Repository - Single char operations on a text field

Mar 2008

David Barrett-Lennard

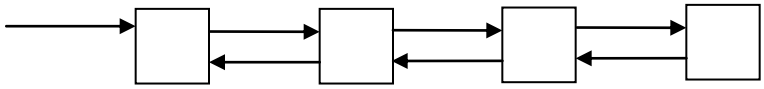
Introduction

This document concerns the representation within a ceda repository of all checked in single character insertion operations to a single text field of a single object.

It is not expected that these algorithms will be directly implemented. In practice the data structures and algorithms need to be generalised to support stringwise insertion operations. Also there is no discussion of delete or move operations.

Representation

The ceda repository records the insertions in a double linked list as follows



Each box in this figure represents a single character insertion to the text field. The following information is recorded

s	the site identifier where the operation was originally generated
t	the assigned sequence number for the operation on that site
value	the value of the inserted character
cisn	the *check-in sequence number* (CISN) assigned by the repository.

Note that insertion positions are not recorded!

There is a total ordering on the insertions (according to the so called *effects relation*), so there is no ambiguity in the order of these insertions in the double linked list.

An implementation could store the insertions on the field in the repository as follows

```
struct ReposInsertion
{
    SiteId s;
    int t;
    char value;
    int cisn;
};

typedef list<ReposInsertion> ReposTextField;
```

Representation of an operation

The following is a suitable representation for an operation that supports information preservation under merging of insertions to a given text field:

```
struct SingleInsertionOp
{
    SiteId s;
    int t;
    char value;
    int q;
};

typedef list<SingleInsertionOp> MultiInsertionOp;
```

This representation allows for a composite operation to record the original (s,t) on the individual character insertions. It is assumed that the list is sorted by q-position, and the q-positions are in *post insertion coordinates*.

Effects relation

Definition: We write $O_1 <_e O_2$ if the character inserted by O_1 appears on the left of the character inserted by O_2 in all document states where both O_1 and O_2 have been executed. This is a total ordering on all the single character insertion operations. The insertion operations recorded in this repository are uniquely ordered according to this *effects relation*.

i.e. $O_1 <_e O_2 \quad \Leftrightarrow \quad O_1$ appears on left of O_2 in the repository.

Check-in sequence number

The entire repository is assumed to record a single zero based *check in sequence number* (CISN). This is incremented for each check-in to the repository, irrespective of the number of objects or fields modified by a single check-in.

Each single character insertion in the repository records its associated CISN. Note that once assigned, the CISN is immutable.

[Note: Later when we generalise to stringwise insertion operations, we will assume that when multi-character intervals are split into smaller intervals, the CISN is inherited by the sub-intervals]

A single check-in may involve multiple single character insertions at various positions within a single text field. Therefore we don't expect the CISN to be unique for all the insertions on a single text field.

Calculating the state for a given vector time

For vector time v, consider that in the linked list we filter out (i.e. remove) the single character insertions that are not in $\chi(v)$. Then remaining we see a sequence of character insertions that have index positions 0,1,2,... in the state corresponding to the given vector time.

Therefore for vector time v, the value of the field can be found by iterating through the insertions from left to right building up the string using the character insertions satisfying $t < v(s)$.

```
String GetFieldValue(ReposTextField& f, VectorTime v)
{
    String s;
    for each i in f
    {
        if (i.t < v(i.s)) s += i.value;
    }
    return s;
}
```

Calculating the difference between two given vector times

Let vector times v_1,v_2 be given satisfying $v_1 \leq v_2$. We require a function that represents the state difference $\chi(v_2) \setminus \chi(v_1)$ as an operation (a `MultiInsertionOp`). This operation is assumed to be applied in the context of $\chi(v_1)$ to transition the state to the one that corresponds to $\chi(v_2)$.

In the following, the q-position coincides with the document state associated with v_2. The insertions are therefore using *post insertion coordinates* (as required by definition of a `MultiInsertionOp`).

```
MultiInsertionOp GetDiff(ReposTextField& f, VectorTime v1, VectorTime v2)
{
    MultiInsertionOp o;
    int q = 0;
    for each r in f
    {
        if (r.t < v2(r.s))
        {
            if (r.t >= v1(r.s))
            {
                o += SingleInsertionOp(r.s, r.t, r.value, q);
            }
            ++q;
        }
    }
    return o;
}
```

Check-In

Consider the check-in of operation O which is a `SingleInsertionOp`, with execution context given by vector time v = ec(O). i.e. O is assumed to be executed in the context of executing all operations in $\chi(v)$.

For the given vector time v we can distinguish between single character insertions that are in $\chi(v)$ versus not in $\chi(v)$. As an example, suppose we want to check-in O with O.q = 1, and the insertions in the repository can be depicted as follows

$$[\ x_0\ \ x_1\ \ q_0\ \ x_2\ \ x_3\ \ x_4\ \ q_1\ \ q_2\ \ q_3\ \ x_5\ \ x_6\ \ q_4\ ]$$

where the q_i depict insertions in $\chi(v)$ and the x_i depict insertions not in $\chi(v)$.

