Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems

Claudia Ignat
Institute for Information Systems
ETH Zurich
CH-8092, Switzerland
ignat@inf.ethz.ch

Moira Norrie
Institute for Information Systems
ETH Zurich
CH-8092, Switzerland
norrie@inf.ethz.ch

ABSTRACT

Operational transformation has been identified as an appropriate approach for consistency maintenance in real-time collaborative editing systems. Various operational transformation algorithms [2,10,16,15,12,19] can be applied only for applications that use a linear representation of the document. We propose a new algorithm called treeOPT (tree OPerational Transformation) that relies on a tree representation of the document. Applications using this algorithm achieve better efficiency, the possibility of working at different granularity levels and improvements in the semantic consistency. Our algorithm applies the same basic mechanisms as the existing operational transformation algorithms recursively over the different document levels.

Keywords
Collaborative editor, consistency maintenance, operational transformation, document models

INTRODUCTION

Real-time collaborative editing systems are groupware systems that allow members of a team to simultaneously edit shared documents from different sites. Shared objects involved in the team activity are subject to concurrent accesses and real-time constraints.

The real-time aspect necessitates that each user sees the effects of their own actions immediately and those of other users as soon as possible. To ensure high responsiveness, a replicated architecture where users work on copies of the shared document and instructions are exchanged by message passing is necessary.

High concurrency is also an important requirement for real-time collaborative editing systems, i.e. any number of users should be able to concurrently edit any part of the shared document.

Approaches such as turn-taking protocols, locking or serialization-based protocols fail to meet at least one of these requirements. Turn-taking protocols [3] allow only one active participant at a time, the one who “has the floor”; this approach is equivalent to document locking, in this way lacking concurrency. Locking [4] guarantees that users access objects in the shared workspace one at a time. Concurrent editing is allowed only if users are locking and editing different objects. Non-optimistic locking introduces delays for acquiring the lock. Optimistic locking avoids the delays, but it is not clear what to do when locks are denied and the object optimistically manipulated by the user must be restored to its original state. In the case of serialization-based protocols, operations are executed in the same total order at all sites. Non-optimistic serialization delays the execution of an operation until all totally preceding operations have been executed [7]. Optimistic serialization executes the operations upon their arrival, but use undo/redo techniques to repair the out-of-order execution effect [6].

The operational transformation approach for maintaining consistency of the copies of the shared document allows local operations to be executed immediately after their generation and remote operations need to be transformed against the other operations. The transformations are performed in such a manner that the intentions of the users are preserved and at the end the copies of the documents converge. Various operational transformation algorithms have been proposed: dOPT[2], adOPTed[10], GOT[16], GOTO[15], SOCT2[12,13], SOCT3 and SOCT4 [19]. Although these algorithms are generic operational transformation algorithms, they can be applied only for applications that use a linear representation of the document. The real-time collaborative text editors relying on these algorithms represent the document as a sequence of characters. In this paper, we propose a new algorithm called treeOPT (tree OPerational Transformation) that relies on a tree representation of the document. Applications using this algorithm achieve better efficiency, the possibility of working at different granularity levels and improvements in the semantic consistency relative to other existing operational transformation algorithms.

Our algorithm relies on the same principles for consistency maintenance as the GOT algorithm, thereby we begin our paper by giving a short overview of the GOT algorithm.

We then go on by presenting our algorithm highlighting its advantages over other existing algorithms. Next, a section dedicated to related work is presented. Concluding remarks and the main directions of our future work are presented in the last section.

CONSISTENCY MODEL

Our algorithm follows the same principles for consistency maintenance as presented in [16]. Therefore, in this section we give a brief overview of the consistency model underlying the GOT algorithm.

We start by defining the notions of causal ordering relations and dependent and independent operations.

Causal ordering relation "$\rightarrow$"
Given two operations O_a and O_b generated at sites i and j respectively then O_a is causally ordered before O_b, denoted $O_a \rightarrow O_b$ iff: (1) $i=j$ and the generation of O_a happened before the generation of O_b; or (2) $i \neq j$ and the execution of O_a at site j happened before the generation of O_b; or (3) there exists an operation O_x such that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$.

Dependent and independent operations
Given any two operations O_a and O_b, (1) O_b is *dependent* on O_a iff $O_a \rightarrow O_b$; (2) O_a and O_b are said to be independent or concurrent iff neither $O_a \rightarrow O_b$, nor $O_b \rightarrow O_a$. This is denoted $O_a || O_b$.

