Operational transform
Single character move operations

20 Sept 2006
David Barrett-Lennard

Abstract
This paper extends the work in [3] by applying the approach to *single character move operations*. Move operations have richer semantics than insert and delete operations in the sense that move operations can easily model insertions or deletions through the use of additional text buffers. However, the converse is false! Even though a move seems to be nothing more than a deletion followed by an insertion, the semantics are different in the case where a conflict occurs - when two users concurrently try to move the same character to different places. If this is modeled as a delete and insert then the result is that the character is duplicated. This is often undesirable, particularly when the operational transform is used for an array of object identifiers rather than an array of characters.

For example, in an application written by the author, a jigsaw program implemented the z-order of the pieces using an ordered array of pieces, and clicking on a piece caused it to move to the top in the z-order. When this was implemented as a delete + insert it was found that pieces on the jigsaw could be duplicated when multiple users concurrently worked on the jigsaw.

Move operations have the nice feature of *conserving matter* in the system, under operational transform.

Introduction
As in [1], by definition characters in text documents have identity. A character is originally inserted by a particular user at a particular site. The character keeps its unique identity even though its index position in the document needs to be adjusted according to continued editing. Note that a character doesn't simply relate to its appearance (eg its ASCII code) - for example each appearance of the letter 'A' in a document represents a different character.

We take the convention that strings use zero based index positions. Given string s, let s[i] be the i^{th} character. s[i,j) denotes the sub-string corresponding to the half open interval [i,j).

Properties required to achieve convergence

TP1
$\forall\ O_1, O_2 \quad S + [O_1\ IT(O_2, O_1)] = S + [\ O_2\ IT(O_1, O_2)\ ]$

TP2
$\forall\ O_1, O_2, O_3, \quad IT(\ IT(\ O_3, O_1),\ IT(O_2, O_1)) = IT(\ IT(\ O_3, O_2),\ IT(O_1, O_2))$

Moves on the effects document
Before attacking the general problem, we first solve a slightly simpler problem by limiting ourselves to move operations on the *effects document* [1]. This is normally only a hypothetical document in which deletes are never performed. We require convergence of the effects document at all sites.

So for the time being we assume characters are never extracted by a move operation. Therefore move operations are really just insert operations! As shown in [3] the following IT and ET functions for insertion operations satisfy TP1 and TP2.

```
IT(Operation& O1, const Operation& O2)
{
```

```
    if (O2.p < O1.p || O2.p == O1.p && O2.id < O1.id) ++O1.p;
}

ET(Operation& O1, const Operation& O2)
{
    if (O2.p < O1.p) --O1.p;
}
```

Evidently we will have no problem achieving convergence on the effects document. Note however that we will ultimately need to have a concept of disabling move operations under IT (when there is a conflict because they try to move the same character). In order to ensure convergence of the effects document we will assume that "disabled" move operations are not really disabled. When a move operation is executed, it always inserts one character at the destination position in the effects document, whether the operation is "disabled" or not.

Consider that for each character in the effects document we have a flag for whether the character is present in the real document (ie where deletes are really applied). A disabled move operation inserts a character into the effects document with its present flag initialised to false.

In this section we don't actually store or manipulate the real document, hence there is no p-position. We are only interested in the q-position with respect to the effects document.

Source position tracking
When two enabled move operations move the same character, each tracks where it was inserted by the other operation, and one is "disabled".

When two disabled operations move the same character, tracking is not applied. Each operation simply tracks its original source character.

As far as the effects document is concerned a move operation is regarded as a special type of tracking operation (for its source), followed by an insertion operation into its destination. It never performs a delete on the effects document. There is no self-interference to worry about because the source tracker is not mutative.

A "disabled" operation only affects the tracking rule for when two operations try to "move" the same character.

Disabling a move operation correctly
Disabling move operations correctly is rather subtle, and compares to the problem of correctly selecting a unique winner that somehow dominates all other sites as for assignment operations. According to [4], a simple enable flag won't work. Neither will a simple disable counter because of cycles in the dominance relation. Instead it is necessary to use a q-position to uniquely select a "winner".

For move operation O, we let O.e be the q-position used to represent its enable status. O.e is initialised to zero when the operation is first created, and zero means that the operation is enabled. It is transformed as a q-position in the cases where two operations try to move the same character.

Multi-document state
The state allows for multiple effects documents. In state S, let S_i be the i^{th} effects document.

