

# Customizable Collaborative Editor Relying on treeOPT Algorithm

Claudia-Lavinia Ignat and Moira C. Norrie

ETH Zurich, Switzerland

*ignat@inf.ethz.ch, norrie@inf.ethz.ch*

**Abstract.** Research in collaborative editing tends to have been undertaken in isolation rather than as part of a general information or application infrastructure. Our goal is to develop a universal information platform that can support collaboration in a range of application domains. Since not all user groups have the same conventions and not all tasks have the same requirements, this implies that it should be possible to customize the collaborative editor at the level of both communities and individual tasks. One of the keys to customization is to use a structured rather than linear representation of documents that can be applied to both textual and graphical editors. In this paper, we propose the treeOPT (tree OPERational Transformation) algorithm that, relying on a tree representation of documents, applies the operational transformation mechanism recursively over the different document levels. Applications using this algorithm achieve better efficiency, the possibility of working at different granularity levels and improvements in the semantic consistency.

## Introduction

Within the CSCW field, collaborative editing systems have been developed to support a group of people editing a document collaboratively over a computer network. These systems can be used in a wide range of advanced computing application areas, including collaborative writing, collaborative CAD (Computer Aided Design) and CASE (Computer Aided Software Engineering). The major benefits of collaborative editing include reduced task completion time and distributed collaboration. On the other hand, the challenges that it raises are

many, ranging from the technical challenges of maintaining consistency coupled with good performance to the social challenges of supporting group activities and conventions across many different communities.

Within the existing collaborative-editors' community, research tends to have been undertaken in isolation rather than as part of a general information or application infrastructure. Also, most of the research tends to be theoretical with limited implementation and studies of use in practice. Where applications have been considered, i.e. collaborative editing for a particular task such as writing scientific articles or music, the solutions often assume particular characteristics of both the users and the documents that they are editing. We consider that it is important to take into account all aspects of collaborative editing together, inclusive of theoretical foundations, technical aspects of implementation and issues of user interaction. Further, it is important to integrate it within a general information and application infrastructure so that it can support collaboration for a range of communities and activities within these communities. Since not all user groups have the same conventions and not all tasks have the same requirements, this implies that it should be possible to customize the collaborative editor at the level of both communities and individual tasks.

Most existing collaborative editors deal either with textual or graphical editing, using quite different document representations. In the case of textual editors, a linear representation is usually used. Our goal is to develop general textual and graphical collaborative editors that have a more structured representation that enables us to deal with consistency maintenance efficiently in both forms of editing, while offering the flexibility of customization of collaborative access.

In this paper, we propose the treeOPT (tree OPerational Transformation) algorithm that relies on a tree representation of the document. Our algorithm relies on the same principles for consistency maintenance as the GOT (Sun et al., 1998), and GOTO (Sun and Ellis, 1998) algorithms, but applies the same basic mechanisms recursively over the tree. 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.

We begin in the next section by motivating our choice of the operational transformation approach and giving a short overview of the consistency model on which our algorithm is based. We then present our algorithm in the following section, highlighting its advantages over other existing algorithms which rely on a linear structure representation. Next, we present some problems encountered when integrating the algorithm into the collaborative editor and the solutions we have adopted. Features of the customizable collaborative editor are presented in a separate section and this is followed by a section dedicated to a discussion of related work. Concluding remarks and the main directions of our future work are presented in the last section.

## Principles of consistency underlying the algorithm

Real-time operation is an important aspect to be considered in the design of collaborative editing systems as users should be able to see 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 essential requirement of 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 (Greenberg, 1991) allow only one active participant at a time, the one who “has the floor”; this approach is equivalent to document locking and lacks concurrency. Locking (Greenberg and Marwood, 1994) 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 (Lamport, 1977). Optimistic serialization executes the operations upon their arrival, but uses undo/redo techniques to repair the out-of-order execution effect (Karsenty and Beaudouin-Lafon, 1993).

The *operational transformation* approach has been identified as an appropriate approach for maintaining consistency of the copies of the shared document in real-time collaborative editing systems. It 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 (Ellis and Gibbs, 1989), adOPTed (Ressel et al., 1996), GOT (Sun et al., 1998), GOTO (Sun and Ellis, 1998), SOCT2 (Suleiman et al., 1997; Suleiman et al., 1998), SOCT3 and SOCT4 (Vidot et al., 2000). 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.

We therefore base our work on the operational transformation approach. Specifically, our algorithm follows the same principles for consistency

maintenance as presented in Sun et al. (1998). In the remainder of this section, we give a brief overview of the consistency model underlying our 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  $O_a \parallel O_b$ .

