time because the algorithm processes each mutation in o_1 and o_2 exactly once. After completing this procedure, o_3 has $O(m + n)$ mutations. It is likely that there are subsequences of adjacent mutations in o_3 which have the same mutation type, and these can be merged together in a single pass of the list in $O(m + n)$ time, or this merging can be done in the original algorithm when mutations are inserted at the end of o_3. Even with merging, the number of mutations in o_3 is $O(m + n)$.

As an example of how the composition algorithm works, in Figure 10, we see the partial progress of composing operation

```
o2 = [Delete 4, Skip 2, InsertString "!@",
  Skip 1,         InsertString "#$^",
  Skip 1, Delete 2, Skip 2]
```

with

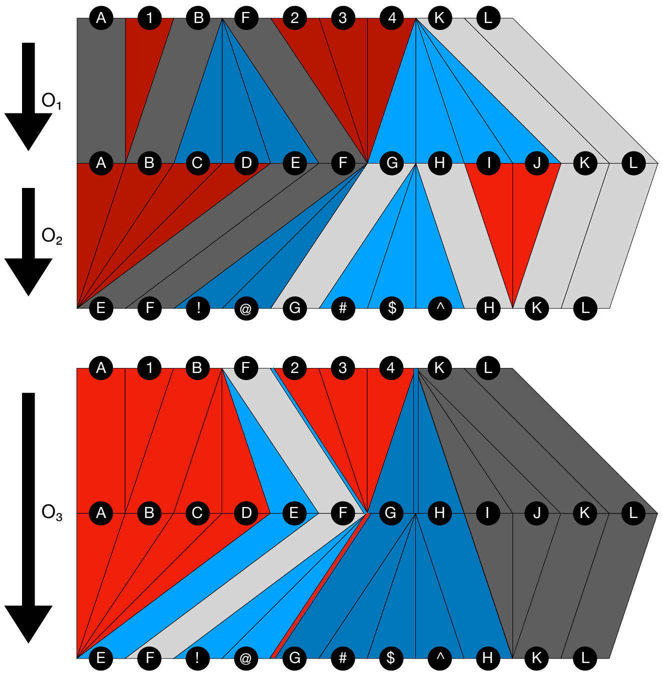


Fig. 11. An example of the stream composition algorithm. The top line represents the original document state, the middle one the document state after the execution of operation O_1, and the bottom after the subsequent execution of O_2. Red indicates a deletion, blue represents an insert, and grey represents a skip mutation. In the top sub-diagram, we have used darker colour to indicate the mutations which have already been produced, and lighter colour to indicate those yet to be produced. In the bottom sub-diagram, the darker colours indicate mutations which are yet to be produced whilst lighter colours indicate mutations which have already been produced.

```
o1 = [Skip 1, Delete 1, Skip 1,
  InsertString "CDE", Skip 1, Delete 3,
  InsertString "GHIJ", Skip 2]
```

At this point in its execution, the algorithm has already processed the first six mutations of o_1 and the first three mutations of o_2. In its next step, since $M($`Insert "GHIJ"`$) > N($`Skip 1`$)$, it will split the operation `Insert "GHIJ"` into the operations `[Insert "G", Insert "HIJ"]`, so that it can compose `Skip 1` with `Insert "G"` to produce the operation `Insert "G"` which it adds to the end of o_3. In the following step, since $N($`Insert "#$^"`$) = 0$, it adds `Insert "#$^` to the end of o_3. The final composed operation will be

```
o3 = [Delete 3, Insert "E", Skip 1,
Insert "!@", Delete 3,
Insert "G#$^H", Skip 2]
```

after merging of adjacent mutations of the same type.

