A Log Compression Algorithm for Operation-based Version Control Systems

A Log Compression Algorithm for Operation-based Version Control Systems

Haifeng Shen and Chengzheng Sun
School of Computing and Information Technology
Griffith University
Brisbane, Qld 4111, Australia
{Hf.Shen, C.Sun}@cit.gu.edu.au

Abstract

Version control systems are widely used to support distributed concurrent software development, where document merging is a key function. Most existing systems adopt state-based merging, which relies on the derivation of deltas among documents. The derivation of deltas involves transferring documents over the network and executing time-consuming text differentiation algorithms, which may result in a poor system response. Operation-based merging saves executed operations in logs as deltas, thus eliminating the need for deriving deltas. However, for the operation-based merging to be adopted in version control systems, a major technical challenge is how to keep the size of logs small so that it requires less time to transfer the log over the network and to re-execute operations in the log. In this paper, we contribute a novel compression algorithm, which is able to minimize the size of a log as well as the number of operations within it. It has been proven both correct and complete in the sense that the compressed log has the same effect as the original one and operations that can be merged have already been merged.

1. Introduction

It is very common that multiple developers are involved in developing a large software project. These developers, possibly dispersed in different offices or even scattered over the world, need to simultaneously modify the code. Version control systems are therefore widely used nowadays to facilitate distributed software development. A version control system works in the manner of *Copy-Modify-Merge* [2]: each developer checks out a working copy of a source code from the repository; then modifies her/his working copy independently; and finally merges her/his working copy with other copies. *Merging* is the process of integrating multiple documents to generate a new one, which is an essential function is version control systems.

As shown in Figure 1, merging may happen at two stages. One is at the *committing stage* when a working copy is committed into the repository to form a new version that is the same as that working copy. At this stage, updates made in that working copy are merged into its original copy in the repository. The other is at the *updating stage* when a working copy is updated by some committed working copies (i.e. versions) from the repository. At this stage, updates made in those committed working copies are merged into that working copy.

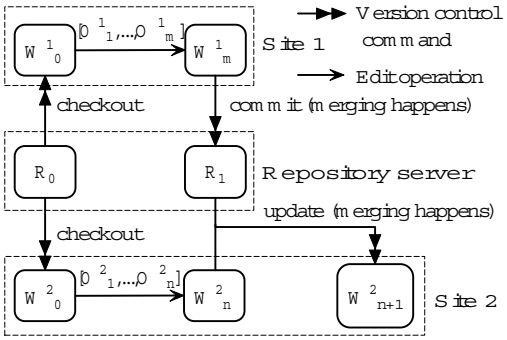

Figure 1. Merging happens at two stages

Most existing systems, such as *RCS* (Revision Control System) [10], *CVS* (Concurrent Versions Systems) [1], and *NSE* (Network Software Environment), adopt the *state-based merging* approach, and therefore can be called state-based version control systems. State-based merging [4, 8], as the name implies, is performed by comparing different document states to generate *deltas* among them, and then applying these deltas on one of the document state to generate one. However, the derivation of deltas involves transferring documents over the network and executing time-consuming text differentiation algorithms, which may result in a poor system response.

As an alternative to state-based merging, *operation-based merging* [4, 8], saves actually performed operations on a document in a log as the deltas between the current

Proceedings of the 26 th Annual International Computer Software and Applications Conference (COMPSAC'02)
0730-3157/02 $17.00 © 2002 IEEE

document state and the initial document state. Therefore the need for deriving deltas can be eliminated, which could result in a better system response. Operation-based merging is done by re-executing operations performed in one working copy in another. For the operation-based merging approach to be adopted in version control systems, a major technical challenge is how to keep the size of logs small so that it requires less time to transfer the log over the network and to re-execute operations in the log.

In this paper, we contribute a novel compression algorithm, which is able to minimize the size of a log as well as the number of operations within it by the proposed technique of operational merging and the technique of operational transformation [9]. The algorithm has been implemented in the *FORCE* (Flexible Operation-based Revision Control Environment) [8].