The consistency model satisfies the following consistency properties:

- The *convergence* property requires that all copies of the same document are identical after executing the same collection of operations.
- 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.
- 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 satisfy the above consistency properties different algorithms follow different approaches.

To achieve *causality preservation*, most operational transformation algorithms (dOPT, adOPTed, GOT(O), SOCT2) use a timestamping scheme based on a data structure called a State Vector (Ellis and Gibbs, 1989). With the aid of this vector, the conditions for execution of an operation at a certain site (causally-ready operation) are defined.

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. In the GOT algorithm two types of transformations are defined: inclusion transformations and exclusion transformations. 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  $O_a$ . Additionally, in the GOTO algorithm, a *transpose* function is defined to change the execution order of two operations while respecting the user intentions. The dOPT algorithm uses *dOPT transformation*, the equivalent of inclusion transformation. To achieve intention preservation, the adOPTed algorithm uses an N-dimensional interaction model graph and the *L-transformation* based on the same principle as inclusion transformation. SOCT2 and SOCT3 algorithms use *forward transposition* (the

equivalent of inclusion transformation) and *backward transposition* with the same underlying ideas as the transpose operation.

To achieve *convergence* most of the algorithms (adOPTed, GOTO, SOCT2) require that two conditions C1 and C2 be satisfied by the transformation functions. Condition C1 guarantees that the operation resulting from the transformation operation of two concurrent operations will not depend on the order in which they are serialized. Condition C2 aims at making the transformation of an operation with a sequence of operations independent of the order of the operations in the sequence. The dOPT algorithm uses only C1, but it fails to ensure the convergence of copies in all cases. GOT imposes neither of these conditions, but requires a total ordering relation “ $\Rightarrow$ ” between operations and an undo/do/redo scheme. SOCT3 and SOCT4 require only C1 and replace C2 by a continuous global order of execution of operations.

## The treeOPT algorithm

Most real-time collaborative editors relying on existing operational transformation algorithms for consistency maintenance use a linear representation for the document, such as a sequence of characters in case of text documents. This way of representing documents has several crucial disadvantages, which we present below.

All existing operational transformation algorithms keep a single history of operations already executed in order to compute the proper execution form of new operations. When a new remote operation is received, the whole history needs to be scanned and transformations need to be performed, even though different users might work on completely different sections of the document and do not interfere 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). Exceptionally, the dOPT algorithm has a complexity of order  $n$ , but convergence of copies 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 editing systems.

Dourish (1996) classifies conflicts as either syntactic or semantic. Syntactic conflicts occur at the system infrastructure level, while semantic conflicts are inconsistencies from the perspective of the application domain. Therefore, in the case of a multi-user text editor, consistency from the users' perspective is often not the same as consistency from the system's. 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: “*The child*

*go alone to school.*”. Assume that a user adds the letters “e” and “s” at the end of the word “go” intending to obtain: “*The child goes alone to school.*”. At the same time, another user, deletes “go” and inserts “went” aiming to obtain: “*The child went alone to school.*”. Unfortunately, there is no automatic way to execute these conflicting operations and obtain a semantically consistent result. The best that the algorithms such as GOT(O) can obtain is the following: “*The child wentes alone to school.*”. The same kind of inconsistencies happen if operations insert or delete not letters as previously described, but whole words. Consider again the previous example. First user, would then delete the word “go” and insert the word “goes” in order to obtain: “*The child goes alone to school.*”. Suppose now that, simultaneously, the second 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 operational transformation algorithms.

We propose a new algorithm overcoming the disadvantages presented above. The algorithm relies on operational transformation and on modelling the document using a hierarchical rather than linear 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. In the case of text documents, we model the document as consisting of paragraphs, each paragraph consisting of sentences, each sentence consisting of words and each word consisting of letters. Therefore, the tree structure has the following levels of granularity together with their assigned numeric values: document (0), paragraph (1), sentence (2), word (3) and character (4), corresponding to the common syntactic elements used in natural language.

We are now in a position to formally present our algorithm and we begin by defining the basic notions of node and composite operation.

**Definition Node**

A node N is a structure of the form  $N = \langle level, children, length, history, content \rangle$ , 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  $\{child_1, \dots, child_n\}$ ,  
 $level(child_i) = level + 1$ , for all  $i \in \{1, \dots, n\}$
- *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

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

Note that *operations* are equivalent to those defined by the model used in the GOT(O) algorithm.