The GOT algorithm proposes a consistency model comprising the following consistency properties:

1. The *convergence* property requires that all copies of the same document are identical after executing the same collection of operations.
2. The *causality preservation* property requires that, for any pair of operations O_a and O_b, if $O_a \rightarrow O_b$, then O_a is executed before O_b at all sites.
3. The *intention preservation* property requires that, for any operation O, the effects of executing O at all sites are the same as the intention of O and the effect of executing O does not change the effects of independent operations.

To achieve causality-preservation a timestamping scheme based on a data structure called a State Vector (SV) is used [2]. With the aid of this vector, the conditions for execution of an operation at a certain site (causally-ready operation) are defined.

To achieve convergence, a total ordering relation "$\Rightarrow$" between operations is defined. Based on this total ordering, and on the *history buffer(HB)* with executed operations maintained at each site, an undo/do/redo scheme is defined:

(1) Undo operations in HB which totally follow O_{new} to restore the document to the state before their execution;
(2) Do O_{new};
(3) Redo all operations that were undone from *HB*.

To achieve intention preservation, a causally ready operation has to be transformed before its execution in order to cope with the modifications performed by other executed operations. Two types of transformations are defined: inclusion transformation and exclusion transformation. An *inclusion transformation* of an operation O_a against an independent operation O_b, denoted $IT(O_a,O_b)$, transforms O_a such that the impact of O_b is included in O_a. An *exclusion transformation* of an operation O_a against a causally-preceding operation O_b, denoted $ET(O_a,O_b)$, transforms O_a such that the impact of O_b is excluded from O_a.

Also, some functions are defined for including a list of operations OL into the context of operation O, i.e. $LIT(O,OL)$ and for excluding a list of operations OL from the context of operation O, i.e. $LET(O,OL)$.

THE treeOPT ALGORITHM

The real-time collaborative editors relying on existing operational transformation algorithms for consistency maintenance use a linear representation for the document. For example, a text document is viewed as a sequence of characters. This way of representation has several crucial disadvantages, which we present below.

All existing operational transformation algorithms keep a history of already executed operations in order to compute the proper execution form of new operations. Given the linear representation of the document, all past operations are stored in a single common buffer. When a new remote operation is received, it must be transformed against all of the operations in the history buffer, even though different users might work on completely different sections of the document, not interfering with each other. Keeping the history of all operations in a single buffer decreases the efficiency. The existing algorithms for integrating a new causally ready operation into the history have a complexity of order n^2, where n is the size of the examined history buffer (for example GOT, SOCT2, SOCT3). The dOPT algorithm has a complexity of order n, but copies convergence is not always achieved. Consequently, a long history results in a higher complexity. This complexity negatively affects the response time, i.e. the time necessary for the operations of one user to be propagated to the other users, which is a factor of critical importance in real-time systems.

Finally, although the existing algorithms solve the syntactic inconsistency problems, they do not enforce semantic consistency.

Let us consider that a shared document contains the text: "*Helo everybody!*". Suppose one user, noticing the misspelling of the word "*Hello*", tries to insert the letter "*l*", aiming to obtain the text: "*Hello everybody!*". At the same time, another user, noticing also the incorrect spelling of the word "*Helo*", deletes it completely, inserting instead the word "*Bye*", in order to obtain the following final result: "*Bye everybody!*". Unfortunately, there is no automatic way to execute these conflicting operations and obtain a semantically consistent result. The best that the algorithms (for example GOT) can obtain is the following: *"Byel everybody!"*.

In natural language, a character does not have a semantic value associated with it. But what is the elementary semantic unit in the case of natural language? Is it the word? Let us see what happens if all operations delete or insert whole words. Consider that the shared document contains the following text: "*The child go alone to school.*". Assume that a user deletes the word "*go*" and inserts "*goes*" instead, intending to obtain: "*The child goes alone to school.*". Simultaneously, another user inserts the word *"can"*, changing the text into: "*The child can go alone to school.*". Unfortunately, after each user receives the operations performed by the other one, the result is: "*The child can goes alone to school.*".

As we can see, even though all operations were operations involving whole words, semantic consistency could not be enforced. The conclusion we can draw is that working at any level of granularity can result in semantic inconsistencies, but working at a higher level usually translates into a more semantically consistent final result. However, semantic consistency remains an open issue that should also be tackled by consistency maintenance algorithms.

