Operational transform
Multi-character move operations

22 September 2006
David Barrett-Lennard

Abstract
The algorithm presented in [6] 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 move operations.

Introduction
In this paper an operation efficiently stores a set of characters to be extracted as well as a set of characters to be inserted, in a manner similar to [4] and [5]. 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.

Note that both the extractions and insertions are expressed in post-insertion coordinates. This allows the operations to extract the same characters that it has inserted. This would not be the case if extractions where expressed in pre-insertion coordinates.

As in [4] and [5], left to right scans of the intervals are used to provide algorithms that are linear, not quadratic in the number of extraction and insertion strings.

Review of algorithm for single character move operations
According to [5], we have the following functions for IT and ET

```
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;
    }
}

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;
        }
    }
}
```

Note that the extraction is expressed in pre-insertion coordinates.

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

Using post-insertion coordinates

The following DualIT algorithm assumes that the operations specify extractions and insertions in post-insertion coordinates.

```
void DualIT(SingleCharEffectsMoveOp2& O1, SingleCharEffectsMoveOp2& O2)
{
    if (O1.di == O2.di)
    {
        if (O2.dq < O1.dq || O2.dq == O1.dq && O2.GetId() < O1.GetId()) ++O1.dq; else ++O2.dq;
    }
    if (O1.di == O2.si && O1.dq <= O2.sq) ++O2.sq;
    if (O2.di == O1.si && O2.dq <= O1.sq) ++O1.sq;
    if (O1.si == O2.si && O1.sq == O2.sq)
    {
        if (O1.e == 0) { O2.si = O1.di; O2.sq = O1.dq; }
        if (O2.e == 0) { O1.si = O2.di; O1.sq = O2.dq; }
        if (O2.e < O1.e || O2.e == O1.e && O2.GetId() < O1.GetId()) ++O1.e; else ++O2.e;
    }
}
```

Notes

- Firstly the DualIT function transforms each operation's insertion position past the other operation's insertions. Therefore the insertion positions are shifted to the right as required.
- Now that the insertion position O1.dq accounts for the insertions performed by O2, it is possible to compare O1.dq and O2.sq to see whether O2.sq needs to be shifted right to include the effect of the insertion by O1. The comparison is valid because O1.dq and O2.sq both include the insertions by O2, noting that O2.sq is specified in post-insertion coordinates.
- Similarly O1.sq is transformed to include the effect of the insertion at O2.dq by O2.
- After shifting O1.sq, O2.sq as required according to the other operation's insertions, these source positions are now specified in the same document state – ie containing the union of all insertions by both O1 and O2. Therefore they can be directly compared to see whether they moved the same character.
- If O1, O2 have indeed moved the same character then it is necessary for each operation to track where the other operation really put the character. So if the other operation is enabled, the source position is assigned to the other operation's destination position. Note that it is

appropriate to use the transformed destination position which is specified relative to the document state containing the union of all insertions by both O1 and O2.

- Finally, if O1 and O2 have moved the same character it is necessary to transform the e position in the manner of a q-position. This has the effect of selecting a winner versus loser in the sense of the conflict over moving the same character.

The following Transpose function assumes that [O1, O2] are contextually serialized operations satisfying O1 || O2, and O1, O2 specify extractions and insertions in post-insertion coordinates. It is an in-place algorithm that transforms O1, O2 so that after calling the function [O2, O1] are contextually serialized.

```
void Transpose(SingleCharEffectsMoveOp2& O1, SingleCharEffectsMoveOp2& O2)
{
    int prev2dq = O2.dq;
    if (O2.di == O1.si && O2.dq <= O1.sq) ++O1.sq;
    if (O1.di == O2.di) { if (O1.dq < O2.dq) --O2.dq; else ++O1.dq; }
    if (O1.si == O2.si && O1.sq == O2.sq || O1.di == O2.si && O1.dq == O2.sq)
    {
        if (O1.e == 0) { O2.si = O1.si; O2.sq = O1.sq; }
        if (O1.e < O2.e) --O2.e; else ++O1.e;
        if (O2.e == 0) { O1.si = O2.di; O1.sq = prev2dq; }
    }
    if (O1.di == O2.si && O1.dq < O2.sq) --O2.sq;
}
```

