Multi-char CDM operations

David Barrett-Lennard
20 Feb 2008

Introduction
In this paper we provide an algorithm for operational transform on multi-character CDM operations on text documents. It extends the work from [1] which only concerned single character operations.

CDM = Create, Delete, Move

To support multi-characters we use half open intervals to efficiently represent ranges of characters to be created, deleted or moved.

Information preserving merging
In this implementation we are interested in support for *merging* of operations that is *information preserving*. Basically this means that the merge of operations is compatible with IT and ET. Nevertheless we still refer to merging as providing *log compression* - the hope being that half open intervals will coalesce where possible, reducing the storage requirements and subsequent processing time.

From [2] we have the following:

Given contextually serialised operations $[O_1, O_2]$, we define $O_1 \oplus O_2$ as the merge of O_1, O_2 into a single composite operation..

We require the following properties

MP1 $(O_1 \oplus O_2) \oplus O_3 = O_1 \oplus (O_2 + O_3)$ (associativity)

MP2 $S + [O_1\ O_2] = S + (O_1 \oplus O_2)$

MP3 $IT(O_1, O_2 \oplus O_3) = IT(O_1, [O_2, O_3])$

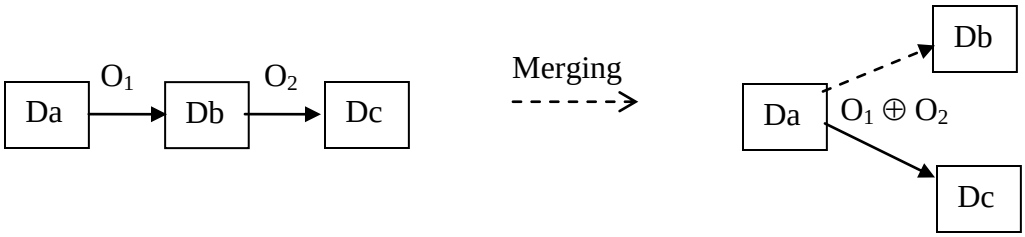
MP4 $IT(O_1 \oplus O_2, O_3) = IT(O_1,O_3) \oplus IT(O_2, IT(O_3, O_1))$

Aliasing extraction intervals
In order to support information preserving closure under merging of concurrent operations we allow for multiple extraction intervals to alias the same characters in an effects document.

In the interests of a well understood declarative meaning, we assume that characters can only be aliased at the post-operation locations. Therefore there will never be aliasing with the extraction of an enabled move operation (because the characters are really somewhere else).

Merging of chaining move ops
Let Da, Db, Dc refer to three different locations for a contiguous range of characters in the effects document.

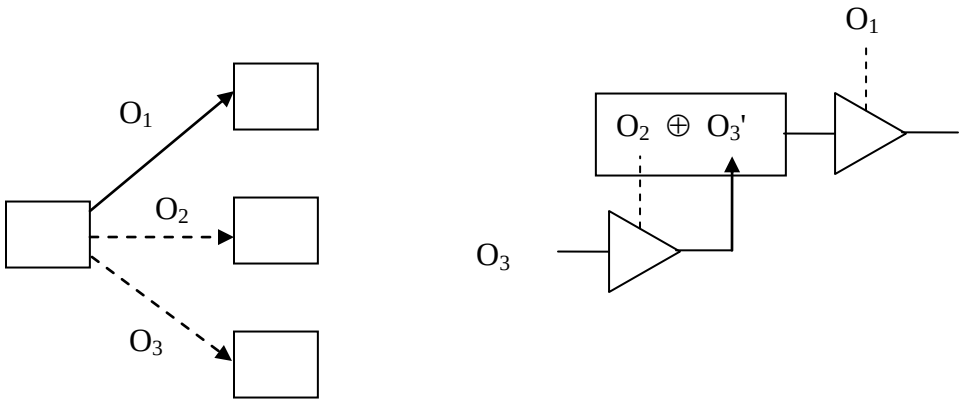
Let O_1 move from Da to Db, and O_2 from Db to Dc. Then $O_1 \oplus O_2$ moves directly from Da to Dc.



Our approach is to still think of $O_1 \oplus O_2$ as two distinct moves (as before), except there is no sense in which its internal representation can be regarded as a time ordered sequence of moves. Basically this means that the extraction of O_2 must be ET'd backward past the move performed by O1. The result is that both of the moves alias the same extraction location Da.