We propose a new algorithm overcoming the disadvantages presented above. The algorithm relies on operational transformation, its novelty consisting in modelling the document using a hierarchical structure. We present the algorithm applied to a text document, but it can be easily adapted for any other document that uses a hierarchical structure, the only difference being the number of levels in the hierarchy. In the case of text documents, the hierarchical structure has several levels of granularity: document, paragraph, sentence, word and character (see Fig. 1.), corresponding to the common syntactic elements used in natural language. In order to process these elements in a uniform manner when presenting the algorithm, we assign numeric values for each granularity level, as follows: for document level the value 0, for paragraph level the value 1, for sentence level the value 2, for word level the value 3 and for subword (character) level the value 4.

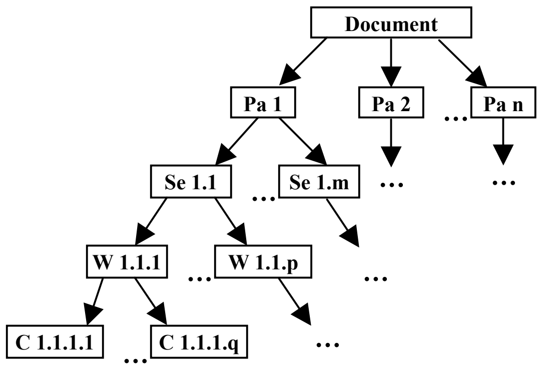

Fig. 1. Tree representation of the document

In order to describe the algorithm we first formally define the notions of node and composite operation.

Definition *Node*
A node N is a structure of the form N=<*level*, *children*, *length*, *history*, *content*>, where
- *level* is a granularity level, $level \in \{0,1,2,3,4\}$, corresponding to the element type represented by node (i.e. document, paragraph, sentence, word or character)
- *children* is an ordered list of nodes $\{child_1,...,child_n\}$, $level(child_i)=level+1$, for all $i \in \{1,...,n\}$
- *length* is the length of the node,

$$length = \begin{cases} 1, & \text{if } level = 4 \\ \sum_{i=1}^{n} length(child_i), & \text{otherwise} \end{cases}$$

- *history* is an ordered list of already executed *operations* on children nodes
- *content* is the content of the node, defined only for leaf nodes

$$content = \begin{cases} \text{undefined}, & \text{if } level < 4 \\ aCharacter, & \text{if } level = 4 \end{cases}$$

Please note that *operations* are equivalent to those defined by the model used in the GOT algorithm.

Definition *Composite Operation*
A composite operation is a structure of the form *cOp*=<*level*, *type*, *position*, *content*, *stateVector*, *initiator*>, where:
- *level* is a granularity level, $level \in \{1,2,3,4\}$
- *type* is the type of the operation, $type \in \{Insertion, Deletion\}$
- *position* is a vector of positions
  position[i]= position for the ith granularity level, $i \in \{1,...,level\}$
- *content* is a node representing the content of the operation
- *stateVector* - the state vector of the generating site
- *initiator* - the initiator site identifier

The *level* of a composite operation can be equal to 1, 2, 3 or 4, but not 0 (deleting the whole document or inserting a whole new document are not permitted). The vector *position* specifies the positions for the levels corresponding to a coarser or equal granularity than the granularity of the operation. For example, if we have an insertion operation of word level (3), we have to specify the paragraph and the sentence in which the word is located, as well as the position of the word inside the sentence. The *content* of a composite insertion operation specifies the node to be inserted in the position given by the *position* vector. In the case of deletion, *content* can be used to keep undo information. The attributes *stateVector* and *initiator* have the same meaning as in the case of the operations used by the GOT algorithm.

In future examples, for simplicity, we might also denote operations by specifying their *type*, *level*, *position* and sometimes the text conversion of *content*, ignoring the other attributes. For example *InsertWord(3,1,2,"love")* denotes a composite operation of *type Insertion*, having the *level word*, in paragraph 3, sentence 1, at word position 2 inside the sentence, and having as *content* a node of type *word* which stands for the text "*love*".

In what follows we will give an intuitive explanation of the algorithm, and afterwards describe it formally.