The paper is organized as follows. Problems and related work is described following the introduction. Then the next section systematically present the compression algorithm. Two important properties of the algorithm is proven subsequently. Finally the paper is concluded with our major contributions and future work.

2. Problems and related work

Two parameters have been defined to measure the merging performance in distributed environments. One is *MD* (Merging Duration), which is the time interval between the user issues a merging command (*commit/update*) [1] and a new document state has been generated. The other parameter is *SR* (System Response), which is the time interval between the user issues a merging command (*commit/update*) [1] and the user is able to continue issuing another command.

For the configuration in Figure 1, when the user at *Site 1* issues the *commit* command [1] to generate a new version $R_1 = W_m^1$ in the repository, the *MD* and *SR* parameters at this stage in state-based merging and operation-based merging are illustrated in Figure 2(a) and (b) respectively. When the user at *Site 2* issues the *update* command [1] to merge version R_1 into her/his working copy W_n^2 to produce a new document state W_{n+1}^2, the *MD* and *SR* parameters at this stage in state-based merging and operation-based merging are illustrated in Figure 2(c) and (d) respectively.

It can be seen from Figure 2 that operation-based merging normally has a shorter *MD* and a better *SR* than state-based merging does. The reason is operation-based merging has eliminated the time for deriving deltas, which requires transferring full state of documents over the network between working sites and the repository, and executing time-consuming text differentiation algorithms, such as *diff* [7] and *diff3* [5].

However, there is a major technical challenge for operation-based merging to be adopted in version control

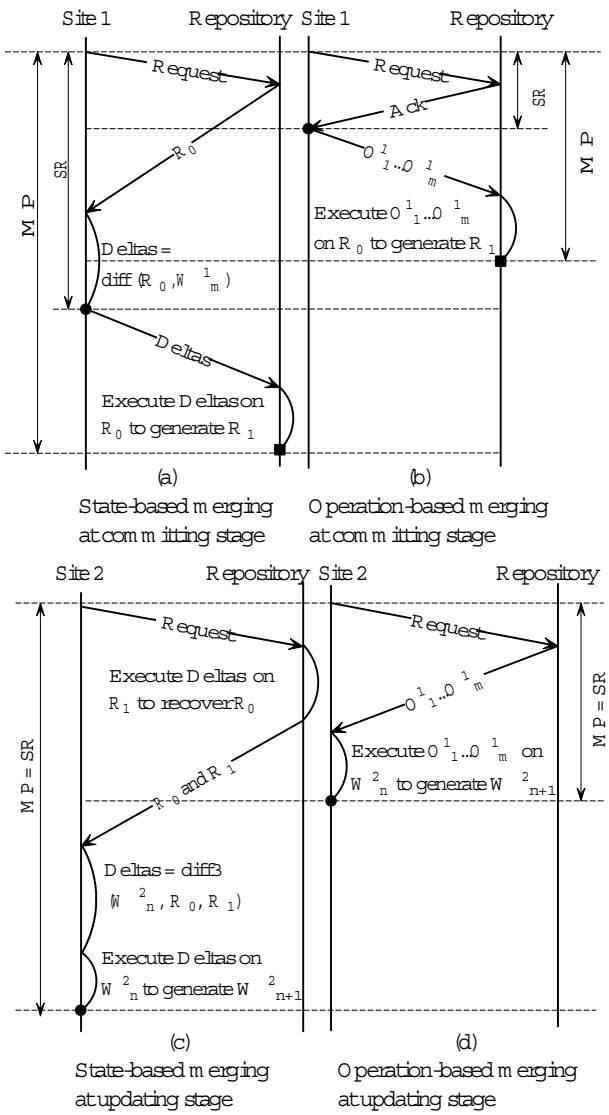

Figure 2. Merging performance

systems, that is how to control the size of logs that store saved operations. In a distributed team-working environment, each participant works independently on her/his own working copy and her/his independent work tends to be very long. With executed operations accumulated in a log, the log could grow very huge, resulting in a long merging duration and a poor system response because the larger a log is, the longer transferring it over the network plus executing operations within it takes. It is even possible that a log is so large that an operation-based merging process has a longer merging duration and a poorer system response than a corresponding state-based merging process does.