Extraction intervals must support aliasing

Aliased extraction intervals aren't particularly pleasant for an implementation. So it's worth establishing that they cannot be avoided.



Let O_1,O_2,O_3 all move the same character to different locations with $O_1 \parallel O_2 \parallel O_3$ and O_1.id < O_2.id < O_3.id.

Let $O_3' = IT(O_3,O_2)$. Consider that we merge $[O_2,O_3']$ into a single operation and transform past O_1. Then we expect that we will have two disabled extraction intervals that each track to the destination position of O_1.

Conclusion : Merging requires that we support aliasing of extraction intervals.

Deletes must be in post creation coordinates

Let the initial state S consist of a single character 'a' in the effects document, marked as present and not deleted.

S	effects doc	a
	present flags	1
	deleted flags	0

On state S, we apply O_1 = insert 'b' at q = 0, to yield the following state

S + O_1	effects doc	ba
	present flags	11

	deleted flags	00

On this state we apply O_2 = delete 'b' at q = 0, to yield the following state

$S + O_1 + O_2$	effects doc	ba
	present flags	11
	deleted flags	10

This example shows that in order to support merging of operations we must use post insertion coordinates for the deletion O_2 in order for it to be able to reference the character inserted by O_1.

Extractions must be in post creation coordinates
Let the initial state S consist of a single character 'a' in the effects document, marked as present.

S	effects doc	a
	present flags	1

On state S, we apply O_1 = insert 'b' at q = 0, to yield the following state

$S + O_1$	effects doc	ba
	present flags	11

On this state we apply O_2 = move b at q = 0 to q = 2, to yield the following state

$S + O_1 + O_2$	effects doc	bab
	present flags	011

This example shows that that in order to support merging of operations we must use post insertion coordinates for the extraction by O_2 in order for it to be able to reference the character inserted by O_1.

Declarative rule on extraction positions
When we merge operations we can have chains of moves - eg suppose we have the sequence of operations $[O_1\ O_2\ O_3]$ where $O_1 \rightarrow O_2$ and $O_2 \rightarrow O_3$. O_1 moves a character, and this character is in turn moved by O_2, which in turn is moved by O_3.

Under merging (ie $O1 \oplus O2 \oplus O3$) we need to specify what happens to the extraction intervals.

There are two different concepts of character identity, defined as follows

Definition: Let a *w-character* refer to a character that is originally inserted by some operation, and the character may subsequently move around to different locations yet keeps its identity as a w-character. Note that at any given time a w-character exists in at most one location in the working documents. Note as well that once a w-character has been created, at any subsequent time the w-character will be marked as present at precisely one location in the effects documents, and irrespective of whether it has been marked as deleted.

Definition: Let an *e-character* refer to a particular character in a particular effects document. ie it is assumed that characters in different effects documents always represent different e-characters, even though they represent the same w-character. Let existing e-characters in effects documents keep their identity after performing insertions into the effects documents (which merely shift e-characters after the insertion position further to the right).

Definition: In state S, let E(S) be the set of all e-characters across all effects documents, and let W(S) be the set of all w-characters present (even if marked as deleted) in S.

Definition: Given e-character e, let wchar(e) refer to its associated w-character.

Definition: Given w-character w, let ecreate(w) be the uniquely defined e-character where w was originally inserted by a create operation.

Definition: Given $w \in W(S)$, let echar(S,w) be the uniquely defined e-character associated with the current position of w in state S.

Definition : For e-characters e_1,e_2 we write $e_1 \sim e_2$ if $wchar(e_1) = wchar(e_2)$. Note that ~ is an equivalence relation.

Aliasing of extraction intervals means that a single operation O may need to reference a given w-character more than once.

Rule: Let operation O be performed on state S. Let w be a given w-character. w may be involved in multiple moves or deletes. It is required that all extraction positions are associated with a well defined e-character x(S,w). This is defined as follows

if $w \in W(S)$ then $x(S,w) = echar(S,w)$
else $x(S,w) = ecreate(w)$

Implementation of CDM operations

A CDM operation records the following double linked lists on each text field of each object

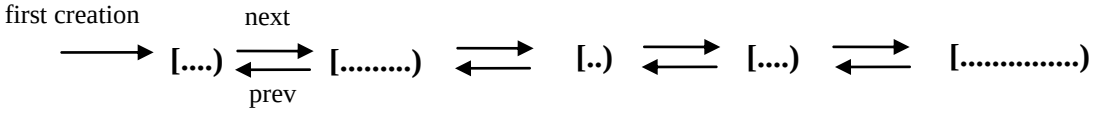
1. Creation intervals
2. Delete intervals
3. Move insertions
4. Move extractions

The implementation requires that every interval be non-empty.