Notes

- Firstly O2.dq is saved in a temporary variable, because it is possible that later in the function it will be necessary to set O1.sq to the *original* value of O2.dq.
- O1.sq is transformed (shifted right as required) to account for the insertion performed by O2. This must be done before transforming O2.dq (ie shifting left to exclude the insertion performed by O1). The reason is that O2.dq and O1.sq should be compared in a document state that includes the insertion performed by O1.
- Next O2.dq is shifted left to exclude the insertion by O1, and O1.dq is shifted right to include the insertion by O2.
- It is now possible to test whether O1 and O2 are moving the same character. There are two possibilities: If O1 is enabled then O2 will find that its source character matches the destination position of O1. Alternatively, of O1 is disabled then O2 will find that its source character matches the source character of O1. These tests are all done in the document state that includes both the insertions by O1 and O2. Therefore it must be done after transforming O1's source and destination position (to include the effect of the insertion by O2). Also it must be done before transforming O2.sq (which excludes the effect of the insertion by O1).
- If O1 and O2 are found to have moved the same character and O1 is enabled then it is necessary for O2's source position to be reassigned to O1's source position. Now O2 requires this position to include the effect of its own (ie O2's) insertion, but not O1's insertion. At this point in the algorithm it is already the case that O1.sq has been shifted right as required to include the effect of O2's insertion. However it will include the effect of O1's own insertion. As it turns out there is a subsequent final step in the algorithm that will shift O2.sq left to exclude the insertion by O1, exactly as we require.
- Continuing on in the case where O1 and O2 are moving the same character, we transform O1.e and O2.e in the manner of the transpose of q-positions. This may case O2.e to decrement to zero, causing it to be re-enabled.
- If O1 and O2 are moving the same character and O2 is enabled then it is necessary to reassign O1's source position to the original destination position of O2 (which includes the insertions by both O1 and O2). This is where the saved value of O2.dq comes in useful.

- Finally O2.sq is shifted left as required to exclude the insertion by O1. This must be done after having transformed O1.dq so it includes the insertion by O2.

A note on merging

The choice of whether extraction coordinates are expressed in pre or post insertion coordinates relates back to the requirements of merging.

When a contextually serialized sequence of operations [O1, …, On] are merged into a single operation O we are happy to lose information about the individual edits. However, we may not be prepared to lose the "redundant" changes to the effects document. These are insertions that were subsequently removed.

If an operation stores extractions in pre-insertion coordinates then it is unable to represent insertions that are subsequently extracted. This is good or bad depending on the point of view. A problem is that it actually drops information about changes to the effects document. An advantage is that the compression is stronger.

When we calculate a difference from a repository it is necessary to store detailed site identity information in the extraction/insertion intervals. This is needed to correctly support IT/ET. It is also necessary to store the extractions in post-insertion coordinates.

Operation representation

Following [5] we only store q-positions in operations, and assume that the documents store the p-q map, allowing q-positions to be mapped to p-positions when operations are executed, and p-positions to be mapped to q-positions when operations are generated.

It is useful to think of operations as representing insertions in the effects documents, rather than move operations! The extractions are more of a book keeping device to ensure the present flags on the characters are maintained correctly to give the illusion of conservation of matter.

An operation stores the following information

- For each destination effects document, a list of insertion intervals ordered from left to right
- For each source effects document, a list of extraction intervals ordered from left to right
- An intermediate buffer area that holds the values of the characters being extracted or inserted. It is permissible for a character to be inserted without having been extracted to emulate creation of matter, or for a character to be extracted without being inserted to emulate destruction of matter. Let index r be used to index into this intermediate buffer.
- Each insertion or extraction interval stores its r-position. This has the effect of tying extractions and insertions.

When two operations conflict because they are moving the same character it is necessary to track source positions to the other operation's destination position. Therefore we see that we need to efficiently map from source position to destination position.

When doing an operation we first perform the insertions by iterating through the insertion intervals from left to right. We need the enable status in order to correctly update the p-q map. Then we perform the extraction. Again we scan left to right, and for each enabled extraction interval we clear the present status in the effects document.

Conclusions

- Need fast left-right iteration through insertion intervals
- Need fast left-right iteration through extraction intervals
- Need fast navigation to enable status of a given insertion interval
- Need fast navigation to enable status of a given extraction interval
- Need fast navigation to from extraction position to insertion position

The operation is *persisted* as a single linear array of MoveInterval.

```
struct MoveInterval
{
    SiteId id;
    int e;
    DocId si;
    int sq;
    DocId di;
    int dq;
    std::string str;
};
```

This is quite an efficient representation on disk.

