Operational transform
Multi-character insertion and deletion operations

29 August 2005
David Barrett-Lennard

Abstract
The algorithm presented in [3] is simple and correct, but not practical for certain applications, such as configuration management. The paper provides a more efficient algorithm for the case of multi-character insertion and deletion operations.

Introduction

In this paper an operation efficiently stores a set of characters to be deleted as well as a set of characters to be inserted. It follows the representation in [4], ie the characters to be deleted are represented by an ordered set of intervals. Similarly the characters to be inserted are represented by an ordered set of insertions strings. This means a single operation can efficiently represent arbitrary changes to the document state, allowing it to be closed under merging. Efficient merging algorithms have been presented in [4].

Note that the extractions are in pre-extraction coordinates, while the insertions are in post-insertion coordinates.

As in [4], left to right scans of the intervals are used to provide algorithms that are linear, not quadratic in the number of deletion and insertion strings.

Review of algorithm for single character operations

The following table from [3] defines the IT,ET algorithms for single character insert/delete operations:

O_1	O_2	$IT(O_1,O_2)$
ins	ins	`if (O2.q < O1.q || O2.q == O1.q && O2.id < O1.id) { ++O1.q; ++O1.p; }`
del	ins	`if (O2.q <= O1.q) { ++O1.q; ++O1.p; }`
ins	del	`if (O2.enabled && O2.q < O1.q) --O1.p;`
del	del	`if (O2.enabled && O2.q < O1.q) --O1.p;` `if (O2.enabled && O2.q == O1.q) O1.enabled = false;`

O_1	O_2	$ET(O_1,O_2)$
ins	ins	`if (O2.q < O1.q) { --O1.q; --O1.p; }`
del	ins	`if (O2.q < O1.q) { --O1.q; --O1.p; }`
ins	del	`if (O2.enabled && O2.q < O1.q) ++O1.p;`
del	del	`if (O2.enabled && O2.q < O1.q) ++O1.p;` `if (O2.enabled && O2.q == O1.q) O1.enabled = true;`

The aim of this paper is to provide *equivalent* algorithms for multi-character delete/insert operations.

Multi-character insertion operations

Firstly we limit ourselves to insertion operations that apply to a single text document. There is no need for a separate q-position.

Dual IT

Insertion positions in O_1 and O_2 need to be shifted to the right. We accumulate the number of inserted characters in O_1 and in O_2. These are the required shifts to be applied. At each step of the algorithm we have an insert in O_1 and an insert from O_2. We simply compare their positions. If they are equal then we use site ids to break the tie.

```
void DualIT(MultiCharInsertOp& O1, MultiCharInsertOp& O2)
{
    int s1 = 0;     // Accumulated characters inserted by O1
    int s2 = 0;     // Accumulated characters inserted by O2

    iterator i1 = O1.begin();
    iterator i2 = O2.begin();

    while(i1 != O1.end() && i2 != O2.end())
    {
        int d = (s1 + i2->p) - (s2 + i1->p);
        if (d < 0 || d == 0 && O2.id < O1.id)
        {
            // Process i2
            i2->p += s1;
            s2 += i2->size();
            ++i2;
        }
        else
        {
            // Process i1
            i1->p += s2;
            s1 += i1->size();
            ++i1;
        }
    }
    if (s1)
    {
        while (i2 != O2.end())
        {
            i2->p += s1;
            ++i2;
        }
    }
    if (s2)
    {
        while (i1 != O1.end())
        {
            i1->p += s2;
            ++i1;
        }
    }
}
```

Adjacent swap

Insertion coords are in post insertion coordinates. Coords in O_2 need to be shifted left to exclude the effect of O_1. Coords in O_1 need to be shifted right to include the effect of O_2.