This issue was raised in [3, 6], but few solutions were proposed. The solution in [3] is to propagate the full

database state from one site to another when replicas at both sites are far from synchronized and then to prune their logs accordingly. But this solution is not applicable for version control systems because it is infeasible to check the current states of various working copies at different sites if none of those working copies has been committed into the repository. Our proposal is to compress logs. Compression is achieved by the proposed operational merging technique and operational transformation technique [9] has been be applied to achieving full compression.

3. The compression algorithm

3.1. Concepts

Two types of primitive operations can be abstracted. An *insert* operation is denoted as $INS[P, L, S]$ to insert string S whose length is L at position P. A *delete* operation is denoted as $DEL[P, L, S]$ to delete string S whose length is L at position P. Given any operation O = *INS/DEL*$[P, L, S]$, $P(O) = P$ is used to denote O's position parameter, $L(O) = L$ is used to denote O's length parameter, $S(O) = S$ is used to denote O's string parameter, and $T(O) = INS/DEL$ is used to denote O's type.

Definition 1 *Operation context*
Given an operation O, its *context*, denoted as CT_O, is the document state on which O's parameters are defined.

Definition 2 *Context preceding relation "* $\mapsto$*"*
Given two operation O_a and O_b, O_a is context preceding O_b, denoted as $O_a \mapsto O_b$, $iff\ CT_{O_b} = CT_{O_a} \circ [O_a]$.

So given a log $L = [O_1, \cdots, O_n]$ performed on the initial document state S_0, $CT_{O_1} = S_0$, $CT_{O_i} = CT_{O_{i-1}} \circ [O_{i-1}]$ $(1 < i \leq n)$, and $O_{i-1} \mapsto O_i$ $(1 < i \leq n)$.

Definition 3 *Operation overlapping relation "* $\oplus$*"*
Given two operations O_a and O_b, where $O_a \mapsto O_b$, then O_a and O_b are overlapping, denoted as $O_a \oplus O_b$, *iff*:

1. $T(O_a) = T(O_b) =$ *INS*, and $P(O_a) < P(O_b) < P(O_a) + L(O_a)$; or
2. $T(O_a) = T(O_b) =$ *DEL*, and $P(O_b) < P(O_a) < P(O_b) + L(O_b)$; or
3. $T(O_a) =$ *INS* and $T(O_b) =$ *DEL*, and $P(O_a) \leq P(O_b) < P(O_a) + L(O_a)$, or $P(O_b) \leq P(O_a) < P(O_b) + L(O_b)$.

Two operations are regarded overlapping if their effects are overlapping. In particular, if an insert operation inserts a string within another string inserted by its preceding insert operation, then these two operations are overlapping; if a delete operation deletes a string S, then a subsequent delete operation deletes another string that was originally separated before and after S, then these two operations are overlapping; or if a delete operation deletes a string part of which was inserted by its preceding insert operation, then these two operations are overlapping. An insert operation could never overlap with its preceding delete operation because under no circumstance a delete operation could delete anything that has not been inserted by a subsequent insert operation.

Definition 4 *Operation adjacent relation "* $\ominus$*"*
Given two operations O_a and O_b, where $O_a \mapsto O_b$, then O_a and O_b are adjacent, denoted as $O_a \ominus O_b$, *iff*:

1. $T(O_a) = T(O_b)$; and
2. $P(O_b) = P(O_a)$, or $P(O_b) + L(O_b) = P(O_a)$.

Two operations are regarded adjacent if they are the same type of operations and their effects are adjacent. In particular, if an insert/delete operation inserts/deletes a string immediately before/after another string that was inserted/deleted by its preceding insert/delete operation, then these two operations are adjacent.

Definition 5 *Operation disjointed relation "* $\odot$*"*
Given two operations O_a and O_b, where $O_a \mapsto O_b$, then O_a and O_b are disjointed, denoted as $O_a \odot O_b$, *iff* neither $O_a \oplus O_b$ nor $O_a \ominus O_b$.