### **Definition Composite Operation**

A composite operation is a structure of the form

$cOp = \langle level, type, position, content, stateVector, initiator \rangle$ , 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  $i^{th}$  granularity level,  $i \in \{1, \dots, level\}$
- *content* is a node representing the content of the operation
- *stateVector* is the state vector of the generating site
- *initiator* is 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 that 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 within the sentence. The *content* of a composite insertion operation specifies the node to be inserted in the position given by the *position* vector. The attributes *stateVector* and *initiator* have the same meaning as in the case of the operations used by the GOT(O) algorithm.

For the sake of simplicity, in future examples, we will denote operations by specifying only their *type*, *level*, *position* and the text conversion of *content*, ignoring the other attributes. For example, *InsertWord(3,1,2, "ECSCW")* 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 "ECSCW".

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 *hierarchical structure* of the shared document. 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 *insertion* or *deletion* operations associated with its *children nodes*. An example showing the structure of a document is illustrated in Figure 1: the document contains three paragraphs; paragraph 3 contains two sentences; sentence 1 of paragraph 3 contains three words; 2<sup>nd</sup> word of sentence 1 in paragraph 3 is “ECSCW”.

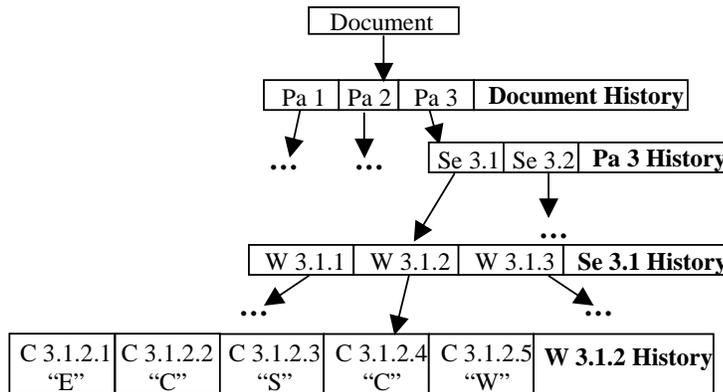


Figure 1. Example of structure of a document

The algorithm follows the same principles as those of the GOT(O) algorithm. Each site can generate composite operations, representing insertions or deletions of subtrees in the document tree. 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, being timestamped using a state vector. 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 it will be transformed and then executed. Transforming the operation is somewhat more difficult (but also much more efficient) than in the case of the GOT(O) algorithm. We will illustrate the way transformations are performed using an example.

Consider a site receiving the following remote composite operation: *InsertWord(3,1,2,“ECSCW”)*. It is an operation intending to insert the word “ECSCW” in paragraph 3, sentence 1, as the 2<sup>nd</sup> word. The newly received operation must be transformed against the previous operations, as described below.

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 “ECSCW” 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. Note that this could be done using any existing operational transformation algorithm working on linear structures such as the GOT(O) 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(4,1,2,“ECSCW”)*. Here it is important to 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 the 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 “ECSCW” 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(4,2,2,“ECSCW”)*. 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 repeatedly applying an existing concurrency control algorithm on small portions of the entire history of operations, which, rather than being kept in a single linear structure, is distributed throughout the tree.

We now present the general form of the treeOPT algorithm.

**Algorithm** *treeOPT(cOp, rootNode, noLevels)*{

Given a new causally ready composite operation, *cOp*, the root node of the hierarchical representation of the local copy of the document, *rootNode*, and the number of levels in the hierarchical structure of the document, *noLevels*, the execution form of *cOp* is returned.

```

currentNode = rootNode;
for (l = 1; l <= noLevels; l++)
    onew = Composite2Simple(cOp, l);
    eonew = Transform(onew, history(currentNode));
    position(cOp)[l] = position(eonew);

```

```

if (level(cOp) = l)
    return cOp;
currentNode = childi(currentNode), where i=position(eonew);
}

```

In the case of the text editor  $noLevels=4$  and  $rootNode=document$ .

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 that 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 to find the execution form of the corresponding regular operation. Traditional algorithms do not perform transformations on composite operations, but rather on regular ones. Therefore, we had to define the function *Composite2Simple*, that takes as arguments a composite operation, together with the granularity level at which we are currently transforming the operation, and returns 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, to find the corresponding paragraph position, transformations need to be performed against the operations kept in the document history). The  $l^{\text{th}}$  element in the *position* vector will be equal to the *position* of the execution form of the regular operation. If the current granularity level  $l$  is equal to 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, with *currentNode* being updated accordingly.

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 and adapted to the undo/do/re-do scheme of this algorithm. A detailed implementation of the *treeOPT-GOT* algorithm as well as of the *Composite2Simple* function can be found in (Ignat and Norrie, 2002). Combining our algorithm with dOPT can be easily performed. The transform function should be replaced with the part of dOPT algorithm for executing a causally ready operation (Ellis and Gibbs, 1989). 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 (Suleiman et al., 1998).