F. Composition trees

A monoid is any data type T with an binary operation $\circ : T \times T \to T$ such that $\circ$ is associative and T has an identity element. Monoids capture the idea of function composition, and in particular, our stream operations are monoids under operation composition, due to the requirements (C4) and (C5). Let T be a monoid where each element x of T has an

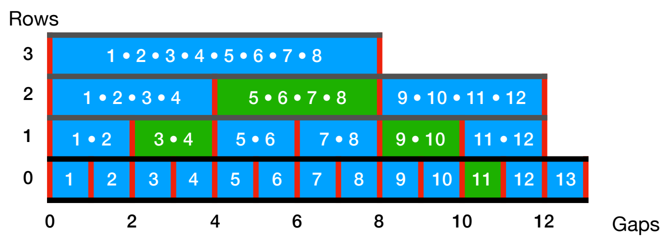


Fig. 12. Depicts a composition tree for the list $L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]$. The rows are labeled with their row numbers, starting with row 0 at the bottom. The vertical red lines are the gaps between elements in the tree. The gaps between elements in row 0 are labelled with their numbers, starting with gap 0 before element 1. The green path is the path chosen by the sublist composition algorithm, when applied to the sublist $L(2, 9) = [3, 4, 5, 6, 7, 8, 9, 10, 11]$ of L. The algorithm first composes 11 with the identity to form element 11, then it composes $9 \circ 10$ with the result of the first computation to form $9 \circ 10 \circ 11$, and then $5 \circ 6 \circ 7 \circ 8$ with the result of the second computation to form $5 \circ 6 \circ 7 \circ 8 \circ 9 \circ 10 \circ 11$, and finally $3 \circ 4$ with the result of the third computation to form the desired $3 \circ 4 \circ 5 \circ 6 \circ 7 \circ 8 \circ 9 \circ 10 \circ 11$.

associated length, denoted $\ell(x)$, and for any two elements x and y, $x \circ y$ has $\mathcal{O}(\ell(x) + \ell(y))$ many elements, and can be computed in $\mathcal{O}(\ell(x)+\ell(y))$ many steps. Our stream operations also satisfy these additional requirements, as we've already shown.

Let $L = [x_1, \ldots, x_n]$ be a list of elements of T, and let $L[i, m]$ denote the sublist $[x_{i+1}, \ldots, x_{i+m}]$ for any $0 \leq i \leq n$ and $0 \leq m \leq n - i$. Also let $\ell(i, m) = \sum_{k=1}^{m} \ell(x_{i+k})$ be the total length of the sublist $L[i, m]$, and $C(i, m)$ denote the composition of the elements in $L[i, m]$. That is, $C(i, m) = x_{i+m} \circ \cdots \circ x_{i+2} \circ x_{i+1}$. We'd like to be able to compute the composition $C(i, m)$ efficiently for any $0 \leq i \leq n$ and all $0 \leq m \leq n - i$. We can do this by storing the elements of L in the base of a composition tree.

A composition tree consists of multiple rows of operations, where each element in the tree is the composition of the two elements directly beneath it, which we call its left and right child. We number the rows 0, 1, 2, ... from bottom to top. The elements of the underlying list L form row 0 of the tree. The $(j + 1)$-th row of the tree is constructed by pairing up adjacent elements of the j-th row of the tree, and composing them together. Clearly, all elements in the j-th row of the tree are ultimately a composition of the 2^j elements in row 0 beneath it. On a computer, a composition tree could be stored in a linked data structure similar to a skip list.

In composition trees, we like to think in terms of the gaps between the elements in level 0. We number the gaps left to right, starting with gap 0 before the first element. Thinking in terms of gaps, $L[i, m]$ is the sublist of L between gap i and gap $i+m$. For any element x in a composition tree, we define $N(x)$ to be the number of operations in level 0 from which x is composed.

Assuming we have already constructed a composition tree whose base is the list L, we can compute $C(i, m)$ in time $\mathcal{O}(\ell(i, m) \log m)$ time in the following way. If $m = 0$, then we return the identity element. Otherwise, we find the highest row which contains an element x whose left most endpoint is gap i and for which $N(x) \leq m$. We then compute $C(i+N(x), m-N(x))\circ x$ recursively. An example of the algorithm's execution is illustrated in Figure 12.

It is clear that the algorithm is correct, given that the monoid laws hold for T, because each of elements it composes lie over different sublists of the tree's base elements $L[i, m]$, all elements of $L[i, m]$ are covered by an element which the algorithm composes into the result, and the algorithm does all compositions in the correct order.