Each site stores locally a copy of the shared document. The document will be stored as a *hierarchical structure* (a tree). The root node of the tree will be the document node, having as children paragraph nodes. Each paragraph node, in its turn, will have as children sentence nodes, and so on. The leaf nodes will be character nodes. For a leaf node, the content of the node is explicitly specified in the *content* field. For nodes situated higher in the hierarchy, the *content* field will remain unspecified, but the actual content of each node will be the concatenation of the contents of its children. Each node (excluding leaf nodes) will keep a history of operations associated with its children. The operations stored in this history represent previous *insertions* or *deletions* of one of the *node's children*. An example showing the structure of a document is illustrated in Fig. 2.: the document contains three paragraphs; paragraph 3 contains two sentences; sentence 1 of paragraph 3 contains three words; 2nd word of sentence 1 in paragraph 3 is "CAR".

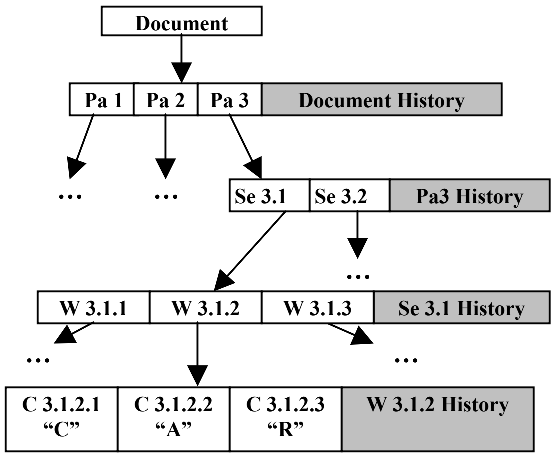

Fig. 2. Example of structure of a document

The algorithm follows similar principles to the ones described for the GOT algorithm. Each site can generate composite operations, representing insertions or deletions of subtrees in the document tree. Please note that each node of a subtree to be inserted has an *empty* history buffer. The site generating a composite operation executes it immediately. The operation is also recorded in the history buffer associated to the parent node of the inserted or deleted subtree. Finally, the new operation is broadcast to all other sites. The operation is timestamped using a state vector (similar to GOT algorithm). Upon receiving a remote operation, the receiving site will test it for causal readiness. If the composite operation is not causally ready it will be queued, otherwise transformed and then executed. Transforming the operation is somewhat more difficult (but also much more efficient) than in the case of the GOT algorithm. We will illustrate the way transformations are performed using an example.

Consider a site receiving the following remote composite operation: *InsertWord(3,1,3,"love")*. It is an operation intending to insert the word *"love"* in paragraph 3, sentence 1, as the 3rd word. The newly received operation must be transformed against the previous operations, as follows.

First of all, we consider the paragraph number specified by the composite operation, which in this case is equal to 3. We do not know for sure that paragraph number 3 of this site's local copy of the document is the same paragraph as that referred to by the original operation. Suppose a concurrent operation inserts a whole new paragraph before paragraph 3. Then, in this case, we should insert the word *"love"* not in paragraph 3, but in paragraph 4. Therefore, we must first transform the new operation against previous operations involving whole paragraphs, which are kept in the *document history* buffer. This can be done using any existing operational transformation algorithm working on linear structures (for example the GOT algorithm). After performing these transformations, we obtain the position of the paragraph in which the operation has to be performed, paragraph number 4 in our example. Consequently, the new composite operation will become *InsertWord(4,1,3,"love")*. Note that previous concurrent operations of finer granularity are not taken into account by these transformations, because the *document history* buffer contains only operations at paragraph level. Indeed, we are not interested in whether another user has just modified another paragraph, because this fact does not affect the number of the paragraph where the word *"love"* has to be inserted.

The next step obtains the correct number of the sentence where the word has to be inserted. Therefore, the new operation is transformed against the operations belonging to *Pa4 history*. *Pa4 history* only contains insertions and deletions of sentences that are children of paragraph 4. We again apply an existing operational transformation algorithm, and obtain the correct sentence position (for example sentence 2), transforming the operation into *InsertWord(4,2,3,"love")*. The algorithm continues by obtaining the correct word position in the same manner.

Finally, the operation can be executed and recorded in the history. Because it is an operation of word level, it must be recorded in the history associated with the parent sentence.