Let char(S,i,q) be the q^{th} character in S_i, and present(S,i,q) be a boolean to indicate whether the character at position q is present in the real document associated with S_i.

Let Insert(S,i,q,c,f) insert character c with presence flag f into S_i at position q.

Move operation
Let each operation contain the following fields

Field	Description
id	The site identifier of the site that originally generated the operation.
e	q-position used to ensure there is a unique winner when operations try to move the same character. An operation is enabled if e = 0, otherwise it is disabled
si	Identifies the source effects document
sq	Zero based source position in the source effects document
di	Identifies the destination effects document
dq	Zero based insertion position in the destination effects document
c	Value of character to be moved

The following function executes operation O on state S.

```
void ApplyOp(const Op& O, State& S)
{
    assert( O.c == char(S, O.si, O.sq) );
    assert( present(S, O.si, O.sq) );

    if (O.e == 0)    // is O enabled?
    {
        present(S, O.si, O.sq) = false;    // mark source character as no longer present
        Insert(S, O.di, O.dq, O.c, true);
    }
    else
    {
        Insert(S, O.di, O.dq, O.c, false);
    }
}
```

Note the precondition that the source character to be moved is marked as present. This precondition must be maintained under IT and ET.

When the operation is enabled, the source character is marked as no longer present, and the character is inserted into the destination document and marked as present.

When the operation is disabled, the source character remains marked as present, and the character is inserted into the destination document but marked as not present.

ApplyOp() maintains the following invariant: although a character can appear in many places simultaneously in the effects documents, exactly one will have the present status marked. This represents its "real" location.

It is easy to see that ApplyOp() ensures matter is conserved. More formally the total number of characters that are marked as present is invariant.

Inclusion Transform
The following function lets O1 := IT(O1,O2)

```
void IT(Op& O1, const Op& O2)
{
    // Are O1, O2 trying to move the same character?
    bool moveSameChar = (O1.si == O2.si && O1.sq == O2.sq);
```

```
    if (moveSameChar)
    {
        // Update enable status.  This is like a q-position
        if (O2.e < O1.e || O2.e == O1.e && O2.id < O1.id)
        {
            ++O1.e;
        }
    }

    // Adjust O1 source position according to affect of O2
    if (moveSameChar && O2.e == 0)
    {
        // O1 must track where character was moved by O2
        O1.si = O2.di;
        O1.sq = O2.dq;
    }
    else
    {
        if (O2.di == O1.si && O2.dq <= O1.sq) ++O1.sq;
    }

    // Adjust O1 destination position according to affect of O2
    if (O1.di == O2.di)
    {
        if (O2.dq < O1.dq || O2.dq == O1.dq && O2.id < O1.id) ++O1.dq;
    }
}
```

Convergence of the characters in the effects document

From the definition of `ApplyOp()` we see that the source character is never removed from the effects document, and a single character is inserted at the destination position irrespective of whether the move operation is enabled or not. From the definition of IT(O1,O2), we see that the destination position O1.dq is adjusted according to the IT algorithm in given in [3], irrespective of the enable status of O1 or O2. Therefore, as shown by [3] the provided definition of IT will achieved convergence of the effects document at quiescence at all sites.

Source positions track correctly

Recall the invariant maintained by ApplyOp() : although a character can appear in many places simultaneously in the effects documents, exactly one will have the present status marked. This is called the real location of the character in a given document state.

Claim : Source positions track correctly. ie an invariant of IT is that the source position of an operation tracks the real location of the character in the input document state of the operation.

From the definition of IT(O1,O2), we see that O1 tracks its source position to the destination of O2 if and only if O2 is both an enabled and conflicting move operation. Otherwise it is assumed that O2 merely inserts a character at its destination position (irrespective of whether it is enabled or not), and the source position of O1 is adjusted unambiguously on that basis.