We first show that the algorithm composes $\mathcal{O}(\log m)$ many elements.

The path through the tree which the algorithm chooses will only ever contain at most two adjacent elements from the same row. Assume, for a contradiction, that the path contains three adjacent elements from the same row, say elements l, m, r in that order. Since the elements in the row above are formed by composing elements in this row in pairs, there must be an element c in the layer above which is either the composition of l and m, or of m and r. Additionally, c has a left most endpoint which is above the rightmost endpoint of some other element in the path (either the element before l in the path, or l itself), so c was eligible to be chosen by the algorithm instead of the left child of c. Moreover, the rightmost endpoint of c is no further right than the right most endpoint r, and so l (c) would have been less than or equal to the value of m at the time the algorithm was considering c. This is a contradiction because the algorithm should have chosen c instead of c's left child.

Once the algorithm starts descending the tree, it will never ascend again. Suppose the algorithm is at gap i on level l, the terminating gap is m base elements ahead, and the algorithm chooses level $h < l$ for its next element e. This means that $2^{h+1} > m$, which implies $m/2 - 2^h < 0$ and so $m - 2^h < m/2$. Since element e is on level h, we have $N(e) = 2^h$, and so the number of base elements which must be covered by the remainder of the path after e is less than $m/2 < 2^{h+1}/2 = 2^h$. It follows that the element chosen by the algorithm after e must come from a level whose number is less than h.

The highest level the algorithm can ever reach is level $\lfloor \log m \rfloor$. If m isn't a power of 2, then $\lceil \log m \rceil = \lfloor \log m \rfloor + 1$. Since $2^{\lceil \log m \rceil} > m$, the algorithm would never choose an element from level $\lfloor \log m \rfloor + 1$. If m is a power of two, then $\lfloor \log m \rfloor = \log m$, and clearly $2^{\log m+1} = 2m > m$ and so the algorithm would also never choose an element from level $\lfloor \log m \rfloor + 1$.

Since the algorithm only ever stays in the same level for two adjacent elements, always increases when it can, can only ever go as high as level $\lfloor \log m \rfloor$, and once it starts decreasing, it continues to decrease until it reaches the finishing point, it composes at most $2+2(2\lfloor \log m \rfloor) = \mathcal{O}(\log m)$ many elements and so does $\mathcal{O}(\log m)$ many compositions.

Suppose at some point in its execution, the algorithm must compose elements x and y, which have themselves been constructed by composing the base elements $[x_a, \ldots, x_b]$ and $[x_c, \ldots, x_d]$ at some point in the past. It is clear that $b+1 = c$ by the way the algorithm works, and so the time taken to compute $x \circ y$ is $O(\ell(x) + \ell(y)) = O(\ell(x_a) + \ldots + \ell(x_d)) = O(\ell(x_i) + \ldots + \ell(x_{i+m})) = O(\ell(i, m))$.

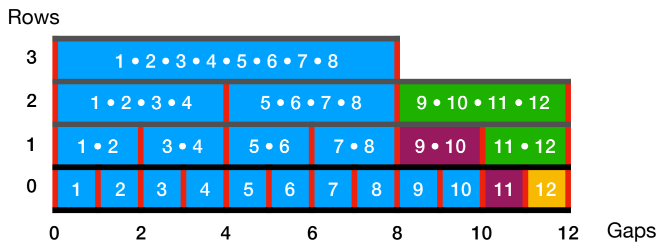


Fig. 13. An example of the insertion algorithm for a composition tree. The orange element 12 is the element to insert. The green elements are the ones which must be created to complete the tree. The purple elements are the elements in the top layer of the tree before insertion, which are involved in compositions during the insertion. First we compose 11 and 12 to produce the parent of 12, and then $9 \circ 10$ with $11 \circ 12$ to produce the parent of $11 \circ 12$. No more elements need to be created, because $9 \circ 10 \circ 11 \circ 12$ will be the left child of the element above it when it is eventually created, or alternatively, because the gap to the right of element 12 at the base will not extend any higher than layer 2 in the tree.