These linked lists are homogenous, allowing the implementation to avoid run time polymorphism or discriminated unions.

Creations

The following depicts a double linked list of creation intervals recorded on a single text field. The intervals are ordered by q-position.



Creations intervals never overlap in q-coordinate.

Each creation interval records the following information

- The (s,t) associated with the operation that generated the creation

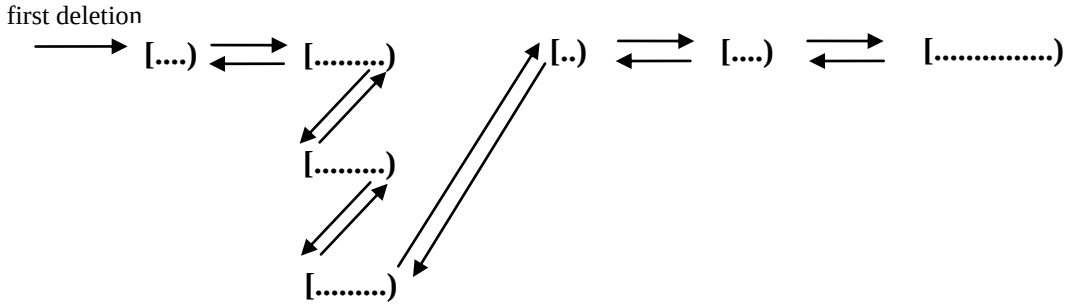
- The string being created
- The q-position for where to insert the string.

It is important to understand that an original generated operation can perform arbitrary many creations, deletions and moves, and under merging with other operations, we end up with linked lists of intervals with a mixture of (s,t) values. Hence the information for a single operation can be interspersed with other operations under merging.

The creation q-positions are expressed in post-creation coordinates. Conceptually this means we are specifying their final q-positions after the creations have all been applied (but before the moves). In practice this means we can apply the operation by simply iterating forwards through the linked list inserting the intervals as we go, knowing that each interval's q-position accounts for all the intervals that have already been inserted on its left.

Deletions

Deletion intervals are also stored in a double linked list ordered by q-position. However unlike creations, deletion intervals can overlap (alias). This corresponds to the case where multiple users have concurrently deleted the same text. The implementation splits intervals as required so that aliased intervals are always exactly equal in size. ie we require that the next interval either perfectly overlaps with the previous interval or else be further to the right, without any overlap.



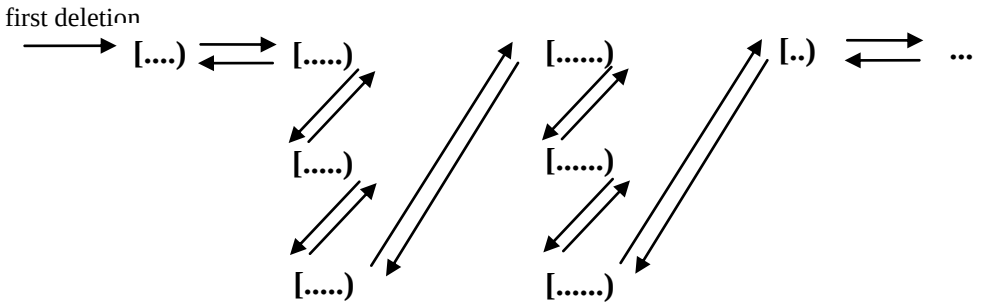
Each deletion interval records the following information

- The (s,t) associated with the operation that generated the deletion
- The string being deleted
- The q-position for where to delete the string.

In the implementation aliasing intervals are ordered by site identifier. Therefore the deletion intervals are totally ordered.