It is important to realize that the order of the intervals is not significant. The q-position coordinates are expressed in post-insertion coordinates, for *all* insertions. Therefore, the only way to make sense of the positions is to build data structures that order the intervals by q-position independently for each document.

To support this, the MoveInterval structure can be augmented with four pointers as follows…

```
struct MoveInterval
{
    SiteId id;
    int e;
    DocId si;
    int sq;
    DocId di;
    int dq;
    std::string str;
    mutable MoveInterval* prevX;
    mutable MoveInterval* nextX;
    mutable MoveInterval* prevI;
    mutable MoveInterval* nextI;
};
```

These pointers are redundant, and allow MoveInterval nodes to simultaneously take part in two double linked lists at the same time.

After reading the MoveInterval's from disk it should be possible to initialize the pointers using auxiliary maps.

Move operation
A move operation is represented as follows

```
struct MultiCharMoveOp
{
    std::map<DocId,MoveInterval*> X;
    std::map<DocId,MoveInterval*> I;
};
```

Member X is a map from document ID to a pointer to the first (ie left-most) extraction MoveInterval on that document.

Similarly member I is a map from document ID to a pointer to the first (ie left-most) insertion MoveInterval on that document.

Specialising for pure deletes and inserts

For extra space efficiency it may be worth specializing for pure deletes and inserts. This will complicate the code, but the reduction in space could be significant.

struct InsertInterval
{
```
    SiteId id;
    DocId di;
    int dq;
    std::string str;
    mutable MoveInterval* prevI;
    mutable MoveInterval* nextI;
```
};

struct DeleteInterval
{
```
    SiteId id;
    int e;
    DocId si;
    int sq;
    std::string str;
    mutable MoveInterval* prevX;
    mutable MoveInterval* nextX;
```
};

Note furthermore that pure inserts or deletes can easily be persisted in the order in which they are applied to the intervals.

DualIT

The DualIT function is broken up into four simpler functions as follows:-

```
void DualIT(MultiCharMoveOp& O1, MultiCharMoveOp& O2)
{
    DualIT_ii(O1,O2);
    DualIT_id(O1,O2);
    DualIT_id(O2,O1);
    DualIT_dd(O1,O2);

    O1.MergeAdjacentIntervals();
    O2.MergeAdjacentIntervals();
}
```

DualIT_ii() transforms insertions against insertions.

```
void DualIT_ii(MultiCharMoveOp& O1, MultiCharMoveOp& O2)
{
    for (MultiCharMoveOp::MAP::iterator m1 = O1.I.begin() ; m1 != O1.I.end() ; ++m1)
    {
        MultiCharMoveOp::MAP::iterator m2 = O2.I.find(m1->first);
        if (m2 != O2.I.end())
        {
            int s1 = 0, s2 = 0;
            MoveInterval* i1 = m1->second; MoveInterval* i2 = m2->second;
            while(i1 && i2)
            {
                int d = (s1 + i2->dq) - (s2 + i1->dq);
                if (d < 0 || d == 0 && i2->id < i1->id)
                {
                    i2->dq += s1; s2 += i2->size(); i2 = i2->nextI;
                }
                else
                {
                    i1->dq += s2; s1 += i1->size(); i1 = i1->nextI;
```

```
                }
            }
            if (s1 != 0) while (i2) { i2->dq += s1; i2 = i2->nextI; }
            if (s2 != 0) while (i1) { i1->dq += s2; i1 = i1->nextI; }
        }
    }
}
```

DualIT_ii() iterates through all the map entries in O1.I, and for each entry finds the corresponding entry in O2.I (if any). For each document, i1 and i2 are pointers to the insertion MoveIntervals from O1 and O2 respectively. These are processed in a single left to right scan through the document. Insertion positions in O_1 and O_2 need to be shifted to the right. We accumulate the number of inserted characters by O_1 in a local variable s1, and by O_2 in a local variable s2 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.

After performing DualIT_ii() the insertions are expressed in post-insertion coordinates. Therefore it is now possible to adjust source positions in order to track the new locations of characters.