For example, as shown in Figure 3, the initial document contains string *xyz*. Five operations have been executed and stored in the log $L = [O_1, O_2, O_3, O_4, O_5]$ where O_1 = *INS*[3, 1, *c*] to insert character *c* after *z*, O_2 = *INS*[3, 2, *ab*] to insert string *ab* between *z* and *c*, O_3 = *DEL*[2, 2, *za*] to delete string *za*, O_4 = *DEL*[1, 2, *yb*]to delete string *yb*, and O_5 = *INS*[2, 3, *123*] to insert string *123* after *c*.

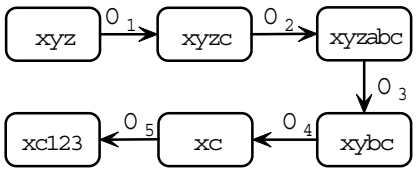

Figure 3. A log with five operations

Then $O_1 \ominus O_2$ because $T(O_1) = T(O_2) =$ *INS*, and $(P(O_2)=3) = (P(O_1)=3)$; $O_2 \oplus O_3$ because $T(O_2) =$ *INS* and $T(O_3) =$ *DEL*, and $(P(O_3)=2) < (P(O_2)=3) < (P(O_3) + L(O_3) = 2 + 2 = 4)$; $O_3 \oplus O_4$ because $T(O_3) = T(O_4) =$ *DEL*, and $(P(O_4)=1) < (P(O_3)=2) < (P(O_4) + L(O_4) = 1 + 2 = 3)$; and $O_4 \odot O_5$ because $T(O_4) =$ *DEL* and $T(O_5) = INS$.

3.2. Operational merging

A log can be compressed by merging overlapping and adjacent operations. Concretely speaking, same type of two overlapping/adjacent operations are merged to be one operation that combines their effects. Different types of two

overlapping operations can be merged by eliminating their overlapping part. Given any two operations O_a and O_b, if O_a ($\oplus$/$\ominus$) O_b, they can be merged as follows.

- If $T(O_a)$ = $T(O_b)$ = *INS/DEL*, then O_a and O_b can be merged to be one operation O where $P(O)$ is the smaller value between $P(O_a)$ and $P(O_b)$, $S(O)$ is the combination of $S(O_a)$ and $S(O_b)$, and $L(O)$ = $L(O_a)$ + $L(O_b)$. For example, in Figure 3, O_1 and O_2 can be merged to be *INS*[3, 3, *abc*], O_3 and O_4 can be merged to be *DEL*[1, 4, *yzab*].
- If $T(O_a)$ = *INS* and $T(O_b)$ = *DEL*, then O_a and O_b can be merged by removing their common substring CS from $S(O_a)$ and $S(O_b)$.
  - If CS = $S(O_a)$ = $S(O_b)$, then O_a and O_b can be merged to be an I(Identity);
  - If CS = $S(O_a)$, O_a and O_b can be merged to be one operation O where $T(O)$ = *DEL*, $P(O)$ = $P(O_b)$, $S(O)$ is obtained by removing CS from $S(O_b)$, and $L(O)$ = $L(O_b)$ - $Len(CS)$;
  - If CS = $S(O_b)$, O_a and O_b can be merged to be one operation O where $T(O) = INS$, $P(O)$ = $P(O_a)$, $S(O)$ is obtained by removing CS from $S(O_a)$, and $L(O)$ = $L(O_a)$ - $Len(CS)$;
  - Otherwise, O_a and O_b can be merged to be O'_a and O'_b where $T(O'_a)$ = $T(O_a)$, $P(O'_a)$ = $P(O_a)$, $S(O'_a)$ is obtained by removing CS from $S(O_a)$, $L(O'_a)$ = $L(O_a)$ - $Len(CS)$; $T(O'_b)$ = $T(O_b)$, $P(O'_b)$ = $P(O_b)$, $S(O'_b)$ is obtained by removing CS from $S(O_b)$, and $L(O'_b)$ = $L(O_b)$ - $Len(CS)$.