The following shows an initial document state, and strings inserted first by O_1 then by O_2.

```
          O1                   O2
  abcde   -->    ab111cde      -->  a222b111c2222de
                   [ )               [ )     [  )
              (coords for O1)         (coords for O2)

          O2'                  O1'
  abcde   -->    a222bc2222de  -->  a222b111c2222de
                   [ )  [  )             [ )
              (coords for O2')        (coords for O1')
```

The following algorithm in C++ shows how a left to right scan though the insertion strings in O_1,O_2 can be used to transpose O_1, O_2 in time linear in the number of insertion strings in O_1 and O_2.

```
void AdjSwap(MultiCharInsertOp& O1, MultiCharInsertOp& O2)
{
    int s1 = 0;     // Accumulated characters inserted by O1
    int s2 = 0;     // Accumulated characters inserted by O2

    iterator i1 = O1.begin();
    iterator i2 = O2.begin();

    while(i1 != O1.end() && i2 != O2.end())
```

```
    {
        if (s2 + i1->p < i2->p)
        {
            // Process i1
            i1->p += s2;
            s1 += i1->size();
            ++i1;
        }
        else
        {
            // Process i2
            i2->p -= s1;
            s2 += i2->size();
            ++i2;
        }
    }

    if (s1)
    {
        while (i2 != O2.end())
        {
            i2->p -= s1;
            ++i2;
        }
    }
    if (s2)
    {
        while (i1 != O1.end())
        {
            i1->p += s2;
            ++i1;
        }
    }
}
```

Multi-character insertion and deletion operations
Each operation contain the following fields

Field	Description
id	The site identifier of the site that originally generated the operation.
X	An ordered set of extraction intervals
I	An ordered set of insertion strings

An extraction interval stores the following fields

Field	Description
p	p-position of string to be extracted
q	q-position of string to be extracted
s	string to be extracted
enabled	flag to indicate whether the extraction is enabled

An insertion string stores the following fields

Field	Description
p	p-position of string to be inserted
q	q-position of string to be inserted
s	string to be inseted

Splitting an extraction interval
It is sometime necessary to split an extraction interval. The C++ algorithm presented below ensures equivalence of the operation when an extraction interval is split. This requires attention to the fields within an interval - the string, p-position, q-position and enabled status.

```
// Insert an element into the list just after position i, which must not be at the end of the list
// Returns iterator to inserted element.
template <class T>
inline std::list<T>::iterator InsertAfter(std::list<T>& L, std::list<T>::iterator i)
{
    cxAssert(i != L.end());
    return L.insert(++i);    // Insert just before (i+1), which is immediately after i
}

/*
Split the interval in list X at iterator position i, so that the left interval is of size n1

        [iiiiiiiiiiiiiiiii)

 --->   [iiiiiiii)[jjjjjjjj)

        <-- n1 --><-- n2 -->
*/

void SplitRange(MCID_XRanges& X, MCID_XRanges::iterator i, int n1)
{
    int n2 = i->size() - n1;
    MCID_XRanges::iterator j = InsertAfter<MCID_XRange>(X,i);
    j->p = i->p;
    if (i->enabled) j->p += n1;     // Note: don't offset p-position when disabled
    j->q = i->q + n1;
    j->s = std::string(i->s.begin() + n1, i->s.end());
    j->enabled = i->enabled;
    i->s.erase(i->s.begin() + n1, i->s.end());
}
```

DualIT_dd

This algorithm handles the dual IT of deletions with deletions. The following part of the algorithm for IT is relevant.

O_1	O_2	$IT(O_1,O_2)$
del	del	`if (O2.enabled && O2.q < O1.q) --O1.p;` `if (O2.enabled && O2.q == O1.q) O1.enabled = false;`

As we scan left to right, we treat the deletions as intervals (not merely positions). There are a number of cases for processing the next two intervals. Eg they can be disjoint, partially overlap, coincide or one interval can contain the other.

Where intervals are both enabled and overlap then the overlapping part needs to be disabled. This may require splitting of the intervals.