As we can see, the algorithm achieves consistency by applying repeatedly an existing concurrency control algorithm on small portions of the entire history of operations, which is not kept in a single linear structure, but distributed throughout the tree. However, traditional algorithms do not perform transformations on composite operations, but on regular ones. Because of this fact, we must define a function to transform a composite operation into a simple operation (by considering only the element of the *position* vector that is of interest at the moment).

```
Function Composite2Simple(cOp, level){
    construct a new operation op;
    if (level(cOp) = level)
        type(op) = type(cOp);
    else
        type(op) = Deletion;
    position(op) = position(cOp)[level(cOp)];
    length(op) = 1;
    stateVector(op) = stateVector(cOp);
    initiator(op) = initiator(cOp);
    return op;
}
```

In other words, having the composite operation *cOp*, and knowing the granularity level at which we are currently transforming the operation, we construct a regular one. This new operation will be transformed using an existing operational transformation algorithm. This regular operation will always have the length 1 (the algorithm only operates on one character, one word, one sentence or one paragraph at a time). In order to explain how we generate the *type* of the new operation, we will also use an example.

Suppose we have the composite operation *InsertWord(3,1,3,"love")*, and we are at the step when we need to compute the position of the paragraph in which the insertion takes place. Having this composite operation, we must generate a regular operation, which will then be transformed against the operations in *document history*, using an existing operational transformation algorithm. However, we only intend to find out the position of the paragraph where the word *"love"* has to be inserted, not to insert or delete a paragraph. Text editor systems relying on operational transformation work with the primitives *Insert* and *Delete*. Introducing a new *Modify* primitive is not a convenient solution, because new transformations need to be defined. A simpler solution is to simulate a delete, when in fact we do not intend to delete the paragraph, but merely to modify it. Therefore, if we work at a coarser granularity level than the level of the composite operation, we simulate a delete, and consequently we generate a regular delete operation. If, on the other hand, we work at the same granularity level as the level of the composite operation, we use the actual operation type in generating the regular operation.

After introducing this auxiliary function, we can present the general form of the algorithm.

```
Algorithm treeOPT(cOp, document, noLevels){
Given a new causally ready composite operation, cOp, a
local copy of the document, document and the number of
levels in the hierarchical structure of the document,
noLevels, the execution form of cOp is returned.
    currentNode = document;
    for (l = 1; l <= noLevels; l++)
        $o_{new}$ = Composite2Simple(cOp, l);
        $eo_{new}$ = Transform($o_{new}$, history(currentNode));
        position(cOp)[l] = position($eo_{new}$);
        if (level(cOp) = l)
            return cOp;
        currentNode = $child_i$(currentNode),
                              where i=position($eo_{new}$);
}
```

In the case of the text editor *noLevels*=4.

As we have seen in the previous examples, determining the execution form of a composite operation requires finding the elements of the *position* vector corresponding to a coarser or equal granularity level than the level of the composite operation. For each level of granularity *l* (starting with paragraph level and ending with the level of the composite operation) an existing operational transformation algorithm is applied for finding the execution form of the corresponding regular operation. The operational transformation algorithm is applied on the history of the *currentNode* whose granularity level is *l*-1 (recall that, for example, for finding the corresponding paragraph position, transformations need to be performed against the operations kept in the document history). The *l*th element in the *position* vector will be equal to the *position* of the execution form of the regular operation. If the current granularity level *l* equals with the level of the composite operation, the algorithm returns the execution form of the composite operation. Otherwise, the processing continues with the next finer granularity level, *currentNode* being updated accordingly.

The algorithm in its general form computes the execution form of the composite operation, but executing the operation and saving it in the history must be done after performing the algorithm.

By *Transform(op, history)* we denote any existing concurrency control algorithm, that, taking as parameters a causally-ready regular operation *op* and a history buffer *history*, returns the execution form of *op*. The implementation of the *Transform* method depends on the chosen consistency maintenance algorithm working on a linear structure of the document. We tested the operation of our algorithm when combined with the GOT algorithm. Because GOT must be used in combination with an undo/do/redo scheme, we have adapted our algorithm as well. Combining our algorithm with dOPT can be easily performed, requiring only that the *Transform* function be replaced with the part of dOPT algorithm for executing a causally ready operation [2]. When combined with SOCT2, the algorithm has to be adapted to the mechanism of integrating an operation into the history by performing forward and backward transpositions [12].