Claim: for each i, $O \parallel x_i$.

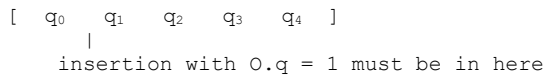
Proof:

$x_i \notin \chi(v)$ (distinction between x_i and q_i)

$x_i \notin \chi(ec(O))$	(because v = ec(O))
$\neg(x_i \rightarrow O)$	(contrapositive of $x_i \rightarrow O \Rightarrow x_i \in \chi(ec(O))$ from [1])
Also $\neg(O \rightarrow x_i)$	(x_i checked in before O, so $O \notin \chi(gc(x_i))$)
So $O \parallel x_i$	

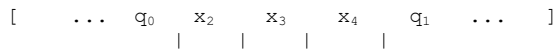
It follows therefore that for each i, $O.s \neq x_i.s$.

We interpret the insertion position O.q = 1 with respect to the index positions of the insertions with all the x_i removed. i.e.



This implies that O must be inserted after q_0 but before q_1.

In the original full list of operations, we actually have x_2,x_3,x_4 between q_0,q_1. Therefore there are four possible insertion positions:



In turns out there is a unique position where O must be inserted. Let the z_i refer to x_2,x_3,x_4 in this example.

Suppose firstly that the z_i are mutually concurrent operations. i.e. for each i,j $z_i \parallel z_j$. This implies they were generated on different sites. Furthermore the z_i will be ordered according to siteid. i.e. $z_i <_e z_j \Leftrightarrow z_i.s < z_j.s$. Clearly O must be inserted in order to maintain the ordering by site identifier. This uniquely defines the insertion position amongst the z_i.

However more generally there can be causal orderings amongst the z_i, and this complicates the problem of where to correctly insert O amongst the z_i.

The CISN provides an ordering on the z_i that is compatible with the precedes relation (because check-ins are done in an order that is compatible with the precedes relation).

For the purpose of writing the algorithm in C++ style notation let $Z[i] = z_i$.

Consider the following example

	z_0	z_1	z_2	z_3	z_4
cisn	8	11	3	5	9

where cisn is the assigned CISN. This means that the relative check-in order is

$[z_2, z_3, z_0, z_4, z_1]$.

Consider that we define C[] as the array of pairs (i,cisn) where i is the index position in Z[] and cisn is the CISN as follows

C[] = { (0,8), (1,11), (2,3), (3,5), (4,9) }

C[] can be initialised using the following code

```
for (int i=0 ; i < |Z| ; ++i)
{
    C[i].i = i;
    C[i].cisn = Z[i].cisn;
}
```

Let C[] be sorted using its second coordinate cisn to yield

C'[] = { (2,3), (3,5), (0,8), (4,9), (1,11) }

Then reading out the index positions we obtain

L[] = { 2,3,0,4,1 }

From L we can calculate insertion q-positions for the contextually serialised sequence of operations X = [z_2, z_3, z_0, z_4, z_1] as follows

	z_2	z_3	z_0	z_4	z_1
q	0	1	0	3	1

Q[] = { 0,1,0,3,1 } can be obtained by processing each element of L in turn, calculating the insertion position with respect to the sorted elements M from L so far.

After applying this contextually serialised sequence of operations	Document state is associated with	L[] = last applied operation	Q[] = q of last applied operation
[z_2]	[z_2]	z_2	0
[z_2, z_3]	[z_2, z_3]	z_3	1
[z_2, z_3, z_0]	[z_0, z_2, z_3]	z_0	0
[z_2, z_3, z_0, z_4]	[z_0, z_2, z_3, z_4]	z_4	3
[z_2, z_3, z_0, z_4, z_1]	[z_0, z_1, z_2, z_3, z_4]	z_1	1

```
M = [];
for (int i=0 ; i < |L| ; ++i)
{
    int j;
    for (j=0 ; j < |M| ; ++j)
    {
        if (L[i] < M[j]) break;
    }
    M.insert(j,L[i]);    // Insert L[i] into M at position j
    Q[i] = j;
}
```

Let S[] be the site identifiers corresponding to the operations in X. S is initialised as follows

```
for (int i=0 ; i < |L| ; ++i)
{
    S[i] = Z[L[i]].s;
}
```

Given operation O with site identifier O.s, we assume initially its insertion position is O.q = 0, and we simply IT O past X to find where O will be inserted in the context of X.

```
SiteId s = O.s;
int q = 0;
for (int i=0 ; i < |L| ; ++i)
{
    if (q < Q[i] || q == Q[i] && s < S[i]) ++q;
}
```

The final value of q is the insertion position of O with respect to Z[].

The last three code snippets can be combined as follows:

```
SiteId s = O.s;
int q = 0;
M = [];
for (int i=0 ; i < |L| ; ++i)
{
    int j;
    for (j=0 ; j < |M| ; ++j)
    {
        if (L[i] < M[j]) break;
    }
    M.insert(j,L[i]);    // Insert L[i] into M at position j
    if (q < j || q == j && s < Z[L[i]].s) ++q;
}
```

References

[1]	David Barrett-Lennard. Vector Time. Feb 2008