  For example, in Figure 3, O_2 and O_3 can be merged to be $O'_2 = INS$[3, 1, b] and $O'_3 = DEL$[2, 1, z].

The $OM(O_a, O_b)$ merging function is defined to merge operation O_a and O_b provided $O_a \mapsto O_b$.

Function 1 *OM* (O_a, O_b): (O'_a, O'_b)
{ **if** ($T(O_a)$ = *INS* **and** $T(O_b)$ = *INS*) $OM\_II(O_a, O_b)$;
if ($T(O_a)$ = *INS* **and** $T(O_b)$ = *DEL*) $OM\_ID(O_a, O_b)$;
if ($T(O_a)$ = *DEL* **and** $T(O_b)$ = *DEL*) $OM\_DD(O_a, O_b)$;
if ($T(O_a)$ = *DEL* **and** $T(O_b)$ = *INS*) **return** (O_a, O_b);
}

As an example, the OM_II function looks like:

Function 2 OM_II (O_a, O_b)
{ **if** $P(O_b) \geq P(O_a)$ **and** $P(O_b) \leq P(O_a)$+$L(O_a)$
{ $head$ = **Substring**($S(O_a)$, 0, $P(O_b)$-$P(O_a)$);
$tail$ = **Substring**($S(O_a)$, $P(O_b)$-$P(O_a)$, $L(O_a)$);
$T(O)$ = $T(O_a)$; $P(O)$ = $P(O_a)$;
$L(O)$ = $L(O_a)$+$L(O_b)$; $S(O)$ = $head$+$S(O_b)$+$tail$;
return (I, O);
}
return (O_a, O_b);
}

3.3. Control algorithm

After applying the OM merging function on the log L = $[O_1, \cdots, O_5]$ in Figure 3 to merge overlapping and adjacent operations, the compressed log L_{com} would be [CO_1, CO_2, CO_3] where CO_1 = *INS*[3, 1, c], CO_2 = *DEL*[1, 2, yz], and CO_3 = O_5 = *INS*[2, 3, *123*]. The log cannot be compressed anymore because $CO_1 \odot CO_2$ and $CO_2 \odot CO_3$. But intuitively we know CO_1 and CO_3 should be able to be merged because their effects are adjacent.

Given an operation O_i in a log L, O_i could not only be overlapping/adjacent with its preceding neighbor O_{i-1} or subsequent neighbor O_{i+1}, but also could be overlapping or adjacent with any operation $O_j (j \neq i)$. Therefore a control algorithm is needed to exhaust all merging opportunities in order to achieve a fully compressed log. A log is regarded as *fully compressed* if all operations within it are disjointed. The idea is to make $O_j \mapsto O_i$ or $O_i \mapsto O_j$ so that their parameters are directly comparable to determine their relationship by the technique of operational transformation [9].

There are two types of primitive transformation functions [9]: one is the *Inclusion Transformation* function – $IT(O_a, O_b)$, which transforms operation O_a against operation O_b in such a way that the impact of O_b is effectively included in the parameters of the output operation O'_a; and the other is the *Exclusion Transformation* function – $ET(O_a, O_b)$, which transforms O_a against O_b in such a way that the impact of O_b is effectively excluded from the parameters of the output operation O'_a. A function *Transpose* (O_a, O_b):(O'_b, O'_a) can be defined to transpose O_a and O_b where $O_a \mapsto O_b$ to achieve $O'_b \mapsto O'_a$.

Function 3 *Transpose*(O_a, O_b): (O'_b, O'_a)
{ $O'_b := ET(O_b, O_a)$;
$O'_a := IT(O_a, O'_b)$;
return (O'_b, O'_a);
}

The proposed *CALOM* (Compress A Log by Operational Merging) control algorithm is described as follows.