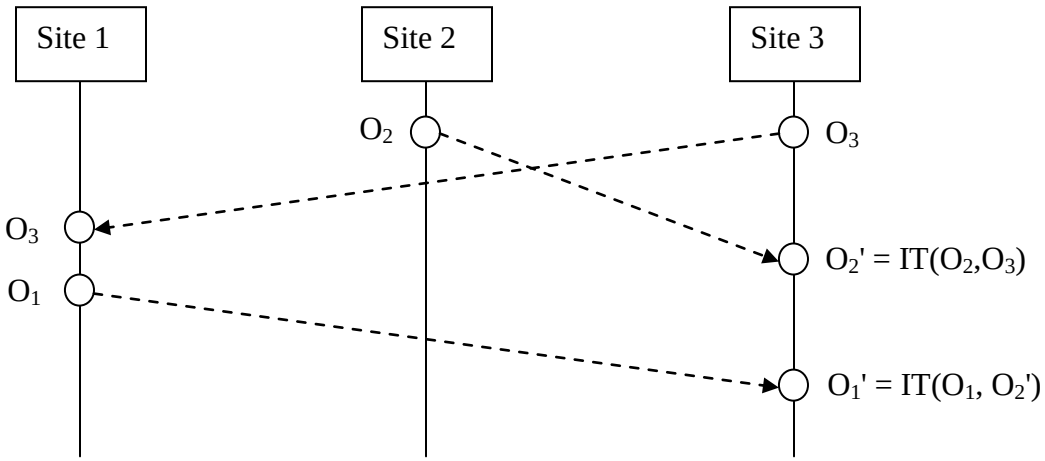
At quiescence all sites agree on which conflicting move operation takes priority

From the definition of IT(O1,O2), we see that the boolean `moveSameChar` is set if and only if the two concurrent move operations are trying to move the same character, irrespective of their enable status.

When the operations conflict the enable q-position e is updated in the manner used by [4] to select a unique and agreed "winner" at all sites.

Correctness depends on the fact that source positions track the one and only real location of the character. As a result, operations that are defined on the same document state and move the same character will always be seen to have exactly the same source positions.

For example consider the following scenario.



Let O1,O2,O3 all move the same character. O1 is generated in the context of O3 and therefore will have initialised its source position to the destination position of O3.

At site 3, O2 is IT'd past O3 and therefore its source position will be tracked to the destination position of O3. Therefore when site 3 receives O1 and ITs it past O2' it will be found that the source positions are equal, as required.

This is actually the example taken from [4] used to demonstrate cycles in the dominance graph if a disable count is used.

Exclusion Transform

The following function lets O1 := ET(O1,O2). A precondition is that [O2 O1] is contextually serialised.

```
void ET(Op& O1, const Op& O2)
{
    if (O1.di == O2.di && O2.dq < O1.dq) --O1.dq;

    // Is O1 moving the character inserted by O2?
    bool moveSameChar = (O2.di == O1.si && O2.dq == O1.sq);

    // Adjust O1 source position according to affect of O2
    if (moveSameChar)
    {
        assert(O2.e == 0);
        O1.si = O2.si;
        O1.sq = O2.sq;
    }
    else
    {
        if (O2.di == O1.si && O2.dq < O1.sq) --O1.sq;
        if (O1.si == O2.si && O1.sq == O2.sq) moveSameChar = true;
    }

    if (moveSameChar)
    {
        if (O2.e < O1.e)
        {
            --O1.e;
        }
    }
}
```

ET is simply the inverse of IT. The steps taken by IT must be undone in reverse order.

Extension to real documents
We now extend the previous approach to allow operations to work directly on the real document. The source and destination q-position are still required. We introduce the p-positions that relate to the corresponding positions in the real document.

Multi-document state
The state allows for multiple documents. In state S, let S_i be the i^{th} document.

We assume the effects document is not explicitly stored. However, it is necessary to store a mapping between p-position and q-position, in order to be able to generate operations correctly. A run-length encoded implementation is suitable.

Let char(S,i,p) be the p^{th} character in S_i

Let Remove(S,i,p,q) remove the character at the given p-position and q-position from Si. This uses the p-position to remove the character from the document. The q-position allows the p-q map to be updated correctly.

Let Insert(S,i,p,q,c,f) insert character c with present status f at the given p-position and q-position in Si. A character is only inserted into the real document at position p if f=true. The q-position allows the p-q map to be updated correctly.