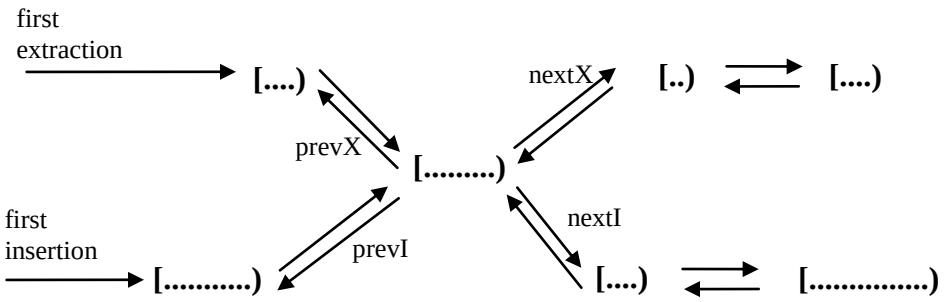
The deletions are expressed in post-creation coordinates.

During OT it may be necessary to split delete intervals. This following figire depicts the result after splitting.



Moves

A move represents both an extraction from one text field as well as an insertion into another. Therefore a move interval takes part in two double linked lists at the same time! In the following diagram a single move interval stores four pointers prevX, nextX, prevI, nextI, in order to take part in a double linked list of extractions (shown along the top of the figure), as well as a double linked list of insertions (show along the bottom of the figure). Each of these double linked lists is in q-position order.



Each text field records a double linked list of extractions and a double linked list of insertions. A given move interval can take part as an extraction on one field and an insertion in an entirely different field. ie we are able to model moves between different text fields (in different objects). Of course this representation allows for moves within a single text field or between different text fields in the same object.

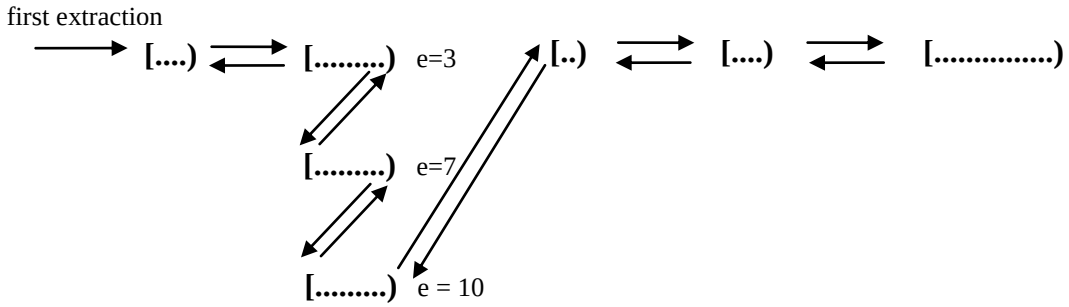
Each move interval records the following information

- (s,t) of the operation that generated the move
- e coordinate (used to pick a unique winner amongst competing moves)
- insertion q-position
- extraction q-position
- the value of the string being moved.

Moves extractions and insertions are expressed in post-creation, post-move-insertion, pre-move-presence coordinates.

Like creation intervals, move insertion intervals are strictly ordered by q-position without any overlap. ie they are not able to alias.

Move extraction intervals are able to alias in the same manner as deletion intervals. For example



As for deletion intervals, we require that the next interval either perfectly overlaps with the previous interval or else be further to the right, without any overlap.

The aliasing extraction intervals (which have the same q-coordinate) are ordered by e-coordinate. Note that e-coordinates will always be distinct so the ordering on all the extraction intervals is uniquely defined.

It must be kept in mind that each of the above extraction intervals independently takes part in a linked list of move insertions. As a result, splitting a move interval is a little tricky. It is necessary to split all the other move intervals that it aliases with as an extraction, and account for the fact that each interval takes part in two linked lists at the same time.

Ordering aliasing move extractions by e coordinate

Consider that we need to dual IT operations O_1, O_2 where both operations have performed around 100 moves of the same text interval.

The dual IT of extractions against extractions needs to perform the following

if (O_2.e < O_1.e || O_2.e == O_1.e && O_2.id < O_1.id) ++O_1.e; else ++O_2.e;

to transform e coordinates (used to select a unique winner amongst all competing move operations).

In this scenario there would be 100 x 100 = 10000 combinations to compare. A linear version of the algorithm involves sorting the competing moves by e-position and using an accumulating shift in the e-position, reducing time complexity to 100 + 100 = 200 - a significant saving.