In the following, we present the recursive version of the algorithm, when combined with the *undo/do/redo* scheme and the GOT algorithm:

```
Algorithm treeOPT-GOT(node, cOp){
   $o_{new}$ = Composite2Simple(cOp, level(node)+1);
   Undo operations in history(node) from right to left until
         an operation $eo_m$ is found such that $eo_m$=>$o_{new}$;
   $eo_{new}$ = GOT($o_{new}$, history(node));
   position(cOp)[level(node)+1] = position($eo_{new}$);
   if (level(cOp) = level(node)+1)
      Do cOp;
      Update length(node);
      Store $eo_{new}$ in history(node);
      Transform each operation $eo_{m+i}$ in
                  history(node)[m+1,n] into the new
                  execution form $eo'_{m+i}$ as follows:
         $eo'_{m+1} = IT(eo_{m+1}, eo_{new})$
         for($i = 2$; $i \le n-m$; $i$++)
            $TO = LET(eo_{m+i},(history(node)[m+1,m+i-1])^{-1})$;
            $eo'_{m+i} = LIT(TO,[eo_{new},eo'_{m+1},...,eo'_{m+i-1}])$;
      Redo $eo'_{m+1}$, $eo'_{m+2}$,..., $eo'_n$ sequentially
   else
      if (length($eo_{new}$) > 0)
         child = $child_i$ (node), where i=position($eo_{new}$);
         treeOPT-GOT (child, cOp);
         Update length(node);
      Redo undone operations $eo_{m+1}$, ...,$eo_n$
}
```

The recursive algorithm computes the execution form of *cOp*, performs it, and updates the history of the proper node in the tree. The above algorithm must be called with the parameter *node* equal to the document node (root node in the document hierarchy).

The algorithm starts by transforming the composite operation *cOp* into a simple one corresponding to the level *level(node)+1*. This simple operation needs to be transformed against the other operations for the same level kept in *history(node)*. The same mechanism undo/do/redo as presented in [16] is used. By applying the GOT algorithm we can find the *position* element of the execution form of eo_{new}. The transformation functions used in the GOT algorithm can be relaxed, i.e. to be used not for strings, but just for characters. However, in order to take advantage of the power of the string-wise approach, we allow strings to be inserted/deleted in a word and we do not generate these operations character by character. For the paragraph, sentence and word level, the transformations are done in the same manner as in the character-wise approach. In this way eo_{new} cannot be an operation split into sub-operations in the case of paragraph, sentence and word level. It can be such an operation only at the subword level, the lowest level in the tree, but no further recursive calls need to be performed in this case.

Another important issue is related to performing the composite operation, after transforming it accordingly. Performing a composite operation implies inserting a subtree or deleting a subtree from the document hierarchical structure. Besides these actions, the nodes on the path taken by the algorithm have to update their lengths accordingly, increasing it by the length of the inserted subtree or decreasing it by the length of the deleted subtree.

An important advantage of the algorithm is related to its improved efficiency. In the case of existing algorithms, in order for a new operation to be transformed accordingly, a large history log, containing all previously executed operations, has to be spanned. This can be extremely inefficient, and the response time is usually affected. In our representation of the document, the history of operations is distributed throughout the whole tree, and, when a new operation is transformed, only the history distributed on a single path of the tree will be spanned (which, for a balanced tree, represents only *log(historylength)*). This will turn out to be a very important increase in speed, especially given the fact that the complexity of the concurrency control algorithm is usually of $O((spanned\_history)^2)$. Moreover, when working on medium or large documents, operations will be localized in the areas currently modified by each user, not interfering with each other at all. In these cases, almost no transformations are needed, and therefore the response times and notification times are very good (recall the fact that in the case of algorithms working on linear structures, every operation interferes with any other, independently of the distance between the positions specified in the operations).

Another important advantage is the possibility of performing, not only operations on characters, but also on other semantic units – words, sentences and paragraphs. An insertion or a deletion of a whole paragraph can be done in a single operation. Therefore, the efficiency is further increased, because there are fewer operations to be transformed, and fewer to be transformed against. Moreover, the data is sent using larger chunks, thus the network communication is more efficient.

Our approach also adds flexibility in using the editor. The users can select the level of granularity they prefer to work on. Some users prefer to wait until writing a whole paragraph, and only then to send an insert operation containing this paragraph, others prefer to send operations character by character, in order to enable the other users to visualize the modifications as soon as possible.

Last, but not least, our algorithm can help users in enforcing the semantic consistency of the documents, because working at a coarser granularity is allowed and cases such as the ones presented at the beginning of this section can be avoided.