Notes

- When comparing q-positions we never apply shifts, because deletions have no effect on the effects document.
- The accumulated shift is applied to the p-position. The accumulated shift should only account for the enabled deletions.
- Intervals are shifted whether they are enabled or not

```
void DualIT_dd(MultiCharInsertOrDeleteOp& O1, MultiCharInsertOrDeleteOp& O2)
{
    MCID_XRanges& X1 = O1.X;
    MCID_XRanges& X2 = O2.X;

    int s1 = 0;     // Accumulated shift from enabled intervals of X1
    int s2 = 0;     // Accumulated shift from enabled intervals of X2

    MCID_XRanges::iterator x1 = X1.begin();
    MCID_XRanges::iterator x2 = X2.begin();

    while(x1 != X1.end() && x2 != X2.end())
    {
```

```
    if (x1->q + x1->size() <= x2->q)
    {
        // Next is [11111)
        x1->p -= s2;
        if (x1->enabled) s1 += x1->size();
        ++x1;
    }
    else if (x2->q + x2->size() <= x1->q)
    {
        // Next is [22222)
        x2->p -= s1;
        if (x2->enabled) s2 += x2->size();
        ++x2;
    }
    else
    {
        // Intervals overlap
        if (x1->q < x2->q)
        {
            // [1111111111111
            //       [2222222
            SplitRange(X1, x1, x2->q - x1->q);
            x1->p -= s2;
            if (x1->enabled) s1 += x1->size();
            ++x1;
        }
        else if (x2->q < x1->q)
        {
            // [2222222222222
            //       [1111111
            SplitRange(X2, x2, x1->q - x2->q);
            x2->p -= s1;
            if (x2->enabled) s2 += x2->size();
            ++x2;
        }

        cxAssert(x2->q == x1->q);
        if (x2->size() < x1->size())
        {
            // [222222)
            // [111111111111)
            SplitRange(X1, x1, x2->size());
        }
        else if (x1->size() < x2->size())
        {
            // [111111)
            // [222222222222)
            SplitRange(X2, x2, x1->size());
        }

        // [111111)
        // [222222)
        cxAssert(x1->size() == x2->size());
        x2->p -= s1;
        x1->p -= s2;
        bool x1enabled = x1->enabled;
        if (x2->enabled) { s2 += x2->size(); x1->enabled = false; }
        if (x1enabled) { s1 += x1->size(); x2->enabled = false; }
        ++x1;
        ++x2;
    }
}

// Apply shifts to remaining intervals in X (if any)
if (s1)
{
    while (x2 != X2.end())
    {
        x2->p -= s1;
        ++x2;
    }
}

// Apply shifts to remaining intervals in I (if any)
if (s2)
{
    while (x1 != X1.end())
    {
        x1->p -= s2;
```

```
            ++x1;
        }
    }

    CoalesceRanges(X1);
    CoalesceRanges(X2);
}
```

DualIT_id

This algorithm handles the dual IT of insertions with deletions. The following part of the algorithm for IT of single character operations is relevant:

O_1	O_2	$IT(O_1,O_2)$
del	ins	`if (O2.q <= O1.q) { ++O1.q; ++O1.p; }`
ins	del	`if (O2.enabled && O2.q < O1.q) --O1.p;`

In the following example, let O_1 insert '1' characters, and O_2 delete '2' characters. An asterisk represents the placeholder for a "deleted" character in the effects document.

```
                      O2                                    O1'
a222bc22222d22ef   --->   a***bc*****d**ef            --->    a111*11**b11c*****d**ef11
 [ )  [   ) []                                                  [ ) []   []          []

                      O1                                    O2'
a222bc22222d22ef   --->   a11121122b11c22222d22ef11    --->    a111*11**b11c*****d**ef11
                            [ ) []   []          []
                              []  []    [   ) []
```

