Flexible Merging of Hierarchical Documents

Claudia-Lavinia Ignat and Moira C. Norrie
Institute for Information Systems, ETH Zurich
CH-8092 Switzerland
{ignat,norrie}@inf.ethz.ch

ABSTRACT

Existing versioning systems offer limited support concerning the definition and resolution of conflicts as well as tracking of user activity. In this paper we propose a flexible hierarchical-based merging approach based on operations, where the conflicts can be specified and resolved at different semantic units corresponding to the document levels. Our algorithm applies an existing operation-based merging approach for linear structures recursively over the different document levels. Our approach also achieves better efficiency compared to existing approaches for merging documents with linear structures.

Keywords

Asynchronous collaborative editing, version control systems, hierarchical documents merging, operational transformation

INTRODUCTION

Asynchronous collaborative editing systems have been developed to support a group of people collaboratively editing documents by allowing members of the group to modify copies of a document in isolation, working in parallel and afterwards synchronising their copies to reestablish a common view of the data.

Well known versioning systems such as CVS [1], RCS [13] and Subversion [3] offer limited support concerning conflict resolution and tracking of user activity. In these systems there is no flexible way of specifying the possible forms of conflict and merging is performed on a line by line basis with the basic unit of conflict therefore being the line. This means that the changes performed by two users are deemed to be in conflict if they refer to the same line. Concurrent changes on different lines are merged automatically. Therefore, these systems cannot handle multiple changes within a single line.

The above mentioned version control systems adopt *state-based merging* where only the information about the states of the documents and no information about the evolution of one state into another is used. The *operation-based merging* approach [7,11] keeps information about the evolution of one document state into another in a buffer containing a history of the operations performed between the two states of the document. The merging is done by executing the operations performed on a copy of the document onto the other copy of the document to be merged. In contrast to the state-based approach, the operation-based approach does not require that the documents are transferred over the network between the local workspaces and the repository. Moreover, merging based on operations achieves a better responsiveness of the system since no complex differentiation algorithms for text such as diff [10] or for XML [2] have to be applied in order to compute the delta between the documents. It also offers better support for conflict resolution by having the possibility of tracking user operations. When a conflict occurs, the operation causing the conflict is presented in the context in which it was originally performed. In contrast, in the state-based merging approach, the conflicts are presented in the order in which they occur within the final structure of the object. For instance, CVS, RCS and Subversion present the conflicts in the line order of the final document, the state of a line possibly incorporating the effect of more than one conflicting operation.

In this paper we propose a flexible operation-based merging algorithm that works on a hierarchical representation of documents, allowing the possibility of defining and resolving conflicts by using different semantic units corresponding to the document levels. Our approach is general for any document conforming to a hierarchical structure, such as XML documents. However, throughout the paper, for a simpler explanation of the approach, we will use text documents modeled as consisting of paragraphs, sentences, words and characters. The approach allows conflicts to be defined and resolved by using the semantic units - paragraphs, sentences, words and characters. For instance, a rule specifying that concurrent insertions in the same sentence are conflicting can easily be defined. The approach proposed in this paper has been implemented in our group as part of a framework for synchronous and asynchronous collaborative editing.

The paper is structured as follows. We begin by presenting an existing linear based merge algorithm that is applied recursively over the document levels by our tree-based merging approach. We then describe our approach for merging and relate our work to other existing approaches for merging. Concluding remarks are presented in the last section.

OPERATIONAL TRANSFORMATION LINEAR-BASED MERGING

In this section we first present the basic notions of the operational transformation mechanism when applied in the merging process. Afterwards, we present the FORCE algorithm [11] which uses operational transformation for the merging of linear representations of text documents.

Merging Operations

The basic operations supported by most configuration management tools are: checkout, commit and update. A *checkout* operation creates a local working copy of a document from the repository. A *commit* operation creates a new version of the corresponding document in the repository by validating the modifications done on the local copy of the document. The condition of performing this operation is that the repository does not contain a more recent version of the document to be committed than the local copy of the document. An *update* operation performs the merging of the local copy of the object with the last version of that object stored in the repository.