DualIT_id(O1,O2) transforms O2 against the insertions by O1. If an insertion falls in the middle of an extraction interval of O2 then it is necessary to first split the interval because the right piece will be shifted more than the left piece. Note that the implementation allows a single interval to be split multiple times as required.

```
void DualIT_id(MultiCharMoveOp& O1, MultiCharMoveOp& O2)
{
    for (MultiCharMoveOp::MAP::iterator m1 = O1.I.begin() ; m1 != O1.I.end() ; ++m1)
    {
        MultiCharMoveOp::MAP::iterator m2 = O2.X.find(m1->first);
        if (m2 != O2.X.end())
        {
            int shift = 0;
            MoveInterval* i = m1->second; MoveInterval* x = m2->second;
            while(i && x)
            {
                if (i->dq <= shift + x->sq)
                {
                    shift += i->size();
                    i = i->nextI;
                }
                else
                {
                    int d = i->dq - (shift + x->sq);
                    if (d < x->size())
                    {
                        O2.SplitInterval(x, i->dq - (shift + x->sq));
                    }
                    x->sq += shift;
                    x = x->nextX;
                }
            }
            if (shift != 0) while (x) { x->sq += shift; x = x->nextX; }
        }
    }
}
```

DualIT_dd() transforms extractions against extractions. For each document the extraction intervals are processed in a single left to right scan. The next interval from O1 and next interval from O2 may partically overlap. In that case it is necessary to split the MoveInterval. Therefore it is possible to process the intervals that perfectly overlap. The processing of these intervals involves tracking the source position to the destination position of the other interval (if it is enabled). This is difficult to do synchronously because we are in the process of iterating through the STL map and double linked lists of extraction intervals. Therefore the tracking of source positions is post-poned to a separate phase.

```
void DualIT_dd(MultiCharMoveOp& O1, MultiCharMoveOp& O2)
{
```

```
    std::vector<SetNewExtractionPosCommand> N1;
    std::vector<SetNewExtractionPosCommand> N2;
    for (MultiCharMoveOp::MAP::iterator m1 = O1.X1.begin() ; m1 != O1.X1.end() ; ++m1)
    {
        MultiCharMoveOp::MAP::iterator m2 = O2.X2.find(m1->first);
        if (m2 != O2.X2.end())
        {
            MoveInterval* x1 = m1->second; MoveInterval* x2 = m2->second;
            while(x1 && x2)
            {
                if (x1->sq + x1->size() <= x2->sq)
                {
                    x1 = x1->nextX;
                }
                else if (x2->sq + x2->size() <= x1->sq)
                {
                    x2 = x2->nextX;
                }
                else
                {
                    if (x1->sq < x2->sq)
                    {
                        O1.SplitInterval(x1, x2->sq - x1->sq);
                        x1 = x1->nextX;
                    }
                    else if (x2->sq < x1->sq)
                    {
                        O2.SplitInterval(x2, x1->sq - x2->sq);
                        x2 = x2->nextX;
                    }
                    if (x2->size() < x1->size())
                    {
                        O1.SplitInterval(x1, x2->size());
                    }
                    else if (x1->size() < x2->size())
                    {
                        O2.SplitInterval(x2, x1->size());
                    }

                    if (x1->e == 0)
                    {
                        N2.push_back(SetNewExtractionPosCommand(x2, x1->di, x1->dq));
                    }
                    if (x2->e == 0)
                    {
                        N1.push_back(SetNewExtractionPosCommand(x1, x2->di, x2->dq));
                    }
                    if (x2->e < x1->e || x2->e == x1->e && x2->id < x1->id) ++x1->e; else ++x2->e;
                    x1 = x1->nextX;
                    x2 = x2->nextX;
                }
            }
        }
    }
    ApplyNewExtractionPosCommands(O1,N1);
    ApplyNewExtractionPosCommands(O2,N2);
}
```