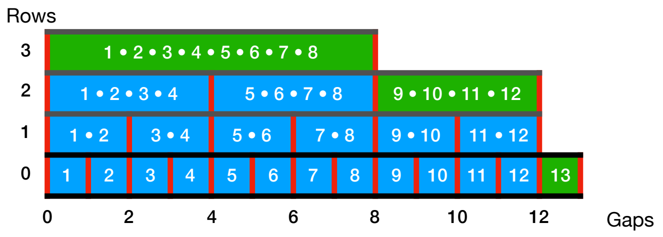


Fig. 14. An example of the top layer of a composition tree. The elements highlighted in green form the top layer of this composition tree. These are the only elements one needs to keep in order to compute $C(0, 13)$. The corresponding composition list would be $[1 \circ 2 \circ \cdots \circ 7 \circ 8, 9 \circ 10 \circ 11 \circ 12, 13]$.

The non-constant time operations the algorithm does are searching for the next level in a gap, and doing the compositions. Since the highest level the algorithm can ever reach is level $\lfloor \log m \rfloor$ and m is an upper bound for the number of gaps where it does a search on the levels, the total time searching for next levels in gaps is $\mathcal{O}(m \log m)$. Since there are $\mathcal{O}(\log m)$ merges, each of which can be done in $\mathcal{O}(\ell(i, m))$ time, all the merges can be done in $\mathcal{O}(\ell(i, m) \log m)$ time. Since $\ell(i, m) > m$, the time taken to do composition dominates, and the time complexity of the overall algorithm is $\mathcal{O}(\ell(i, m) \log m)$.

Given a composition tree built on top of the list L, one can insert an element x of length s at the end of the tree by performing the compositions needed to fill in, from the bottom of the tree to the top, all the newly generated empty spaces for elements whose right endpoint is the gap at the right endpoint of s. See Figure 13 for an example of the insertion process. Since the tree has height $\lfloor \log n \rfloor$, this process involves at most $\log n$ compositions. One can build the whole composition tree on top of L from scratch by inserting the elements of L one by one in sequence. To do this, $\lfloor \log n \rfloor$ layers are built. The time taken to build layer $j+1$ is the time taken to do compositions of the elements in layer j, and this is on the order of the total length of layer j, which itself is on the order of the total length of the elements in layer 0. Hence the total computational time spent building the tree on top of L is $\mathcal{O}(\ell(0, n) \log n)$.

Suppose the elements of the list L are generated sparsely over a relatively long time period. Suppose also that at any time instant where the sublist of L generated so far is $L[0, m]$ where $0 \leq m \leq n$, then we are only ever interested in computing $C(0, m)$ at this time. Then the composition algorithm to compute $C(0, m)$ will always use the elements in the highest layers of the tree in any given column, as shown in Figure 14. Also, to insert the next of L element into the tree built so far, which will produce elements which end up in the highest layer of the tree, we only need to compose with elements which were in the highest layers of the tree before the insert. It follows that in this case, we only need to keep track of the highest layers of the tree, and so the storage requirements for the tree decrease from $\mathcal{O}(\ell(0, n) \log n)$ to just $\mathcal{O}(\ell(0, n))$. We call such a data structure a composition list, and it can be stored on a computer using a linked list.

G. Performance of stream operations

In the original Google OT algorithm, without streaming operations, if a client generated n operations and the server generated m operations concurrently, then $\mathcal{O}(mn)$ operational transformations are required to bring them back in sync with each other. Indeed, if we have a state space diagram where the client's trajectory goes along a line to $(n, 0)$ and the server to $(0, m)$, then we need to compute the transformed operations corresponding to all mn edges in the rectangle $(0, 0)$, $(0, m)$, $(n, 0)$, (n, m) in order to make the client and server converge at (n, m). Of course, in practise, these $\mathcal{O}(mn)$ transformations are spread out over time that the operations are generated.