Operational Transformation Mechanism

Operation transformation has been used for maintaining consistency in real-time collaborative editing [4, 12, 14] as well as in asynchronous collaborative editing [8, 11].

Firstly, we present the notion of *context* of an operation O denoted as CT_O as being the document state on which O is defined. Two operations O_a and O_b having the same context, $CT_{Oa} = CT_{Ob}$, are denoted $O_a =_{CT} O_b$. An operation O_a is *context preceding* operation O_b denoted as $O_a \rightarrow_{CT} O_b$ if $CT_{Ob} = CT_{Oa} \circ O_a$, i.e. the state of the document on which O_b is defined is equal to the state of the document after the application of O_a.

The basic operations of the operational transformation mechanism are the inclusion and exclusion transformations. The *inclusion transformation - $IT(O_a,O_b)$* transforms operation O_a against operation O_b such that the effect of O_b is included in O_a. Consider the following scenario illustrated in Figure 1. Suppose the repository contains the document consisting of one sentence *"We present the merge."* and two users check-out this version of the document and perform some operations in their workspaces. Further, suppose $User_1$ performs the operation O_{11}=*InsertWord ("procedure",5)* intending to insert the word *"procedure"* at the end of the sentence, as the 5th word, in order to obtain *"We present the merge* ***procedure****."* Afterwards, $User_1$ commits the changes to the repository and the repository stores the list of operations performed by $User_1$ consisting of O_{11}. Concurrently, $User_2$ executes operation O_{21}=*InsertWord("next",2)* of inserting the word *"next"* as the 2nd word into the sentence in order to obtain *"We* ***next*** *present the merge."* Before performing a commit, $User_2$ needs to update the local copy of the document. The operation O_{11} stored in the repository needs to be transformed in order to include the effect of operation O_{21}. Because operation O_{21} inserts a word before the insertion position of O_{11}, O_{11} needs to increase its position of insertion by 1. In this way the transformed operation will become an insert operation of the word *"procedure"* as the 6th word, the result being *"We* ***next*** *present the merge* ***procedure****."* The condition of performing $IT(O_a,O_b)$ is that $O_a =_{CT} O_b$.

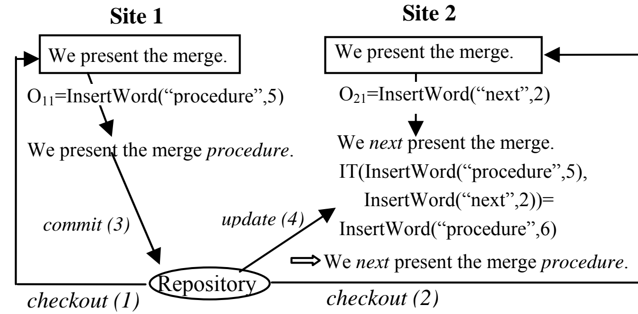


Figure 1. Inclusion Transformation

The *exclusion transformation - $ET(O_a,O_b)$* transforms O_a against the operation O_b that precedes O_a such that the impact of O_b is excluded from O_a.

Consider that a user performs some modifications in the local workspace starting from the version of the document consisting of the sentence *"Our proposed approach was implemented."* The user performs the operations O_1=*DeleteWord("proposed",2)* of deleting the 2nd word *"proposed"* from the sentence and O_2=*InsertWord("successfully",4)* of inserting the word *"successfully"* as the 4th word into the sentence resulting after the execution of O_1, as shown in Figure 2.

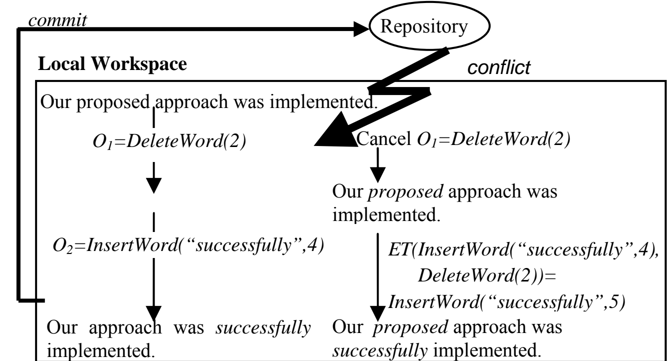


Figure 2. Exclusion Transformation

In the updating process, in the case that a local operation performed in the workspace is in conflict with an operation from the repository, one of the conflicting operations needs to be cancelled. The chosen conflict resolution policy decides which operation needs to be cancelled. Suppose that the operation O_1 is in conflict with an operation from the repository and it has to be cancelled. The operation O_2