Notes

- We scan left to right through O_1 and O_2. At each step we consider the next interval from O_2 and the next insertion position from O_1
- We make sense of what is happening in the document state obtained after performing O_1. In this state all characters are present.
- Coords mentioned in O_1 are already relative to this state. Coords in O_2 need to be shifted right by the number of characters inserted on the left by O_1.
- Extraction intervals may need to be split - ie when there are insertions within the interval

```
void DualIT_id(MultiCharInsertOrDeleteOp& O1, MultiCharInsertOrDeleteOp& O2)
{
    MCID_IRanges& I = O1.I;
    MCID_XRanges& X = O2.X;

    int si = 0;     // Accumulated shift from I
    int sx = 0;     // Accumulated shift from X

    MCID_XRanges::iterator x = X.begin();
    MCID_IRanges::iterator i = I.begin();

    while(i != I.end() && x != X.end())
    {
        if (i->q <= si + x->q)
        {
            //    [xxxxxxx)
            //   i
            // Process next insertion
            si += i->size();
            i->p -= sx;
            ++i;
        }
        else
        {
            int d = i->q - (si + x->q);
            if (d < x->size())
            {
                // [xxxxxx)
```

```
                //     i
                SplitRange(X,x, i->q - (si + x->q));
            }

            // [xxxxxx)
            //          i
            // Process next extraction
            x->q += si;
            x->p += si;
            if (x->enabled) sx += x->size();
            ++x;
        }
    }

    if (si)
    {
        while (x != X.end())
        {
            x->q += si;
            x->p += si;
            ++x;
        }
    }

    if (sx)
    {
        while (i != I.end())
        {
            i->p -= sx;
            ++i;
        }
    }
}
```

DualIT_ii

This algorithm handles the dual IT of insertions with insertions. The following part of the algorithm for IT of single character operations is relevant:

O_1	O_2	$IT(O_1,O_2)$
ins	ins	`if (O2.q < O1.q || O2.q == O1.q && O2.id < O1.id) { ++O1.q; ++O1.p; }`

Insertion positions in O_1 and O_2 need to be shifted to the right. We accumulate the number of inserted characters in O_1 and in O_2. These are the required shifts to be applied.

At each step of the algorithm we have an insert in O_1 and an insert from O_2. We simply compare their (shifted) positions. If they are equal then we use site ids to break the tie.

```
void DualIT_ii(MultiCharInsertOrDeleteOp& O1, MultiCharInsertOrDeleteOp& O2)
{
    MCID_IRanges& I1 = O1.I;
    MCID_IRanges& I2 = O2.I;

    int s1 = 0;     // Accumulated shift from I1
    int s2 = 0;     // Accumulated shift from I2

    MCID_IRanges::iterator i1 = I1.begin();
    MCID_IRanges::iterator i2 = I2.begin();

    while(i1 != I1.end() && i2 != I2.end())
    {
        int d = (s1 + i2->q) - (s2 + i1->q);
        if (d < 0 || d == 0 && O2.id < O1.id)
        {
            // Process i2
            i2->p += s1;
            i2->q += s1;
            s2 += i2->size();
            ++i2;
        }
        else
```

```
        {
            // Process i1
            i1->p += s2;
            i1->q += s2;
            s1 += i1->size();
            ++i1;
        }
    }

    if (s1)
    {
        while (i2 != I2.end())
        {
            i2->p += s1;
            i2->q += s1;
            ++i2;
        }
    }

    if (s2)
    {
        while (i1 != I1.end())
        {
            i1->p += s2;
            i1->q += s2;
            ++i1;
        }
    }
}
```

DualIT

O_1 and O_2 are composite operations of extractions followed by insertions. Using the general algorithm for dual IT of a list with a list leads to the following algorithm for dual IT:

```
void DualIT(MultiCharInsertOrDeleteOp& O1, MultiCharInsertOrDeleteOp& O2)
{
    DualIT_dd(O1,O2);       // Deletions against deletions
    DualIT_id(O1,O2);       // Insertions against deletions
    DualIT_id(O2,O1);       // Deletions against insertions
    DualIT_ii(O1,O2);       // Insertions against insertions
}
```

AdjSwap_dd

The adjacent swap algorithm transforms $[O_1\ O_2]$ to $[O_2'\ O_1']$.