The extractions of competing moves are sorted by e-position. Furthermore they are expressed in post insertion coordinates in the hypothetical effects document associated with e-positions. Repeated e-coordinates should be impossible. The following is an example

```
[-----)     [--------) e=0       [--)      [-----)
            [--------) e=3                 [-----)
            [--------) e=5                 [-----)
            [--------) e=6
            [--------) e=20
```

Thinking in terms of post insertion e-coordinates is very powerful - it uniquely forces certain behaviour when merging operations. For example suppose $O_1 \rightarrow O_2$ and we have chaining moves. Then when we merge to get $O_1 \oplus O_2$, we will need to 1) track O_2's source position back to O_1's source position (so all competing moves are collected together) and 2) bump O_1's e-position to allow for the insertion to the left by O_2 (which is deemed to dominate O_1). So we see that merging can cause e-positions to be adjusted for causally dependent operations!

Applying an operation

Applying an operation is performed in four phases

1. Apply creations. This involves insertion of characters that are marked as present but not deleted.
2. Apply deletions. This involves setting the deleted status (if not set already)
3. Apply move insertions. This involves inserting characters that are marked as present if and only the intervals has e = 0, and the deleted status is false.
4. Apply move extractions. For each interval that has e = 0, we mark the source character as no longer present, and move the deleted status.

Merging

$[O_1\ O_2] = [\ I_1\ \ X_1\ \ \ I_2\ \ \ X_2\ ]$
$\qquad\qquad\qquad\quad X$
$= [\ I_1\ \ I_2\ \ \ X_1'\ \ X_2\ ]$ $\qquad X_1' = IT(X_1,I_2)$
$\qquad\qquad\qquad\qquad\qquad\quad$ (shift X_1 to right to account for insertions by I_2)

$= [\ \ I_2'\ \ \ \ X_1'\ \ X_2\ ]$ $\qquad I_1' = IT(I_1,I_2)$
$\qquad\ \ I_1'\qquad\qquad\qquad\quad$ (shift I_1 to right to account for insertions by I_2)

$= [\ \ I_2'\ \ \ \ X_1'\ \ \ \ ]$ $\qquad X_2' = (O_1.type == MOVE\ \&\&\ X_2 == I_1')\ ?\ X_1'\ :\ X_2$
$\qquad\ \ I_1'\ \ \ \ X_2'\qquad\quad$ (track X_2 to X_1' for overlaps with I_1')

$= [\ I\ X\ ]\qquad\qquad\qquad\quad I = merge(I_1',I_2'),\ \ X = merge(X_1',X_2')$

All q-positions in the merged result [I X] will be expressed in the coordinate system associated with performing all insertions - ie both I_1 and I_2. I_2 and X_2 are already expressed in this coordinate system and therefore don't need to be shifted.

Both I_1 and X_1 are expressed in post I_1 coordinates and therefore require shifting to account for the insertions by I_2.

Let O_1 be performed on state S. Then [I X] will also be performed on state S. Therefore any e-char referenced by X_1 doesn't require adjustment (tracking). However X_2 may possibly require tracking - in those cases where X_2 matches an e-char inserted by a move by O_1.

We test X_2 against I_1' not I_1 so they are in the same coordinate system (ie post I_1 and I_2).

We track X_2 to X_1' not X_1 because it is necessary to account for the insertions I_2.

Repository

The repository is characterised by storing all operations from when the effects documents were initially empty. This allows the intervals to avoid the need to store q-positions. Instead, q-positions (and p-positions) or intervals are thought of as a function of a given vector time.

Consider what x(S,w) reduces to when S is the initial state of the system where all effects documents are empty.

if $w \in W(S)$ then $\quad x(S,w) = echar(S,w)$
else $\qquad\qquad\quad x(S,w) = ecreate(w)$

becomes

x(S,w) = ecreate(w)

ie it suggests that move intervals should always reference the original insertion position of a given w-char.