The *treeOPT* algorithm is a general algorithm in that it can be applied to any document having a hierarchical structure. A trivial application would be the case of a book modelled as being composed of chapters, with each chapter consisting

of sections, each section of paragraphs, each paragraph of sentences and so on. Another application is the case of XML documents. If we consider an XML document as being composed of elements without attributes, the algorithm is straightforward. In the case of elements with attributes, the treeOPT algorithm is still applicable, but an underlying algorithm for a linear structure dealing, not only with concurrent operations of insert and delete, but also with operations for modifying attributes needs to be implemented.

An important advantage of the algorithm is related to its improved efficiency. In our representation of the document, the history of operations is not kept in a single buffer, but rather 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. This will turn out to be a very important increase in speed, especially given the fact that the complexity of the concurrency control algorithms for a linear structure is usually of  $O(\text{spanned\_history}^2)$ . Moreover, when working on medium or large documents, operations will be localized in the areas currently modified by each individual user and these may often be non-overlapping. 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. The transformation functions used in the operational transformation mechanism are kept simple as in the case of character-wise transformations, not having the complexity of string-wise transformations. 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 being able to select the level of granularity they prefer to work on.

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.

## Adapting the algorithm to the collaborative editor

In this section we want to report on some problems we had when adapting the treeOPT-GOT algorithm for the text collaborative editor application and the solutions we have adopted to overcome these problems.

Even though the algorithm works very well with insert and delete primitives at different levels of the hierarchy, in practice these two primitives are not sufficient

to perform all possible operations. Actually this happens due to the introduction of the different hierarchic levels. Let us consider the following example. Suppose the second paragraph of a document consists of the following sentence: “*Nobody influences her like her brother.*” as shown in the Figure 2 (a). Suppose we want to split this sentence into two other sentences: “*Nobody influences her.* *She likes her brother.*” How can we perform this operation by using only insert and delete primitives? One alternative would be to first delete the words “*like*”, “*her*” and “*brother*”, from the first sentence, and then to insert the whole sentence: “*like her brother.*”. As a result of performing these operations, the new structure of Paragraph 2 will be the one illustrated in Figure 2(b).

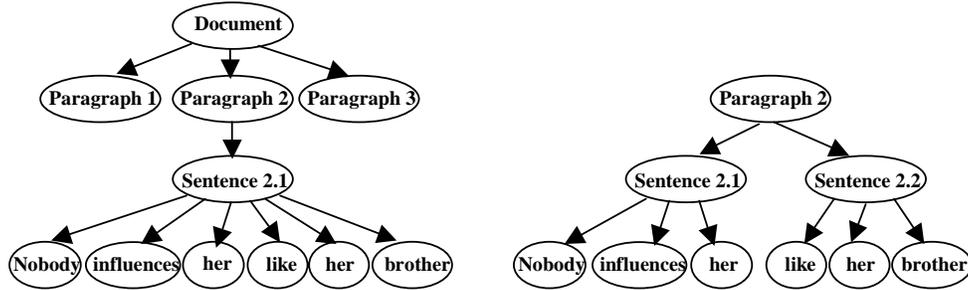


Figure 2. (a) Initial structure of the document; (b) Structure of the Paragraph 2 after splitting the Sentence 2

Unfortunately this approach does not work. Suppose that concurrently with the split operation of the sentence, another user, noticing the poor English, tries to insert the word “*does*” at the end of the sentence, in order to obtain “*Nobody influences her like her brother does.*” The operation sequence is illustrated in Figure 3(a). As we can see, by the time operation *InsertWord(2, 1, 7, “does”)* is received at Site1, the words “*like*”, “*her*” and “*brother*” are already deleted from the paragraph 2, sentence 1, and these operations of word deletion are kept in the history of Sentence 2.1. Applying the algorithm, the operation *InsertWord(2, 1, 7, “does”)* will be transformed into *InsertWord(2, 1, 4, “does”)*. The resulting structure of the paragraph, shown in Figure 3 (b), is not what the user at Site2 intended.

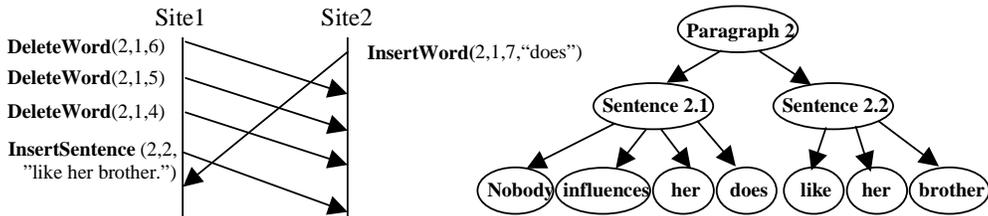


Figure 3. (a) Operation sequence (b) Erroneous result due to emulating split by using insertions and deletions

An unexpected result would be also obtained if the user at Site2 intends to change the word “brother” into “brothers”. The operation is performed on the Sentence 2.1, but the word “brother” has been already deleted from this sentence.