RELATED WORK

Starting with the dOPT algorithm of Ellis and Gibbs [2] various algorithms using operational transformation for maintaining consistency in collaborative systems have been proposed: adOPTed[10], GOT[16], GOTO[15], SOCT2[12, 13], SOCT3 and SOCT4 [19]. All of these algorithms are based on a linear representation of the document. Our algorithm uses a tree representation of the document and applies the same basic mechanisms as these algorithms recursively over the different document levels.

To our knowledge, the only algorithm for solving the concurrency control problem in collaborative text editors that uses a tree model for representing the document is dARB[5]. However, according to this algorithm, not all concurrent accesses are automatically solved. When the system cannot find a solution, the users are asked to solve the inconsistencies manually. Their approach is similar to the dependency-detection approach for concurrency control in multiuser systems. Dependency detection [18] uses operation timestamps for detecting conflicting operations, the conflict being resolved by a process that involves human intervention. Some of the conflicts are resolved by the system invoking the arbitration procedure. According to the arbitration process, only the intentions of one user are preserved, which is not always the desired solution. For example, suppose that two sites have initially the same state: "*This work is part of the UIP (Universal Information Platform) project where multiuser suport is integrated.*". One user splits the sentence into two other sentences, aiming to obtain: "*This work is part of the UIP (Universal Information Platform) project. where multiuser suport is integrated.*". This is the first operation performed by the first user; later on s/he will try to modify the second sentence. The other user corrects the spelling of word "*suport*": "*This work is part of the UIP (Universal Information Platform) project, where multiuser support is integrated.*". According to dARB algorithm, in this case the arbitration scheme is invoked and because the operation performed by the first user (create operation) has a higher priority than the operation performed by the second user (modification), the first operation will win the arbitration, the final result being: "*This work is part of the UIP (Universal Information Platform) project. where multiuser suport is integrated.*". The second user can be frustrated that their modification was not preserved. In our approach we preserve the intentions of all users, even if in some cases the result is a strange combination of all intentions, such as in the example presented in the previous section. However, the use of different colours provides awareness of concurrent changes made by other users and the main thing is that no changes are lost.

Moreover, because operations (delete, insert) are defined only at the character level in this algorithm, i.e. sending only one character at a time, the number of communications through the network increases greatly.

Further, there are cases when one site wins the arbitration and it needs to send, not only the state of the vertex itself, but maybe also the state of the parent or grandparent of the vertex. Sending whole paragraphs or even the whole document in the case that a winning site has performed a split of a sentence or respectively a paragraph is not a desirable option.

In our approach, we tried to reduce the number of communications and transformations as much as possible, thereby reducing the notification time, which is a very important factor in groupware. For this purpose, our algorithm is not a character-wise algorithm, but a string-wise one and we do not need retransmissions of whole sentences, paragraphs or of the whole document in order to maintain the same tree structure of the document at all sites.

CONCLUSIONS AND FUTURE WORK

In this paper we have presented a consistency maintenance algorithm relying on a tree representation of the document. The hierarchical representation of a document is a generalisation of the linear representation and in this way our algorithm can be seen as extending the existing operational transformation algorithms. The algorithm applies the same basic mechanisms as the existing operational transformation algorithms recursively over the different document levels. When used by applications that rely on a hierarchical structure of the document, it achieves better efficiency, the possibility of working at different granularity levels and improvements in the semantic consistency. We have presented the implementation of the algorithm when used for a text document and based on the GOT algorithm, but it is application independent (i.e. it can be adapted for any tree-based model of a document) and it can use any operational transformation algorithm relying on linear representation.

The next step in our work is to adapt the algorithm for a graphical editor. We believe that the tree representation of the document will be suitable, taking into account the grouping features for the scene objects. However, as pointed out in [14], causality preservation techniques are applicable to both text editor and graphic editor, but achieving intention preservation and convergence is different in the two cases. We think that the next step in this direction would be to implement the multi-versioning technique used in GRACE [1] and see how ideas from the tree representation of the document can be incorporated.

We want to investigate the possibility of introducing locking at different granularity levels (paragraph, sentence, word). The novel locking mechanism described in [17] might be integrated.