This algorithm handles the transpose of deletions with deletions. The following part of the algorithm for IT/ET of single character operations is relevant:

O_1 = del, O_2 = del

$IT(O_1,O_2)$	`if (O2.enabled && O2.q < O1.q) --O1.p;` `if (O2.enabled && O2.q == O1.q) O1.enabled = false;`
$ET(O_2,O_1)$	`if (O1.enabled && O1.q < O2.q) ++O2.p;` `if (O1.enabled && O1.q == O2.q) O2.enabled = true;`

In the following example, let O_1 delete '1' characters, and O_2 delete '2' characters. An asterisk represents the placeholder for a "deleted" character in the effects document.

```
                      O1                        O2
    ab11112221111cd   --->   ab****222****cd    --->    ab***********cd

                      O2'                       O1'
    ab11112221111cd   --->   ab1111***1111cd    --->    ab***********cd
```

O_2 may delete characters that have already been deleted by O_1. In that case O_2 should be disabled.

For the purpose of comparing q-positions there is no need to shift positions because these operations have no impact on the effects document.

As we scan left to right we compare the next interval from O_1 against the next interval from O_2. There are a number of cases to consider.

```
void AdjSwap_dd(MultiCharInsertOrDeleteOp& O1, MultiCharInsertOrDeleteOp& O2)
{
    MCID_XRanges& X1 = O1.X;
    MCID_XRanges& X2 = O2.X;

    int s1 = 0;     // Accumulated shift from enabled intervals of X1
    int s2 = 0;     // Accumulated shift from enabled intervals of X2

    MCID_XRanges::iterator x1 = X1.begin();
    MCID_XRanges::iterator x2 = X2.begin();

    while(x1 != X1.end() && x2 != X2.end())
    {
        if (x1->q + x1->size() <= x2->q)
        {
            // Next is [11111)
            x1->p -= s2;
            if (x1->enabled) s1 += x1->size();
            ++x1;
        }
        else if (x2->q + x2->size() <= x1->q)
        {
            // Next is [22222)
            x2->p += s1;
            if (x2->enabled) s2 += x2->size();
            ++x2;
        }
        else
        {
            // Intervals overlap
            if (x1->q < x2->q)
            {
                // [1111111111111
                //       [2222222
                SplitRange(X1, x1, x2->q - x1->q);
                x1->p -= s2;
                if (x1->enabled) s1 += x1->size();
                ++x1;
            }
            else if (x2->q < x1->q)
            {
                // [2222222222222
                //       [1111111
                SplitRange(X2, x2, x1->q - x2->q);
                x2->p += s1;
                if (x2->enabled) s2 += x2->size();
                ++x2;
            }

            cxAssert(x2->q == x1->q);
            if (x2->size() < x1->size())
            {
                // [222222)
                // [111111111111)
                SplitRange(X1, x1, x2->size());
            }
            else if (x1->size() < x2->size())
            {
                // [111111)
                // [222222222222)
                SplitRange(X2, x2, x1->size());
            }

            // [111111)
            // [222222)
            cxAssert(x1->size() == x2->size());
            x1->p -= s2;
            x2->p += s1;
```

```
            if (x1->enabled) { s1 += x1->size(); x2->enabled = true; }
            if (x2->enabled) { s2 += x2->size(); x1->enabled = false; }
            ++x1;
            ++x2;
        }
    }

    // Apply shifts to remaining intervals in X (if any)
    if (s1)
    {
        while (x2 != X2.end())
        {
            x2->p += s1;
            ++x2;
        }
    }

    // Apply shifts to remaining intervals in I (if any)
    if (s2)
    {
        while (x1 != X1.end())
        {
            x1->p -= s2;
            ++x1;
        }
    }

    CoalesceRanges(X1);
    CoalesceRanges(X2);
}
```

AdjSwap_ii

The adjacent swap algorithm transforms $[O_1\ O_2]$ to $[O_2'\ O_1']$.