needs to exclude the effect of operation O_1, i.e. to adapt its position as if operation O_1 had not been executed. By excluding the effect of O_1, the position of insertion of O_2 would become 5, the result being *"Our **proposed** approach was **successfully** implemented."* The condition of performing $ET(O_a,O_b)$ is that $O_b \rightarrow_{CT} O_a$.

Linear-based Merging

In this subsection we are going to present the principles of the FORCE [11] merging algorithm applied to a linear representation of the documents.

In the *commit* phase of merging, a commit is allowed to be performed only if the base version of the document in the local workspace, i.e. the last version from the repository that the user started working on, is equal to the last version in the repository. Otherwise, an update is necessary before committing the data. In the case that a commit is allowed to be performed, the latest version from the repository is replaced with the received operations from the local workspace and the new version of the document in the repository is obtained by sequentially executing the local operations on the last state of the document. Additionally, the corresponding base version number from the local workspace, as well as the latest version number from the repository, is increased and the local log from the local workspace is emptied.

In the *checkout* phase, a request is sent to the repository including the version number of the document that is intended to be checked out. In the case that the requested version number is larger than the latest version number in the repository, the repository sends a rejective reply. In the case that the requested version number equals the latest version number in the repository, the repository sends the full state of the last version of the document to the local workspace. In the case that the requested version number is less than the latest version number from the repository, the repository generates the state of the requested version by executing the inverses of the operations representing the deltas between the latest version in the repository and the requested version. In the case of a positive reply from the repository, the local site makes the sent document the working copy and sets the base version number to be equal to the version number of the document that was sent.

In the *updating* phase, the site sends to the repository the number of the base version. The repository sends to the site a list of operations representing the delta between the latest version in the repository and the base version in the local workspace. Upon receiving the list of operations from the repository, the local workspace performs the merging algorithm and updates the base version number. The merging algorithm has to be performed for the following scenario. The local user started working from version V_k on the repository but cannot commit the changes because meanwhile the version from the repository has been updated to version V_{k+n}. Let us denote by LL the list of operations executed by the user in the local workspace and by DL the list of operations representing the delta between versions V_{k+n} and V_k.

Two basic steps have to be performed. The first step consists of applying the operations from DL to the local copy of the user in order to update the local document to version V_{k+n}. The operations from the repository, however, cannot be executed in their original form as they have to be transformed in order to include the effect of all the local operations before they can be executed in the user workspace. The second step consists of transforming the operations in LL in order to include the effects of the operations in DL, the list of the transformed local operations representing the new delta into the repository.

In addition to the operational transformation algorithms that solve the syntactic inconsistency problems in collaborative text editing, the approach proposed in [11] introduces a semantic level of merging by the definition of a *semanticConflict* function that determines whether two concurrent operations are semantically conflicting.

From the list of operations in the list DL not all of them can be executed in the local workspace because some of these operations may be in conflict with some of the operations from LL. Let us consider that $DL=[O_{d1},..., O_{d(i-1)}, O_{di}, O_{d(i+1)}, ..., O_{dm}]$, where $O_{d1} \rightarrow_{CT} ... \rightarrow_{CT} O_{dm}$. In the case that O_{di} is in conflict with at least one operation from LL, O_{di} cannot be executed in the local workspace. Moreover, all operations following it in the list DL need to exclude its effect from their context. But, the condition to exclude an operation O_a from an operation O_b is that $O_a \rightarrow_{CT} O_b$. Therefore, in order to exclude the effect of operation O_{di} from the context of all the operations following it in the list DL, we need to transpose operation O_{di} towards the end of the list DL. As a result of this transposition the following condition should be fulfilled: $O_{d1} \rightarrow_{CT} O_{d2} \rightarrow_{CT} ... \rightarrow_{CT} O_{d(i-1)} \rightarrow_{CT} O_{d(i+1)} \rightarrow_{CT} ... \rightarrow_{CT} O_{dm} \rightarrow_{CT} O_{di}$.