As we can see, splitting a sentence (or a word, or a paragraph) is not as simple as it seems at first sight. Some other possible ways of simulating the split operation using only insertions and deletions exist, but none of them is feasible. The reason is that a structural element might appear different on two hosts at the same time, and the two structures converge only because the history of operations on that element is kept at both sites. When an element is split into two parts, its history must be also split. Using only elementary insert/delete operations cannot detect the case when the history needs to be split or not. The same problem is encountered in the case of joining two elements. For example, if we delete a sentence separator, the two adjacent sentences will be joined into a single one implying the joining of the histories of the two sentences.

An alternative solution would be to introduce two other primitives: *split* and *join*, and to modify the algorithm by implementing operational transformation functions for these primitives as well. By means of an example, we show that this solution also does not work. Suppose that initially we have the sentence  $S_1 = \text{“He really enjoyed the movie.”}$  in both the local copies of users at Site1 and Site2 and the operation sequence is shown in Figure 4.

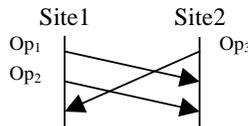


Figure 4. Counterexample for split and join primitives

Operation  $Op_1$  initiated by the user at Site1 splits sentence  $S_1$  in two sentences  $S_{11} = \text{“He really enjoyed.”}$  and  $S_{12} = \text{“the movie.”}$ . Operation  $Op_2$  deletes only sentence  $S_{12}$ . However, when operation  $Op_2$  arrives at Site 2, it cannot be directly executed, because it is independent of the operation  $Op_3$  (whatever  $Op_3$  is), and consequently must include the effect of  $Op_3$ . But before accomplishing this,  $Op_2$  has to have the same initial context as  $Op_3$ . That is why we need an exclusion transformation of  $Op_2$  against  $Op_1$ . If we exclude the effect of  $Op_1$ , that splits  $S_1$  into  $S_{11}$  and  $S_{12}$ , deleting the sentence “the movie.” will have to be transformed into two different operations, which are not of the same granularity:  $DeleteWord(\text{“the”})$ ,  $DeleteWord(\text{“movie”})$ . This is not an acceptable solution because the idea of the treeOPT algorithm is to transform operations against other operations of the same level of granularity, the result being also an operation of the same level of granularity.

The most appropriate solution we have so far found, although somehow disappointing, is not to split or join elements in the tree structure. For example, given the text “god father” composed of two words, deleting the space between

the two words, will still have as a result the two words, even though not separated by anything. The same approach can be adopted in the case of splits. If we insert a sentence separator, for instance a dot, or even a paragraph separator, for instance new line, inside a sentence, the text will be kept as a single sentence. Embracing this approach will however lead to degenerated elements. For example, the text “*Life is beautiful. So are you.*” might be a single large degenerated word, and the text “*deep*” can be stored in three degenerated sentences: “*d*”, “*ee*”, and “*p*”. Obviously, the hierarchical structure resulting in the case of degenerated elements is different from the one obtained by parsing the text and by delimiting the elements using their natural separators.

Even with this drawback, the algorithm works well. We afford degenerated elements because, when issuing an operation, the positions of the elements of different granularity levels (paragraph, sentence and word) are computed by taking into account the length of the previous elements. Consequently, the fact that the elements are degenerated does not matter. The efficiency of the algorithm remains unaffected by the degenerated elements, because the structure of the document will remain hierarchical, and operations will be transformed locally, spanning only a small part of the whole history of operations. Unfortunately, semantic consistency will be more difficult to maintain. It is harder to combine the algorithm with locking on a specific level of granularity. For example, trying to lock a word might result in locking a whole paragraph which is stored into a degenerated word. However, the problem is not as severe as it seems, because the reparsing of the whole document will be performed every time a new user begins the editing of the same document, or when the document is reloaded. Reparsing restores the semantic consistency of elements, only non-degenerated elements being generated. Reparsing the document should be enforced as often as possible. But, parsing the document implies having the same copy of the document at all sites. This means that reparsing can be performed only in moments of quiescence. Either the system can detect the moments of quiescence and initiate reparsing on copies of the document at all sites, or quiescence could be enforced by the system from time to time.