This algorithm handles the transpose of insertions with insertions. The following part of the algorithm for IT/ET of single character operations is relevant:

O_1 = ins, O_2 = ins

$IT(O_1,O_2)$	`if (O2.q < O1.q || O2.q == O1.q && O2.id < O1.id) { ++O1.q; ++O1.p; }`
$ET(O_2,O_1)$	`if (O1.q < O2.q) { --O2.q; --O2.p; }`

In the following example, let O_1 insert '1' characters, and O_2 insert '2' characters.

```
          O1                          O2
abcdef  -->   a111bc1111d11111ef    -->   a222111b2222c1111d111112222e2222f
               [ )  [  ) [   )             [ )    [  )           [  ) [  )
```

We compare q-positions in the state obtained after execution of O_2. q-positions of O_2 don't require adjustment. However q-positions of O_2 need to the shifted to the right.

```
void AdjSwap_ii(MultiCharInsertOrDeleteOp& O1, MultiCharInsertOrDeleteOp& O2)
{
    MCID_IRanges& I1 = O1.I;
    MCID_IRanges& I2 = O2.I;

    int s1 = 0;     // Accumulated shift from I1
    int s2 = 0;     // Accumulated shift from I2

    MCID_IRanges::iterator i1 = I1.begin();
    MCID_IRanges::iterator i2 = I2.begin();

    while(i1 != I1.end() && i2 != I2.end())
    {
        if (i2->q <= s2 + i1->q)
        {
            // Process i2
            i2->p -= s1;
```

```
            i2->q -= s1;
            s2 += i2->size();
            ++i2;
        }
        else
        {
            // Process i1
            i1->p += s2;
            i1->q += s2;
            s1 += i1->size();
            ++i1;
        }
    }

    if (s1)
    {
        while (i2 != I2.end())
        {
            i2->p -= s1;
            i2->q -= s1;
            ++i2;
        }
    }

    if (s2)
    {
        while (i1 != I1.end())
        {
            i1->p += s2;
            i1->q += s2;
            ++i1;
        }
    }
}
```

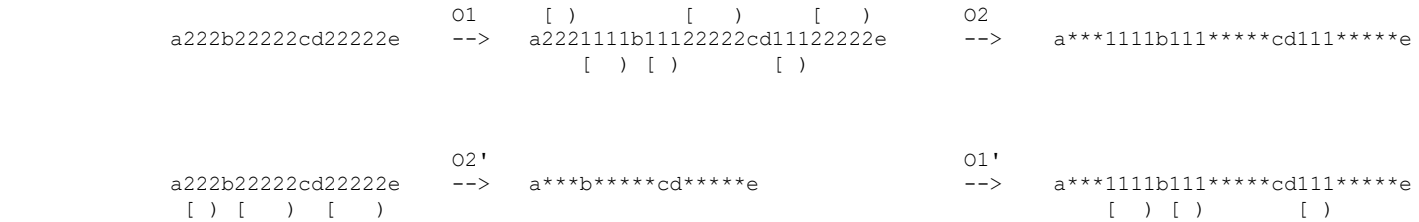
AdjSwap_id
The adjacent swap algorithm transforms [O_1 O_2] to [O_2' O_1'].

This algorithm handles the transpose of insertions with deletions. The following part of the algorithm for IT/ET of single character operations is relevant:

O_1 = ins, O_2 = del

IT(O_1,O_2)	`if (O2.enabled && O2.q < O1.q) --O1.p;`
ET(O_2,O_1)	`if (O1.q < O2.q) { --O2.q; --O2.p; }`

In the following example, let O_1 insert '1' characters, and O_2 delete '2' characters. An asterisk represents the placeholder for a "deleted" character in the effects document.



Note that q-positions are compared in the state just after executing O_1. In this state q-positions can be directly compared without applying shifts.

It is assume that O_1 || O_2, so therefore intervals never overlap - which would mean that a character inserted by O_1 was deleted by O_2.