Algorithm 1 *CALOM (L):* L_{com}
{ $Compress\_Type(L, DEL, L_{com})$;
$Compress\_Type(L, INS, L_{com})$;
return L_{com};
}

Procedure 1 *Compress_Type(L, T,* L_{com}*) // T=DEL/INS*
{ **while** (($L[i]$=Last type T operation in L) $\neq null$)
{ $L_i = L[i]$;
for ($j = i - 1$; $j \geq 0$; $j - -$)
{ $L_j = L[j]$;
$OM(L_j, L_i) = (L'_j, L'_i)$;
if ($L'_j = L'_j = I$)
{ remove L_i and L_j; **break;**}
else if ($L'_j = I$)

{ $L_i = L'_i$; remove L_j;}
else if ($L'_i = I$)
{ $L_j = L'_j$; remove L_i; **break;**}
else (L_i, L_j) = *Transpose*(L'_j,L'_i);
}
if($j < 0$) { put L_i into L_{com}; remove L_i; }
}
}

After applying the *CALOM* algorithm to a log L, the compressed L_{com} would look like: [D_1, $\cdots$, D_r, I_1, $\cdots$, I_s] where D_i ($1 \leq i \leq r$) are delete operations and I_j ($1 \leq j \leq s$) are insert operations. The reason why we process all delete operations before insert operations is a deletion operation executed between two insert operations could prevent the two insert operations from being adjacent. For example, consider a document contains character y. Three operations have been executed and stored in L = [O_1, O_2, O_3] where O_1 = *INS*[0, 1, x] to insert character x before y, O_2 = *DEL*[1, 1, y] to delete character y, and O_3 = *INS*[1, 1, z] to insert character z after x. Then the insert operations O_1 and O_3 could not be adjacent unless the delete operations O_2 has been executed beforehand.

So for the L = [O_1, $\cdots$, O_5] in Figure 3, L_{com} = *CALOM* (L) = [D_1, I_1] where $D_1 = DEL$[1, 2, *yz*] and $I_1 = INS$[1, 4, *c123*]. Because $D_1 \odot I_1$, L has been fully compressed to be L_{com}.

4. Important properties

After applying the CALOM algorithm on a log L, the compressed log L_{com} would be fully compressed and equivalent to L. L_{com} is fully compressed in the sense that all operations in L_{com} are disjointed. L_{com} is equivalent to L(denoted as $L_{com} \equiv L$) in the sense that executing operations in L_{com} on the initial document state S_0 should reach the same final document state as the one reached by executing operations in L on S_0.

Theorem 1 *Full Compression Property (FCP)*

Given log L, if $CALOM(L) = L_{com}$, then all operations in L_{com} are disjointed.

Proof:

1. When $|L| = 1$, $L_{com} = L$, there is only one operation in L_{com}, the theorem apparently holds.
2. Suppose the theorem holds when $|L| \leq m$. That is, given L = [O_1, $\cdots$, O_q] where $q \leq m$, if $L_{com} = CALOM(L)$ = [D_1, $\cdots$, D_r, I_1, $\cdots$, I_s] where D_i($1 \leq i \leq r$) is a delete operation and I_j($1 \leq j \leq s$) is an insert operation, and $r+s \leq q \leq m$, then all operations in L_{com} are disjointed.
3. When $|L| = m + 1$, L = [O_1, $\cdots$, O_m, O_{m+1}], $L_{com} = CALOM(L) = CALOM([O_1, \cdots, O_m, O_{m+1}]) = CALOM([CALOM([O_1, \cdots, O_m]), O_{m+1}]) = CALOM([D_1, \cdots, D_r, I_1, \cdots, I_s, O_{m+1}])$.
   (a) If O_{m+1} is an insert operation, then $D_i \odot O_{m+1}$ where $1 \leq i \leq r$. So $CALOM([D_1, \cdots, D_r, I_1, \cdots, I_s, O_{m+1}]) = [D_1, \cdots, D_r, CALOM([I_1, \cdots, I_s, O_{m+1}])]$.
      i. If O_{m+1} is disjointed with any I_j($1 \leq j \leq s$), the theorem holds because all operations in $L_{com} = CALOM(L) = [D_1, \cdots, D_r, I_1, \cdots, I_s, O_{m+1}]$ are disjointed.
      ii. For $\forall I_k$($1 \leq k \leq s$), if I_k ($\oplus/\ominus$) O_{m+1}, I_k can be merged into O_{m+1}, so $CALOM([I_1, \cdots, I_s, O_{m+1}]) = [I_1, \cdots, I_t, O'_{m+1}]$ where $t < s$ and $O'_{m+1} \odot \forall I_j(1 \leq j \leq t)$. Because $t < s$ and $s \leq m$, $t + 1 \leq m$, according to the assumption in 2, all operations in $CALOM([I_1, \cdots, I_t, O'_{m+1}])$ are disjointed. Therefore the theorem holds because all operations in $L_{com} = CALOM(L) = [D_1, \cdots, D_r, I_1, \cdots, I_t, O'_{m+1}]$ are disjointed.
   (b) If O_{m+1} is a delete operation.
      i. If O_{m+1} can be totally merged into [I_1, $\cdots$, I_s], then $L_{com} = CALOM([D_1, \cdots, D_r, I_1, \cdots, I_s, O_{m+1}]) = [D_1, \cdots, D_r, CALOM([I_1, \cdots, I_s, O_{m+1}])] = [D_1, \cdots, D_r, CALOM([I_1, \cdots, I_t])]$ where $t \leq s$. Because $t \leq m$, according to the assumption in 2, all operations in $CALOM([I_1, \cdots, I_t])$ are disjointed. So the theorem holds because all operations in $L_{com} = CALOM(L) = [D_1, \cdots, D_r, CLOM([I_1, \cdots, I_t])]$ are disjointed.
      ii. If O_{m+1} cannot be totally merged into [I_1, $\cdots$, I_s], then $L_{com} = CALOM([D_1, \cdots, D_r, I_1, \cdots, I_s, O_{m+1}]) = [O'_{m+1}, CALOM([D_1, \cdots, D_l]), CALOM([I_1, \cdots, I_t])]$ where $l \leq r$ and $t \leq s$. Because $l \leq m$ and $t \leq m$, according to the assumption in 2, operations in $CALOM([D_1, \cdots, D_l])$ and $CALOM([I_1, \cdots, I_t])]$ are disjointed. As a result the theorem holds because all operations in $L_{com} = CALOM(L) = [O'_{m+1}, CALOM([D_1, \cdots, D_r]), CALOM([I_1, \cdots, I_t])]$ are disjointed.

Theorem 2 *Equivalence Property (EP)*

Given log L, if $CALOM(L) = L_{com}$, then $L_{com} \equiv L$.

Proof: Given a document state S and a sequence of operations executed on S and stored in L, we need to prove that $S \circ L_{com} = S \circ L$. According to the definition of the *CALOM* algorithm, it is needed to prove *OM* merging functions and *Transpose* function maintain the equivalence. It can be found in [9] that *Transpose* function can maintain

the equivalence. Now we prove OM merging functions can maintain the equivalence. Formally speaking, given any two operation O_a and O_b, where $O_a \mapsto O_b$ and $CT_{O_a} = S$. it must be proven that $S \circ [OM(O_a, O_b)] = S \circ [O_a, O_b]$.

Starting with the $OM\_ID(O_a, O_b)$ function where $S = C_1 \cdots C_n$, $O_a = INS[i, m, X_1 \cdots X_m]$, and $O_b = DEL[k, l, Y_1 \cdots Y_l]$ where $Y_i(1 \leq i \leq l) = C_j(1 \leq j \leq n)$ or $X_r(1 \leq r \leq m)$.