Analyzing the situation, the algorithm still keeps the anticipated efficiency boost, and also enhances the semantic consistency, even though not as well as expected.

We believe that the other approaches that apply operational transformation mechanisms for maintaining consistency in the case of a tree structure (Davis and Sun, 2002; Molli et al., 2002b) are also faced with the same problem of splitting or joining elements. However, nothing related to this problem is mentioned in their publications. Let us consider the example of an XML document modeling information about articles. Part of such a document is given below:

```
<article>
...
  <author> Grady Booch </author>
  <author> Ivar Jacobson </author>
</article>
```

If a user realizes that only first *author* is the author of the article and the second author is one of the authors of another article, then the article element may need to be split as follows:

```
<article>
...
  <author> Grady Booch </author>
</article>
<article>
...
  <author> Ivar Jacobson </author>
</article>
```

From the example presented above, we can see that the problem of splitting XML elements is a similar problem to that of splitting elements of different granularity levels in our approach.

## Customizable Collaborative Editor

It is desirable to have a collaborative editor that is customizable for not only application domains, but also activities and upon user request.

We have implemented a multi-document collaborative text editor relying on the treeOPT-GOT algorithm (Nedevschi, 2002). Users can join or leave the editing of any document or of the whole session whenever they want. Users can be made aware of the modifications done by other users, through the use of different colours and the application can provide a legend with the users editing the same document and the associated colours. Users are also informed by means of messages that appear on the lower part of the editor about the ongoing activity of the group, such as the joining of new users. 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.

In what follows we present some other functionality that we want to offer to the editor.

Optional locking as a complementary technique to operational transformation (Sun, 2002) can be used for maintaining the semantic consistency. Group undo (Sun, 2000) (global versus local, single step, chronological or selective) should be provided for error-recovery or alternative exploration. Members of a team might

wish to work independently in parallel, "insulated" in their private workspaces for some period of time (Molli et al., 2002a; Dourish, 1995). For example, when writing an entire new paragraph, some users might prefer to make it visible only after completing the work, or, in the case of an architectural design, it seems natural that an architect does not want to publish very sketchy initial drafts of their plans.

Social aspects such as audio communication or chat systems between users are also very important for avoiding or resolving conflict between users. Even though our system automatically resolves 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. Messages like "*I'm now working on the Introduction section.*" prevent other users from modifying the same part of the document. Even in the case of conflicts, after an automatic solution has been generated and perhaps produced something unexpected, the users involved in the conflict can communicate with each other, finding out the intentions of the others and agreeing on a solution. Different activities of users can be tracked (Chalmers, 2002). For example, for avoiding conflicts, it might be useful to have information about the frequency of modifications on different parts of the document performed in the last period of time. The scroller can change the intensity of its colour according to the number of modifications performed on that part of the document. Also, for each user, a chronological list of operations performed can be kept. Although almost all of the aspects mentioned above were researched into in isolation, open questions exist and we feel it is important to integrate and study many of these in the context of a single system.

Collaborative applications can offer a real improvement in the users' working activities if the underlying architecture of the implementation platform is able to provide required functionality already integrated at a lower-level, so that applications do not have to deal with aspects such as metadata handling, persistence, distribution and multi-user support. We therefore plan to integrate the collaborative editors into the Universal Information Platform (UIP) (Rivera, 2001), an object-oriented, multi-user, distributed, persistent information management system. For example, the collaborative editor application can easily be enhanced with access rights and roles associated to users and groups of users, as these features are already integrated into the core of the project.

## Related work

Starting with the dOPT algorithm of Ellis and Gibbs (1989), various algorithms using operational transformation for maintaining consistency in collaborative systems have been proposed: adOPTed (Ressel, 1996), GOT(Sun et al., 1998), GOTO (Sun and Ellis, 1998), SOCT2 (Suleiman et al., 1997), SOCT3

and SOCT4 (Vidot et al., 2000). As mentioned previously, all of these algorithms are based on a linear representation of the document whereas our algorithm uses a tree representation of the document and applies the same basic mechanisms as these algorithms recursively over the different document levels.

Other recent research has also looked at tree representations of documents. The dARB (Ionescu and Marsic, 2000) algorithm also uses a tree model for document representation, however it is not able to automatically resolve all concurrent accesses to documents and, in some cases, must resort to asking the users to manually resolve inconsistencies. Their approach is similar to the dependency detection approach for concurrency control in multi-user systems where operation timestamps are used to detect conflicting operations and the conflict is then resolved through human intervention (Stefik et al., 1987). The dARB algorithm may also use special arbitration procedures to automatically resolve inconsistencies; for example, using priorities to discard certain operations, thereby preserving the intentions of only one user. In our approach, we preserve the intentions of all users, even if, in some cases, the result is a strange combination of all intentions. 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 a paragraph, respectively, 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 an element-wise one, i.e. it performs insertions/deletions of different granularity levels (paragraphs, sentences, words and characters). Moreover, 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.