The *Transpose* function that changes the execution order of the operations O_a and O_b and transforms them such that the same effect is obtained as if the operations were executed in their initial order and initial form is defined below. The condition of performing the Transpose function is that $O_a \rightarrow_{CT} O_b$ and after the call of $Transpose(O_a,O_b)$, $O'_b \rightarrow_{CT} O'_a$, where O'_b and O'_a are the transformed forms of O_b and O_a, respectively.

```
Transpose(Oa,Ob){
    O := ET(Ob,Oa);
    Ob := IT(Oa,O);
    Oa := O;
}
```

In order to combine the two steps of the merging, i.e. the transformations of the operations from the repository against the operations from the local log and the transformations of the operations from the local log against

the repository, the symmetric inclusion operation has been defined:

```
SymmetricInclusion(Oa,Ob){
    O := IT(Oa,Ob);
    Ob := IT(Ob,Oa);
    Oa := O;
}
```

The basic merge procedure takes as input arguments two logs, the remote log *RL* containing the operations from the repository and the local log *LL* containing the local operations and the base version number *V.bv* at the local site. The merge procedure generates as output two other logs, the new remote log *NRL* and the new local log *NLL*, logs that have been modified in order to include the effects of the operations in the other log. The new remote log *NRL* will contain the list of operations that should be executed sequentially on the current document state of the working copy in order to update it. It will contain the non conflicting operations from the original remote log, modified in order to include the effects of the operations in the local log. The new local log *NLL* will store the list of operations which has to be sent to the repository and represents the delta between the new version and the old version in the repository. It contains the operations in the local log transformed in order to include the effect of the operations in the remote log. Additionally, it might also include the inverse of the conflicting operations from the remote log. The implementation of the merge procedure is given below.

```
Algorithm merge(RL,LL,V.bv):(NRL,NLL){
 RCT:=V.bv;
 for(i=1;i≤|RL|;i++){
  CLL:=makeCopy(LL);
  CRLi:=makeCopy(RL[i]);
  LCT:=RCT;
  for(j=1;j≤|LL|;j++){
    if semanticConflict(SMR,RL[i],LL[j],LCT){
     LL:=CLL;
     RL[i]:=CRLi;
     O:=removeOperation(i,RL);
     i:=i-1;
     append(makeInverse(O),NLL);
     break;
    } else { LCT:=execute LL[j] on LCT;
             symmetricInclusion(RL[i],LL[j]);
     }
  }
  if(j>|LL|) { append(RL[i],NRL);
               RCT:=execute CRLi on RCT;
  }
 }
 append(LL,NLL);
 return (NRL, NLL);
}
```

In order to perform the correct transformations, the local and remote contexts need to be updated accordingly. Initially the remote context equals the base version of the document in the repository. For each operation in the remote log, a sequence of steps is performed. At the beginning of the iteration, a copy of the local log is saved in case the local log needs to be restored later. Also, a copy of the current operation from the remote log is saved for possible restoration later. All operations in the local log are iterated and a check for conflict between the local operation and the remote one is performed. Two cases are distinguished depending on the existence of conflict.

In the case that the two operations are not in conflict, the symmetric inclusion procedure is called in order to transform the remote operation against the local operation and vice-versa. The local context is updated in order to include the last local operation. If the remote operation is not in conflict with any of the local operations, by the end of the iteration over the local log, the remote operation will have orderly included the effect of each of the local operations and each of the local operations will have included its effect. Therefore, the remote operation is added at the end of the new remote log *NRL* and the remote context is updated in order to include the initial form of the remote operation. By the time all operations in the remote log have been iterated, each of the operations in the local log will have included the effect of each of the operations in the remote log. Therefore, the transformed operations from the local log can be added to the new local log *NLL*, as their context includes all operations from the repository.