Transpose

TransposeConcurrent_di(O1,O2) transposes extractions by O1 with insertions by O2. For each document the extractions/insertion intervals are processed in a single left to right scan.

```
void TransposeConcurrent_di(MultiCharMoveOp& O1, MultiCharMoveOp& O2)
{
    for (MultiCharMoveOp::MAP::iterator m1 = O1.X.begin() ; m1 != O1.X.end() ; ++m1)
    {
        MultiCharMoveOp::MAP::iterator m2 = O2.I.find(m1->first);
        if (m2 != O2.I.end())
        {
            int shift = 0;
            MoveInterval* x = m1->second; MoveInterval* i = m2->second;
            while(x && i)
            {
                if (i->dq <= shift + x->sq)
                {
```

```
                    shift += i->size(); i = i->nextI;
                }
                else
                {
                    int d = i->dq - (shift + x->sq);
                    if (d < x->size())
                    {
                        O1.SplitInterval(x, i->dq - (shift + x->sq));
                    }
                    x->sq += shift;
                    x = x->nextX;
                }
            }

            if (shift)
            {
                while (x) { x->sq += shift; x = x->nextX; }
            }
        }
    }
}
```

From the single character move algorithm we see that when the tracking rule is applied to the source position, the original destination position of O2 must be used rather than the transformed position. To achieve this we delay the transformation of O2 destination positions. More specifically `TransposeConcurrent_ii_1()` is called initially to only transform O1 destination positions, and `TransposeConcurrent_ii_2()` is used later to transform the O2 destination positions.

```
void TransposeConcurrent_ii_1(MultiCharMoveOp& O1, MultiCharMoveOp& O2)
{
    for (MultiCharMoveOp::MAP::iterator m1 = O1.I.begin() ; m1 != O1.I.end() ; ++m1)
    {
        MultiCharMoveOp::MAP::iterator m2 = O2.I.find(m1->first);
        if (m2 != O2.I.end())
        {
            int s2 = 0;
            MoveInterval* i1 = m1->second; MoveInterval* i2 = m2->second;
            while(i1 && i2)
            {
                if (i2->dq <= s2 + i1->dq) { s2 += i2->size(); i2 = i2->nextI; }
                else                       { i1->dq += s2;     i1 = i1->nextI; }
            }
            if (s2)
            {
                while (i1) { i1->dq += s2; i1 = i1->nextI; }
            }
        }
    }
}

void TransposeConcurrent_ii_2(MultiCharMoveOp& O1, MultiCharMoveOp& O2)
{
    for (MultiCharMoveOp::MAP::iterator m1 = O1.I.begin() ; m1 != O1.I.end() ; ++m1)
    {
        MultiCharMoveOp::MAP::iterator m2 = O2.I.find(m1->first);
        if (m2 != O2.I.end())
        {
            int s1 = 0;
            MoveInterval* i1 = m1->second; MoveInterval* i2 = m2->second;
            while(i1 && i2)
            {
                if (i1->dq + i1->size() <= i2->dq) { s1 += i1->size(); i1 = i1->nextI; }
                else if (i2->dq + i2->size() <= i1->dq) { i2->dq -= s1; i2 = i2->nextI; }
            }
            if (s1)
            {
                while (i2) { i2->dq -= s1; i2 = i2->nextI; }
            }
        }
    }
}
```

```
void TransposeConcurrent_dd(MultiCharMoveOp& O1, MultiCharMoveOp& O2)
{
    std::vector<SetNewExtractionPosCommand> N1;
    std::vector<SetNewExtractionPosCommand> N2;
    {
        for (MultiCharMoveOp::MAP::iterator m1 = O1.X1.begin() ; m1 != O1.X1.end() ; ++m1)
        {
            MultiCharMoveOp::MAP::iterator m2 = O2.X2.find(m1->first);
            if (m2 != O2.X2.end())
            {
                MoveInterval* x1 = m1->second; MoveInterval* x2 = m2->second;
                while(x1 && x2)
                {
                    if (x1->sq + x1->size() <= x2->sq)
                    {
                        x1 = x1->nextX;
                    }
                    else if (x2->sq + x2->size() <= x1->sq)
                    {
                        x2 = x2->nextX;
                    }
                    else
                    {
                        if (x1->sq < x2->sq)
                        {
                            O1.SplitInterval(x1, x2->sq - x1->sq);
                            x1 = x1->nextX;
                        }
                        else if (x2->sq < x1->sq)
                        {
                            O2.SplitInterval(x2, x1->sq - x2->sq);
                            x2 = x2->nextX;
                        }
                        if (x2->size() < x1->size())
                        {
                            O1.SplitInterval(x1, x2->size());
                        }
                        else if (x1->size() < x2->size())
                        {
                            O2.SplitInterval(x2, x1->size());