Some very recent publications have used operational transformation applied to documents written in dialects of SGML (Standard General Markup Language) such as XML and HTML (Davis and Sun, 2002; Molli et al., 2002b) or to CRC cards (Molli et al., 2002b). These are particular cases where a tree model of the document is required. These works were performed in parallel to our implementation of the editor relying on the treeOPT algorithm. However, their motivation for developing the algorithms differs from ours. They wanted to adapt operational transformation to XML-like structured documents and this required the transformation functions (Sun et al., 1998; Vidot et al., 2000) to be extended to

allow concurrent operations of insertion/deletion of elements and modification of the attributes of the elements in the XML structure. Our goal is to find a general and efficient algorithm for maintaining consistency applicable to all kinds of documents: raw, XML, graphical, etc. The hierarchical model seems a suitable model for a set of application domain documents. Therefore the treeOPT algorithm was designed and then implemented as the basis of a collaborative text editor where the hierarchical structure is not so obvious.

The existing collaborative graphical editors are based on one of three basic approaches to consistency maintenance, namely, locking - e.g. Aspects (von Biel, 1991), Ensemble (Newman-Wolfe et al., 1992), GroupDraw (Greenberg et al., 1992), serialization - e.g. GroupDesign (Karsenty and Beaudouin-Lafon, 1993), LICRA (Kanwati, 1992) or multi-version techniques - e.g. Tivoli (Moran et al., 1995), GRACE (Sun and Chen, 2002). The only work investigating a tree model for the graphical documents is (Ionescu and Marsic, 2000), but their approach, as previously mentioned, has the main disadvantage of not resolving inconsistencies automatically.

## Conclusions and future work

In this paper, we have presented a consistency maintenance algorithm relying on a tree representation of documents. 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 existing operational transformation algorithms recursively over the different document levels and it can use any of the operational transformation algorithms relying on linear representation. 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 algorithm focusing on its functionality for text documents, but it can also be applied for any kind of document relying on a hierarchical representation. We highlighted some key features of the customizable editor relying on the treeOPT algorithm and also discussed our plans to integrate the editor into the universal information platform UIP which provides a complete and rich API that could be used for the development of collaborative space for general document management.

In the future, we plan to investigate the possibility of introducing locking at different granularity levels (paragraph, sentence, word). We anticipate a set of challenges due to split and join operations as described in the paper and to the distributed locking conflict resolution protocol because of the peer-to-peer architecture of our application. Also, we want to develop a consistency maintenance algorithm for the case of a collaborative graphical editor relying on

some of the main concepts that were used for the text editor, such as the tree representation of the document. We also plan to perform some benchmarking and to evaluate the performance of the treeOPT algorithm in comparison with other existing algorithms.

## References

- Chalmers, M. (2002): 'Awareness, Representation and Interpretation', in *J. CSCW*, vol. 11, 2002, pp. 389-409.
- Davis, A.H. and Sun, C. (2002): 'Generalizing Operational Transformation to the Standard general Markup Language', *Proceedings of Conference on Computer Supported Cooperative Work*, 2002, pp. 58-67.
- Dourish, P. (1995): 'The Parting of the Ways: Divergence, Data Management and Collaborative Work', *Proc. Fourth European Conference on Computer-Supported Cooperative Work ECSCW'95*, Stockholm, Sweden, September 1995.
- Dourish, P. (1996): 'Consistency Guarantees: Exploiting Application Semantics for Consistency Management in a Collaboration Toolkit', *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'96*, 1996, pp. 268-277..
- Ellis, C.A., and Gibbs, S.J. (1989): 'Concurrency control in groupware systems', *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1989, pp. 399-407.
- Greenberg, S. (1991): 'Personalizable groupware: Accomodating individual roles and group differences', *Proceedings of the European Conference on Computer Supported Cooperative Work*, Amsterdam, September 1991, pp.17-32.
- Greenberg, S., Roseman, M., Webster, D. and Bohnet, R. (1992): 'Issues and experiences designing and implementing two group drawing tools', *Proceedings of the 25th Annual Hawaii International Conference on the System Science*, 1992, pp. 138-150.
- Greenberg, S. and Marwood, D. (1994): 'Real time groupware as a distributed system: Concurrency control and its effect on the interface', *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, North Carolina, October 1994, pp. 207-218.
- Ignat, C.L. and Norrie, M.C. (2002): 'Tree-based Model Algorithm for Maintaining Consistency in Real-Time Collaborative Editing Systems', *The Fourth International Workshop on Collaborative Editing Systems, CSCW 2002*, New Orleans, USA, November 2002.
- Ionescu, M. and Marsic, I. (2000): 'An Arbitration Scheme for Concurrency Control in Distributed Groupware', *The Second International Workshop on Collaborative Editing Systems, CSCW 2000*, December 2000.
- Kanwati, R. (1992) 'LICRA: a replicated-data management algorithm for distributed synchronous groupware application', in *Parallel Computing* 22, 1992, pp. 1733-1746.
- Karsenty, A., and Beaudouin-Lafon, M. (1993): 'An algorithm for distributed groupware applications', *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993, pp.195-202.
- Lamport, L. (1977): 'Time, clocks, and the ordering of events in a distributed system', in *Communication of the ACM*, vol. 21, no. 7, July 1977, pp.558-565.
- Molli, P., Skaf-Molli, H. and Oster, G. (2002a): 'Divergence Awareness for Virtual Team through the Web', in *Integrated Design and Process Technology, IDPT-2002*, Pasadena, CA, USA. Society for Design and Process Science, June 2002.