Social aspects are also important for avoiding and resolving conflicts between users. Even though our system solves automatically conflicting operations generated by different users, the conflicts can also be solved more easily by mutual agreements among users. A great feature of our application is that conflicts are always solved, but they should be avoided to begin with. Features such as audio communication and chat systems among users might avoid many conflicts. Messages like "*I'm now working on the Introduction section.*" prevent other users from modifying the same part of the document. Users are aware of the modifications done by other users, since users write with different colours and the application provides a legend with the users editing the same document and the associated colours. Even in the case of conflicts, after an automatic solution was generated and it was not something expected, the users involved in the conflict can communicate to each other, finding out the intentions of the others and agreeing on a solution.

The text editor application that successfully implemented treeOPT [8] is part of the UIP (Universal Information Platform) project under current development in the Global Information Systems group at ETH Zürich. This project has the goal of designing and implementing a universal information platform supporting persistence, distribution and multi-user support [11], in particular with support for collaborative working. The whole functionality of the project relies on the generic object data model OM [9]. The text editor application can be enhanced with access rights and roles associated to users and groups of users, these features being already integrated into the core of the project.

REFERENCES

1. Chen, D. and Sun, C. A distributed algorithm for graphic object replication in real-time group editors. In *Proceedings of the ACM Conference on Supporting Group Work*, New York, 1999, pp. 121-130.
2. Ellis, C.A., and Gibbs, S.J. Concurrency control in groupware systems. *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1989, pp. 399-407.
3. Greenberg, S. Personalizable groupware: Accomodating individual roles and group differences. In *Proceedings of the European Conference on Computer Supported Cooperative Work*, Amsterdam, Sept. 1991, pp.17-32.
4. Greenberg, S., Marwood, D. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, North Carolina, October 1994, pp. 207-218.
5. Ionescu, M. and Marsic, I. An Arbitration Scheme for Concurrency Control in Distributed Groupware, in *The Second International Workshop on Collaborative Editing Systems*, December 2000.
6. Karsenty, A., and Beaudouin-Lafon, M. An algorithm for distributed groupware applications. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993, pp.195-202
7. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, Vol. 21, No. 7, July 1977, pp.558-565
8. Nedevschi, S. Concurrency control in real-time collaborative editing systems, *Diploma Thesis*, ETH Zürich, 2002.
9. Norrie, M.C. Distinguishing Typing and Classification in Object Data Models. In *Information Modelling and Knowledge Bases*, vol. VI, ch. 25. IOS, 1995.
10. Ressel, M., Nitsche-Ruhland, D. and Gunzenbauser, R. An integrating, transformation-oriented approach to concurrency control and undo in group editors, In *Proc. of ACM Conference on Computer Supported Cooperative Work*, Nov. 1996, pp. 288-297.
11. Rivera, G. From File Pathnames to File Objects: An approach to extending File System Functionality integrating Object-Oriented Database Concepts, *Doctoral Thesis* No. 14377, ETH Zurich, September 2001.
12. Suleiman, M., Cart, M. and Ferrié, J., Serialization of Concurrent Operations in Distributed Collaborative Environment. In *Proc. ACM Int. Conf. on Supporting Group Work (GROUP'97)*, Phoenix, November 1997, pp. 435-445.
13. Suleiman M., Cart M., Ferrié J. Concurrent Operations in a Distributed and Mobile Collaborative Environment, In *Proc.14th IEEE Int. Conf. on Data Engineering IEEE/ ICDE'98*, Orlando, February 1998, pp. 36-45.
14. Sun, C. and Chen, D. Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems. In *ACM Transactions on Computer-Human Interaction*, Vol.9, No.1, March 2002, pp. 1-41.
15. Sun, C. and Ellis, C. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In *Proc. ACM Int. Conf. On Computer Supported Cooperative Work (CSCW'98)*, Seattle, November 1998, pp. 59-68.
16. Sun, C., Jia, X., Zhang, Y., Yang, Y. and Chen, D. Achieving Convergence, Causality-preservation, and Intention-preservation in Real-time Cooperative Editing Systems. In *ACM. Trans. on Computer-Human Interaction* 5, 1(March) 1998, pp.63-108.
17. Sun, C., Sosic, R. Optional Locking Integrated with Operational Transformation in Distributed Real-Time Group Editors. In *Proc. of ACM 18th Symposium on Principles of Distributed Computing*, Atlanta, May 1999.
18. Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S. and Suchman, L. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, January 1987, Vol. 30, No.1, pp.32-47.
19. Vidot, N., Cart, M., Ferrié, J., and Suleiman M. Copies convergence in a distributed real-time collaborative environment, *CSCW'00*, Philadelphia, December 2000.