In the case that the remote and local operations are in conflict, according to the resolution conflict policy adopted, one of these two operations is kept and the other one cancelled. In the merge procedure presented above, the local operation is chosen automatically as the winner of the conflict. Therefore, the remote operation should be eliminated from the remote log. The local log has to be restored to its form before some of the local operations included the effect of the remote operation to be removed. The remote operation needs to be reset to its original form before including the effect of the local operations up to the current conflicting local operation. Next, the *removeOperation* procedure has to be applied in order to successively transpose the remote operation to the end of the remote log. In this way, the remote operation includes the effect of the operations that follow it in the remote log and its inverse can be safely added to the beginning of the new local log *NLL*. The inverse operation simply cancels the effect of the original operation from the repository. Once the iterations are finished, the operations from the local log need to be added to the new local log.

MERGING OF HIERARCHICAL DOCUMENTS

In this section we present our merging approach working for hierarchical structures of the document as a generalisation of the merging applied for linear structures.

Model of the Document

We model a document as a hierarchical structure having the following levels of granularity: document (0),

paragraph (1), sentence (2), word (3) and character (4), document being the highest granularity level and character being the lowest granularity level. Each workspace stores locally a copy of the tree structure of the document. Each node (excluding leaf nodes) will keep a history of insertion or deletion operations associated with its children nodes as illustrated in Figure 3.

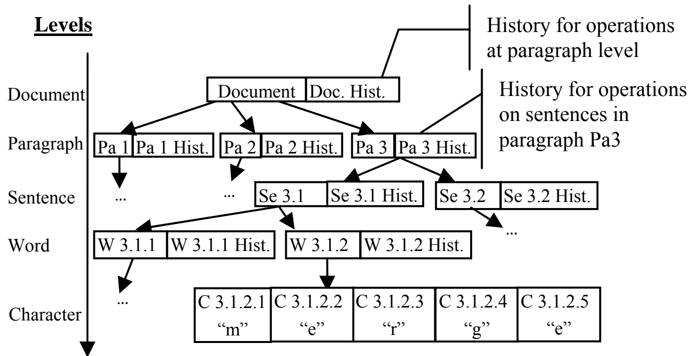


Figure 3. Structure of the document

The hierarchical structure is a general model for a large class of documents and it allows a flexible means of defining and resolving the conflicts. Our approach can be applied on documents representing books, the hierarchical structure consisting of chapters, sections, paragraphs, sentences, words and characters. The proposed approach can also be applied to XML documents.

Moreover, the algorithms for maintaining consistency in collaborative editing based on tree representations of documents achieve an improved efficiency compared to other approaches that use a linear representation of the documents [5]. The existing operation-based linear merging algorithms maintain a single log in the local workspace where the locally executed operations are kept. When the operations from the repository need to be integrated in turn into the local log, the entire local log has to be scanned and transformations need to be performed even though the changes refer to completely different sections of the document and do not interfere with each other. In our approach we keep the log distributed throughout the tree. When an operation from the repository is integrated into the local workspace, only those local logs that are distributed along a certain path in the tree are spanned and transformations performed. The same reduction in the number of transformations is achieved when the operations from the local workspace have to be transformed against the operations from the repository in order to compute the new difference to be kept on the repository. Our merging algorithm recursively applies over the different document levels any existing merging algorithm relying on the linear structure of the document.

The merging algorithm

In this subsection we describe the generalisation of the FORCE merge algorithm presented in the previous section to work on a hierarchical structure of the document. Our merging algorithm recursively applies the FORCE linear approach for merging over the document levels.

The *commit* phase in the case of the tree representation of the documents follows the same principles as in the case of the linear representation. The hierarchical representation of the history of the document is linearised using a breadth-first traversal of the tree: first the operations in the log belonging to the paragraph logs, followed by the operations belonging to the sentence logs and finally the operations belonging to the word logs.

In the *checkout* phase, the local workspace is emptied and all the operations from the repository representing the delta between the version of the document the user wants to work on and the initial version of the document are executed into the local workspace of the user. The checkout phase could also be implemented as described for the linear representation of the documents. The main difference is that, in the FORCE approach, the latest version in the repository is the state of the document and the previous versions are represented by the set of operations constituting the delta between the versions. In our approach, all the versions are represented by the delta set of operations and only the first version in the repository contains the state of the document.