Intervals in both O_1 and O_2 need to be shifted left.

```
void AdjSwap_id(MultiCharInsertOrDeleteOp& O1, MultiCharInsertOrDeleteOp& O2)
{
    MCID_IRanges& I = O1.I;
    MCID_XRanges& X = O2.X;

    int si = 0;     // Accumulated shift from I
    int sx = 0;     // Accumulated shift from X

    MCID_XRanges::iterator x = X.begin();
    MCID_IRanges::iterator i = I.begin();

    while(i != I.end() && x != X.end())
    {
        if (i->q + i->size() <= x->q)
        {
            // Next is [11111)
            i->p -= sx;
            si += i->size();
            ++i;
        }
        else if (x->q + x->size() <= i->q)
        {
            // Next is [22222)
            x->p -= si;
            x->q -= si;
            if (x->enabled) sx += x->size();
            ++x;
        }
        else
        {
            // Intervals shouldn't overlap assuming O1 || O2
            < error >
        }
    }

    if (si)
    {
        while (x != X.end())
        {
            x->p -= si;
            x->q -= si;
            ++x;
        }
    }

    if (sx)
    {
        while (i != I.end())
        {
            i->p -= sx;
            ++i;
        }
    }
}
```

AdjSwap_di
The adjacent swap algorithm transforms $[O_1\ O_2]$ to $[O_2'\ O_1']$.

This algorithm handles the transpose of deletions with insertions. The following part of the algorithm for IT/ET of single character operations is relevant:

O1 = del, O2 = ins

$IT(O_1,O_2)$	`if (O2.q <= O1.q) { ++O1.q; ++O1.p; }`
$ET(O_2,O_1)$	`if (O1.enabled && O1.q < O2.q) ++O2.p;`

In the following example, let O1 delete '1' characters, and O2 insert '2' characters. An asterisk represents the placeholder for a "deleted" character in the effects document.

```
                    O1                                          O2
a111bc1111de1111  --->    a***bc****de****                --->    a*2222**b222c****d222e**222**
 [ )  [  )  [  )                                                    [  )   [ )      [ )   [ )

                    O2'                                         O1'
a111bc1111de1111  --->    a1222211b222c1111d222e1122211   --->    a*2222**b222c****d222e**222**
                            [  )   [ )      [ )   [ )             ()   ()     [  )     ()   ()
```

We make comparisons in the state corresponding to after the execution of O_2. In this state q-positions of O_1 are shifted right to include the effect of O_2. q-positions of O_2 do not need to be shifted for the purposes of comparison of q-position.

As we scan left to right we compare the next interval from O_1 (shifted right) against the next insertion position from O_2. If necessary the interval from O_1 may need to be split.

```
void AdjSwap_di(MultiCharInsertOrDeleteOp& O1, MultiCharInsertOrDeleteOp& O2)
{
    MCID_XRanges& X = O1.X;
    MCID_IRanges& I = O2.I;

    int si = 0;     // Accumulated shift from I
    int sx = 0;     // Accumulated shift from X

    MCID_XRanges::iterator x = X.begin();
    MCID_IRanges::iterator i = I.begin();

    while(i != I.end() && x != X.end())
    {
        if (i->q <= si + x->q)
        {
            //    [11111)
            //   2
            // Process next insertion
            si += i->size();
            i->p += sx;
            ++i;
        }
        else
        {
            int d = i->q - (si + x->q);
            if (d < x->size())
            {
                // [111111)
                //     2
                SplitRange(X,x, i->q - (si + x->q));
            }

            // [111111)
            //          2
            // Process next extraction
            x->q += si;
            x->p += si;
            if (x->enabled) sx += x->size();
            ++x;
        }
    }

    if (si)
    {
        while (x != X.end())
        {
            x->q += si;
            x->p += si;
            ++x;
        }
    }

    if (sx)
```

```
    {
        while (i != I.end())
        {
            i->p += sx;
            ++i;
        }
    }
}
```

AdjSwap

The adjacent swap algorithm transforms $[O_1\ O_2]$ to $[O_2'\ O_1']$.

O_1 and O_2 are composite operations of extractions followed by insertions. Using the general algorithm for transpose of a list with a list leads to the following algorithm for AdjSwap.

```
void AdjSwap(MultiCharInsertOrDeleteOp& O1, MultiCharInsertOrDeleteOp& O2)
{
    AdjSwap_id(O1,O2);
    AdjSwap_dd(O1,O2);
    AdjSwap_ii(O1,O2);
    AdjSwap_di(O1,O2);
}
```

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. July 2005.
[4]	David Barrett-Lennard. Log Compression Algorithm. July 2005.