Google claims in [11] and [12] that it can use these stream based operations and composition trees to reduce the processing time to reach convergence in this scenario to $\mathcal{O}(m \log m + n \log n)$. With streaming operations, the clients will keep a composition list of pending operations, and the server will keep a whole composition tree as its log.

Suppose the server wants to process a client operation c which contains n mutations. It must transform that operation against all the server operations which have occurred since client and server were last in sync. Suppose the total number of mutations in these server operations is m. Since they are stored in a composition tree, their composition s can be calculated in $\mathcal{O}(m \log m)$ time. We can then compute the cross transform of c and s against each other in $\mathcal{O}(m + n)$ time.

Suppose the client receives a server operation s containing m mutations. It must compute the cross transform of this operation against all the elements of its sent and pending queues to update the sent and pending queues to be based off the new server state, and to find a transformed operation s' of s which is based off the local document replica. Suppose the total number of mutations in the composition list is n. It first computes the composition c of all elements of the composition list, in $\mathcal{O}(n \log n)$ time. It does the cross transform of s against the sent queue, to produce s' and a new sent queue, and then s' against c in $\mathcal{O}(m+n)$ time to produce s'' and c'. It can apply s'' to its local document replica, and start a new composition list for the pending queue containing only the operation c'.

We find that $\mathcal{O}(n \log n + m \log m)$ time is spent in total, accounting for the transformations and compositions both client and server spend at the time when processing the two operations. Note that we haven't accounted for the time spent building the composition lists from scratch, nor even the time spent doing the insertions of the operations into the composition lists.

The improvements of using stream operations and composition are in fact more dramatic then appears from this time complexity analysis. Indeed, rather than clients only being able to send changes one at a time to the server, and having to wait for a long client-server round trip between sending adjacently generated changes, with composition, the clients can take their pending queue (which is the top of a composition tree of all the client's generated pending mutations), compose all the operations in the pending queue together in $\mathcal{O}(n \log n)$ time where n is the total number of mutations in all the operations in the composition list, and then produce efficiently and send to the server a single stream operation which contains all the changes which have been made since the last time client and server were in sync.

REFERENCES

[1] C. A. Ellis and S. J. Gibbs, *Concurrency Control in Groupware Systems.* Austin, Texas:ACM, 1989. David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping

[2] D. A. Nichols, P. Curtis, M. Dixon and J. Lamping, *High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System.* Palo Alto, California: Xerox PARC, 1995.

[3] C. Sun and C. Ellis,, *Operational Transformation in Real-Time Group Editors: Issues, Algorithms and Achievements.* , 1998.

[4] D. Sun and C. Sun, *Context-Based Operational Transformation in Distributed Collaborative Editing Systems*, IEEE transactions on parallel and distributed systems, vol. 20, no. 10, October 2009.

[5] Y. Liu, Y. Xu, S. J. Zhang, and C. Sun, *Formal Verification of Operational Transformation*, Nanyang Technological University, Singapore, 2014.

[6] . Y. Xu, C. Sun, M. Li *Achieving Convergence in Operational Transformation: Conditions, Mechanisms and Systems*, Nanyang Technological University, Singapore, 2014.

[7] J. Day-Richter, *Whats different about the new Google Docs: Working together, even apart*, https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_21.html, Tuesday, September 21, 2010.

[8] J. Day-Richter, *Whats different about the new Google Docs: Conflict resolution*, https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_22.html, Wednesday, September 22, 2010.

[9] J. Day-Richter, *Whats different about the new Google Docs: Making collaboration fast*, https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html, Thursday, September 23, 2010.

[10] D. Spiewak, *Understanding and Applying Operational Transformation*, http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation, 17 May, 2010.

[11] D. Wang and A. Mah, *Google Wave Operational Transformation*, https://github.com/scrosby/fedone/blob/master/whitepapers/operational-transform/operational-transform.rst, Google Wave White Papers, May 2009.

[12] C. Whitelaw, D. Danilatos, A. Mah, D. Wang, *Google Wave: Under the hood*, https://www.youtube.com/watch?v=uOFzWZrsPV0, Google I/O 2009.