1. If $O_a \odot O_b$, then $OM\_ID(O_a, O_b) = (O_a, O_b)$. Therefore $S \circ [OM\_ID(O_a, O_b)] = S \circ [O_a, O_b]$.
2. If $O_a \oplus O_b$, for example, $i < k < i+m$ and $k+l > i+m$, then $Y_i(1 \leq i \leq l) = X_r(1 < r \leq m)$ or $C_j(i+1 \leq j \leq n)$. So $OM\_ID(O_a, O_b) = (O'_a, O'_b)$ where $O'_a = INS[i, k-i, X_1 \cdots X_{k-i}]$ and $O'_b = DEL[k, k+l-i-m, C_{i+1} \cdots C_{k+l-m}]$.
   (a) $S \circ O_a = S_a = C_1 \cdots C_i X_1 \cdots X_m C_{i+1} \cdots C_n$, $S \circ [O_a, O_b] = S_a \circ O_b = C_1 \cdots C_i \; X_1 \cdots X_{k-i} \; C_{k+l-m+1} \cdots C_n$.
   (b) $S \circ O'_a = S'_a = C_1 \cdots C_i X_1 \cdots X_{k-i} C_{i+1} \cdots C_n$, $S \circ [OM\_ID(O_a, O_b)] = S'_a \circ O'_b = C_1 \cdots C_i \; X_1 \cdots X_{k-i} \; C_{k+l-m+1} \cdots C_n$.
   (c) Therefore $S \circ [OM\_ID(O_a, O_b)] = S \circ [O_a, O_b]$.
3. Similar deductive method can be applied to prove the theorem holds for other overlapping cases in the OM_ID function, the OM_II function, and the OM_DD function. Therefore $S \circ [OM(O_a, O_b)] = S \circ [O_a, O_b]$.

The FCP property guarantees the completeness of the *CALOM* algorithm, which means a log has been minimized. The EP property guarantees the correctness of the *CALOM* algorithm, which means the effect of the compressed log is the same as that of the original log.

5. Conclusions

In this paper, we propose the technique of operational merging in order to reduce the size of a log and the number of operations within the log. In operational merging, same type of two overlapping/adjacent operations are merged to be one operation that combines their effects and different types of two overlapping operations can be merged by eliminating their overlapping part.

We further contribute a novel compression control algorithm *CALOM* by virtue of the techniques of operational merging and operational transformation. This algorithm exhausts all merging opportunities within a log to achieve full compression. The proposed algorithm has been proven both correct and complete in the sense that the compressed log has the same effect as the original one and operations that can be merged have already been merged.

The effect of the compression would be more significant when operations in a log are more localized. In this case, operations may have more chance to be overlapping/adjacent. In reality, during developing a software project, each software developer tends to localize his operations in constructing or debugging a module. Therefore the algorithm should be effective in keeping a log small in practice. We are now collecting experimental data in order to conduct quantitative statistics study on the algorithm.

Our next-step work aims to integrate non-real-time version control and real-time collaborative edit in one system where switch between them is smooth. In the further, we are going to investigate how to transparently transform state-based version control system repositories into operation-based ones.

References

[1] B. Berliner. CVS II:Parallelizing software development, *Proceedings of USENIX,* 1990.

[2] W. Courington. The Network Software Environment, *Sun Technical Report FE197-0,* Sun Microsystems Inc, February 1989.

[3] W. K. Edwards et al. Designing and Implementing Asynchronous Collaborative Applications with Bayou, *Proceedings of ACM Sympisium on User Interface Software anf Technology,* 1997.

[4] E. Lippe and N. V. Oosterom. Operation-based merging, *Proceedings of the Fifth ACM SIGSOFT Symposium on Software development environments,* 1992.

[5] W. Miller and E. W. Myers. A file comparison program, *Software - Practice and Experience,* 15(1): 1025-1040, 1985.

[6] J. P. Munson and P. Dewan. A Flexible Object Merging Framework, *Proceedings of the conference on Computer-supported cooperative work,* 1994.

[7] E. Myers: An O(ND) difference algorithm and its variations, *Algorithmica,* 1(2): 251-266, 1986.

[8] H. Shen and C. Sun. Operation-based revision control systems, *Proceedings of the 3rd Annual International Workshop on Collaborative Editing Systems in ACM Group Conference,* 2001.

[9] C. Sun et al. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems, *ACM Transaction on Computer Human Interaction,* 5(1), 63-108, 1998.

[10] W. F. Ticky. RCS – a system for version control, *Software Practice and Experience,* 15(7): 637-654, 1985.