                        }
                        if (x1->e < x2->e) --x2->e; else ++x1->e;
                        if (x2->e == 0)
                        {
                            N1.push_back(SetNewExtractionPosCommand(x1, x2->di, x2->dq));
                        }
                        x1 = x1->nextX;
                        x2 = x2->nextX;
                    }
                }
            }
        }
    }

    {
        for (MultiCharMoveOp::MAP::iterator m1 = O1.I.begin() ; m1 != O1.I.end() ; ++m1)
        {
            MultiCharMoveOp::MAP::iterator m2 = O2.X.find(m1->first);
            if (m2 != O2.X.end())
            {
                MoveInterval* i1 = m1->second; MoveInterval* x2 = m2->second;
                while(i1 && x2)
                {
                    if (i1->dq + i1->size() <= x2->sq)
                    {
                        i1 = i1->nextI;
                    }
                    else if (x2->sq + x2->size() <= i1->dq)
                    {
                        x2 = x2->nextX;
                    }
                    else
                    {
                        if (i1->dq < x2->sq)
                        {
                            O1.SplitInterval(i1, x2->sq - i1->dq);
```

```
                            i1 = i1->nextI;
                        }
                        else if (x2->sq < i1->dq)
                        {
                            O2.SplitInterval(x2, i1->dq - x2->sq);
                            x2 = x2->nextX;
                        }
                        if (x2->size() < i1->size())
                        {
                            O1.SplitInterval(i1, x2->size());
                        }
                        else if (i1->size() < x2->size())
                        {
                            O2.SplitInterval(x2, i1->size());
                        }
                        N2.push_back(SetNewExtractionPosCommand(x2, i1->si, i1->sq));
                        if (i1->e < x2->e) --x2->e; else ++i1->e;
                        if (x2->e == 0)
                        {
                            N1.push_back(SetNewExtractionPosCommand(i1, x2->di, x2->dq));
                        }
                        i1 = i1->nextI;
                        x2 = x2->nextX;
                    }
                }
            }
        }
    }
    ApplyNewExtractionPosCommands(O1,N1);
    ApplyNewExtractionPosCommands(O2,N2);
}



void TransposeConcurrent_id(MultiCharMoveOp& O1, MultiCharMoveOp& O2)
{
    for (MultiCharMoveOp::MAP::iterator m1 = O1.I.begin() ; m1 != O1.I.end() ; ++m1)
    {
        MultiCharMoveOp::MAP::iterator m2 = O2.X.find(m1->first);
        if (m2 != O2.X.end())
        {
            int shift = 0;
            MoveInterval* i1 = m1->second; MoveInterval* x2 = m2->second;
            while(i1 && x2)
            {
                if (i1->dq + i1->size() <= x2->sq)
                {
                    shift += i1->size();
                    i1 = i1->nextI;
                }
                else if (x2->sq + x2->size() <= i1->dq)
                {
                    x2->sq -= shift;
                    x2 = x2->nextX;
                }
                else
                {
                    if (i1->dq < x2->sq)
                    {
                        O1.SplitInterval(i1, x2->sq - i1->dq);
                        i1 = i1->nextI;
                    }
                    else if (x2->sq < i1->dq)
                    {
                        O2.SplitInterval(x2, i1->dq - x2->sq);
                        x2 = x2->nextX;
                    }
                    if (x2->size() < i1->size())
                    {
                        O1.SplitInterval(i1, x2->size());
                    }
                    else if (i1->size() < x2->size())
                    {
                        O2.SplitInterval(x2, i1->size());
                    }
                    x2->sq -= shift;
                    x2 = x2->nextX;
```

```
                    i1 = i1->nextI;
                }
            }

            if (shift)
            {
                while (x2) { x2->sq -= shift; x2 = x2->nextX; }
            }
        }
    }
}

void Transpose(MultiCharMoveOp& O1, MultiCharMoveOp& O2)
{
    TransposeConcurrent_di(O1,O2);
    TransposeConcurrent_ii_1(O1,O2);
    TransposeConcurrent_dd(O1,O2);
    TransposeConcurrent_id(O1,O2);
    TransposeConcurrent_ii_2(O1,O2);
    O1.MergeAdjacentIntervals();
    O2.MergeAdjacentIntervals();
}
```

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.
[5]	David Barrett-Lennard. Operational transform – Multi-character insert and delete operations. August 2005.
[6]	David Barrett-Lennard. Operational transform - Single character move operations. Sep 2006.