- Molli, P., Skaf-Molli, H., Oster, G. and Jourdain, S. (2002b): SAMS: 'Synchronous, asynchronous, multi-synchronous environments', *Proceedings of the Seventh International Conference on CSCW in Design*, Rio de Janeiro, Brazil, September 2002.
- Moran, T., McCall, K., van Melle, B., Pedersen, E. and Halasz, F. (1995): 'Some design principles for sharing in tivoli, a whiteboard meeting-support tool', in *Groupware for Real-Time Drawings: A designer's Guide*, S. Greenberg, Ed. McGraw-Hill International(UK), 1995, pp. 24-36.
- Nedevschi, S. (2002): 'Concurrency control in real-time collaborative editing systems', *Diploma Thesis*, ETH Zurich, 2002.
- Newman-Wolfe, R.E., Webb M., and Montes, M. (1992): 'Implicit locking in the Ensemble concurrent object-oriented graphics editor', *Proc. of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, New York, 1992, pp. 265-272.
- Ressel, M., Nitsche-Ruhland, D. and Gunzenbauser, R. (1996): 'An integrating, transformation-oriented approach to concurrency control and undo in group editors', *Proc. of ACM Conference on Computer Supported Cooperative Work*, November 1996, pp. 288-297.
- Rivera, G. (2001): '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.
- Suleiman, M., Cart, M. and Ferrié, J. (1997): 'Serialization of Concurrent Operations in Distributed Collaborative Environment', *Proc. ACM Int. Conf. on Supporting Group Work (GROUP'97)*, Phoenix, November 1997, pp. 435-445.
- Suleiman M., Cart M. and Ferrié J. (1998): 'Concurrent Operations in a Distributed and Mobile Collaborative Environment', *Proc. 14th IEEE Int. Conf. on Data Engineering IEEE/ICDE'98*, Orlando, February 1998, pp. 36-45.
- Sun, C. (2000): 'Undo any operation at any time in group editors', *Proceedings of ACM Conference on CSCW*, Philadelphia, USA, December 2000, pp. 191-200.
- Sun, C. (2002): 'Optional and Responsive Fine-grain Locking in Internet-based Collaborative Systems', in *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 9, September 2002, pp.994-1008.
- Sun, C. and Chen, D. (2002): '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.
- Sun, C. and Ellis, C. (1998): 'Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements', *Proc. ACM Int. Conf. On Computer Supported Cooperative Work (CSCW'98)*, Seattle, November 1998, pp. 59-68.
- Sun, C., Jia, X., Zhang, Y., Yang, Y. and Chen, D. (1998): 'Achieving Convergence, Causality-preservation, and Intention-preservation in Real-time Cooperative Editing Systems', in *ACM Trans. on Computer-Human Interaction*, vol. 5, no. 1, March 1998, pp.63-108.
- Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S. and Suchman, L. (1987): 'Beyond the chalkboard: Computer support for collaboration and problem solving in meetings', *Communications of the ACM*, vol. 30, no.1, January 1987, pp.32-47.
- Vidot, N., Cart, M., Ferrié, J., and Suleiman M. (2000): 'Copies convergence in a distributed real-time collaborative environment', *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, Philadelphia, USA, December 2000, pp.171-180.
- von Biel, V. (1991): 'Groupware Grows Up', in *MacUser*, June 1991, pp. 207-211.