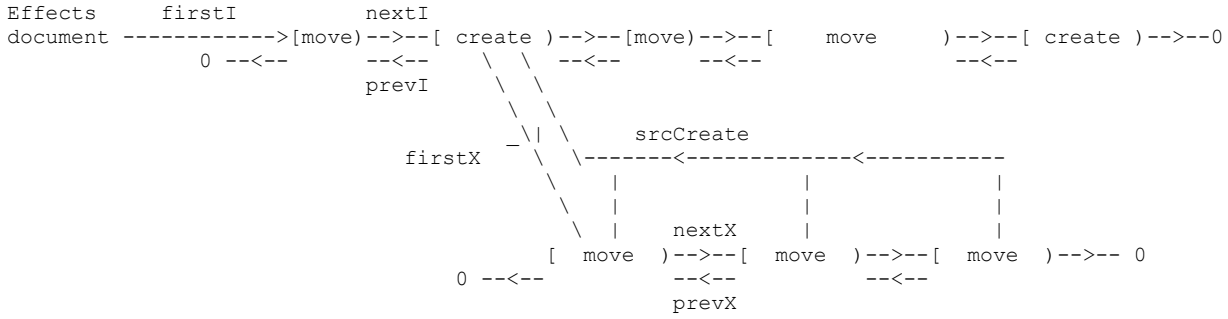
In that light, consider the following model

```
// abstract base
struct Insert
{
   SiteId s;
   int t;
   Insert* prevI;
   Insert* nextI;
};

struct Create : public Insert
{
   String str;
   set<pair<SiteId,int> > deleted;
   Move* firstX;      // can be NULL if no moves have been performed
};

struct Move : public Insert
{
   Create* srcCreate;
   Move* prevX;
   Move* nextX;
};
```

There are two types of insertion interval : creation and move. Both represent an insertion of an interval into a particular effects document. An insertion interval stores pointers prevI, nextI in order to take part in the double linked list of insertion intervals in a total ordering (by q-position) within a given effects document.



Only a creation interval needs to store the string of characters being inserted. A move interval can quickly find its associated string because it stores a pointer 'srcCreate' directly to the unique creation interval that originally inserted the string being moved.

Pure deletions are handled by recording their (s,t) in a set within the creation interval that originally inserted the string. Deletions take precedence over moves (ie a string marked as deleted is deleted regardless of where it is moved).

All moves for a given string form a single double linked list using pointers Move::prevX and Move::nextX. Create::firstX points at the head of the linked list. The moves are ordered by e-position. Therefore it can be assumed that for a given vector time v, we can traverse the linked list in order and the first interval with $t < v(s)$ represents the real location of the string at vector time v. If no Move satisfies $t < v(s)$ then the string is assumed to reside at the creation position. In particular this is the case if there are no moves and firstX = NULL.

Calculating suffix/prefix of an operation for a given vector time
The idea is to avoid the whole idea of a history buffer and instead record all changes that have been made in a session in a single growing operation. Latecommers can be brought up to date very efficiently be sending them the operation.

If a client goes down and reconnects during a collaboration it should be possible to use the received vector time to decompose an operation O into two pieces O1,O2 such that $O \sim [O1\ O2]$, and O1 is in the context of the vector time, and O2 represents all the operations that need to be sent to the site. This allows the site to be brought up to date very efficiently. It also provides the "suffix" which is used to transform incoming operations during the normal processing of a collaborative session. Decomposition of the suffix is needed as operations are received that have an increasing context (ie associated vector time).

Implementation notes
Let an operation store a double linked list of move/create intervals. We certainly need polymorphism in this list.

The fact that moves will always reference ecreate(w) or echar(S,w) means that it is easy do execute a move operation without it actually storing the string. ie only Create intervals need to store the string.

To reduce the needs for polymorphism it is assumed that deletes are stored separately to moves. During IT, deletes and moves must be independently shifted right as required.

More radical again
Consider that creates and moves are not stored in the same double linked list!

O --> linked list of creates
linked list of moves
linked list of deletes

Let creates be in post-creation coordinates. Let moves/deletes be in post-create+move coordinates

This eliminates the need for polymorphic intervals. They are CDM operations (Create-Move-Delete)

Advantages of this approach

* it is more likely that adjacent creates/moves or deletes can merge.
* more space efficient - there is no redundant state, and no type indicator
* serialisation doesn't need dynamic creation
* algorithms are more readable
* algorithms are hard-coded against specific types so they're faster
* string is only stored by create intervals
* Specialisation to not supporting moves is trivialised.

References

[1]	David Barrett-Lennard. Operational transform - Single character move, insert and delete operations. Apr 2007
[2]	David Barrett-Lennard. Operational transform - Merging Operations. Sep 2005