Move operation
Let each operation contain the following fields

Field	Description
id	The site identifier of the site that originally generated the operation.
e	q-position used to ensure there is a unique winner when operations try to move the same character. An operation is enabled if e = 0, otherwise it is disabled
si	Identifies the source effects document
sp	Zero based source position in the real source document
sq	Zero based source position in the source effects document
di	Identifies the destination effects document
dp	Zero based insertion position in the real destination document
dq	Zero based insertion position in the destination effects document
c	Value of character to be moved

The following function executes operation O on state S.

```
void ApplyOp(const Op& O, State& S)
{
    if (O.e == 0)    // is O enabled?
    {
        assert(char(S, O.si, O.sp) == O.c );
        Remove(S, O.si, O.sp, O.sq);
        Insert(S, O.di, O.dp, O.dq, O.c, true);
    }
    else
    {
        Insert(S, O.di, O.dp, O.dq, O.c, false);
    }
}
```

Inclusion Transform

```
void IT(Op& O1, const Op& O2)
{
    if (O1.si == O2.si && O2.sq < O1.sq && O2.e == 0) --O1.sp;

    // Are O1, O2 trying to move the same character?
    bool moveSameChar = (O1.si == O2.si && O1.sq == O2.sq);

    if (moveSameChar)
    {
        if (O2.e < O1.e || O2.e == O1.e && O2.id < O1.id)
        {
            ++O1.e;
        }
    }

    // Adjust O1 source position according to affect of O2
    if (moveSameChar && O2.e == 0)
    {
        // O1 must track where character was moved by O2
        O1.si = O2.di;
        O1.sq = O2.dq;
        O1.sp = O2.dp;
    }
    else
    {
        if (O2.di == O1.si && O2.dq <= O1.sq)
        {
            ++O1.sq;
            if (O2.e == 0) ++O1.sp;  // Note that moveSameChar must be false
        }
    }

    if (!moveSameChar && O2.e == 0 && O1.di == O2.si && O2.sq < O1.dq) --O1.dp;

    // Adjust O1 destination position according to affect of O2
    if (O1.di == O2.di)
    {
        if (O2.dq < O1.dq || O2.dq == O1.dq && O2.id < O1.id)
        {
            ++O1.dq;
            if (!moveSameChar && O2.e == 0) ++O1.dp;
        }
    }
}
```

Exclusion Transform

```
void ET(Op& O1, const Op& O2)
{
    bool decd = false;
    if (O1.di == O2.di && O2.dq < O1.dq)
    {
        decd = true;
        --O1.dq;
    }

    // Is O1 moving the character inserted by O2?
    bool moveSameChar = (O2.di == O1.si && O2.dq == O1.sq);

    // Adjust O1 source position according to affect of O2
    if (moveSameChar)
    {
        assert(O2.e == 0);
        O1.si = O2.si;
        O1.sq = O2.sq;
        O1.sp = O2.sp;
    }
    else
    {
        if (O2.e == 0 && O1.di == O2.si && O2.sq < O1.dq) ++O1.dp;
        if (O2.di == O1.si && O2.dq < O1.sq)
        {
            if (O2.e == 0) --O1.sp;
```

```
            --O1.sq;
        }
        if (O1.si == O2.si && O1.sq == O2.sq) moveSameChar = true;
    }

    if (moveSameChar)
    {
        if (O2.e < O1.e)
        {
            --O1.e;
        }
    }
    else
    {
        if (O2.e == 0)
        {
            if (decd) --O1.dp;
            if (O1.si == O2.si && O2.sq < O1.sq) ++O1.sp;
        }
    }
}
```

Is the extension required?
Consider that operations don't store p-positions at all. This simplifies the IT and ET functions - a good idea before embarking on versions that work on ranges of characters at a time.

The original implementation of ApplyOp() is as follows

```
void ApplyOp(const Op& O, State& S)
{
    assert( O.c == char(S, O.si, O.sq) );
    assert( present(S, O.si, O.sq) );

    if (O.e == 0)    // is O enabled?
    {
        present(S, O.si, O.sq) = false;    // mark source character as no longer present
        Insert(S, O.di, O.dq, O.c, true);
    }
    else
    {
        Insert(S, O.di, O.dq, O.c, false);
    }
}
```

This should be able to unambiguously map the q-position to the corresponding p-position using the p-q map.

Advantages

- Operations are more space efficient
- IT and ET are simpler and faster

References

[1]	Du Li and Rui Li. Ensuring consistency in real-time group editors. *ACM Transactions on Computer-Human Interaction,* April 2004. Under review.
[2]	Du Li and Rui Li. An Operational Transformation Algorithm and Performance Evaluation. *Journal of CSCW,* July 2005. Under review.
[3]	David Barrett-Lennard. Operational transform - Single character insert and delete operations.
[4]	David Barrett-Lennard. Operational transform - Assignment operations.