The *update* procedure presented in what follows achieves the actual update of the local version of the hierarchical document with the changes that have been committed by other users to the repository and kept in the remote log. The remote log contains a linearisation of the logs that were initially part of a tree document structure. The goal of the *update* procedure is the same as of the *merge* procedure generalised for the level of the entire document tree, i.e. the replacement of the local log associated with each node with a new one which includes the effects of all non conflicting operations from the remote log and the execution of a modified version of the remote log on the local version of the document in order to update it to the version on the repository. The update procedure is next presented.

```
Algorithm update(CN, RL){
 LLL:=getLog(CN);
 bInd:=|RL|;
 RLL:=[];
 for(i=0;i<|RL|;i++){
   O:=RL[i];
   if(getLevel(O) = getLevel(CN)) append(O,RLL);
   else{  bInd:=i;
          break;
        }
 }
 updateOpInds(LLL,getInds(CN));
 (NRL,NLL):=merge(RLL,LLL);
 for(i=0;i<|NRL|;i++) applyOperation(NRL[i]));
 setLog(CN,NLL);
 ChildRL:=[];
 for(i=0;i<getNoChildren(CN);i++) ChildRL[i]:=[];
 for(i=bInd;i<|RL|;i++){
```

```
  O:=RL[i];
  for(j=0;j<|NLL|;j++) include(O,NLL[j]);
  append(O,ChildRL[getInd(O,getLevel(CN))]);
 }
 for (i=0;i<getNoChildren(CN);i++)
    update(getChildAt(CN,i),ChildRL[i]);
}
```

The *CN* argument of the update procedure represents the current node in the tree traversal, for the initial call of the procedure the current node being equal with the root of the document tree. The parameter *RL* represents the remote log. The local level log *LLL* and the remote level log *RLL* have the same purpose as in the basic merge algorithm, the only difference being that they contain only the part of the remote and local logs referring to the current node. The *RLL* is initialised with the remote operations pertaining to the current node, by iterating over the remote log and keeping those operations whose level is identical to the level of the current node. The level of an operation is equal to the level of the node in whose history the operation is kept. For instance, an *InsertParagraph* operation belongs to the document history and is of level *0*. The *bInd* variable will contain the index of the first operation that refers to a lower level than the level of the current node.

The next step consists of updating the indices of all the operations in the *LLL* so that they correspond to the current position in the tree of the node whose log they belong to. During the update algorithm, nodes might get inserted or deleted from the tree, as we apply the modified remote operations on the local version of the tree. As the positions of the nodes change, it is clear that all operations belonging to the log of the nodes whose position have changed will no longer have valid indices. For example, in the case of a text document, if the local level log contains the operations *DeleteChar("d",1,3,4,5)* and paragraph *1* has been shifted two positions to the right by the insertion of two new paragraphs before it, the operation has to be transformed to *DeleteChar("d",3,3,4,5)*. The basic merge algorithm is called in order to merge the *RLL* and the *LLL* and generate two new logs, *NRL* and *NLL*. In the version of the update procedure presented in this paper, due to the merge procedure, local operations are kept in the case of conflicts. In our current implementation of the asynchronous text editing system, other policies for merging have also been implemented, as described later.

Afterwards, the operations from the *NRL* are applied on the local copy of the document in order to update it and the local log of the current node is then replaced with the *NLL*. We mention that for our merging algorithm we can use any existing linear approach for the merging of two lists of operations. However, in our current implementation, we have used the FORCE merging algorithm. Next, the operations in the remote log starting from *bInd* need to be divided among the children of the current node and the update method called recursively for each child. Each operation in the remote log starting from position *bInd* will be transformed in order to include the effects of all the operations in the *NLL*. This is necessary as operations in the new local log are of higher level than the remaining operations in the remote log and thus can influence the context of the remote operations. Afterwards, the transformed remote operations will be added into the corresponding *ChildRL* elements chosen by analysing the modified index corresponding to the level of the current node. By the end of the iteration, all remote operations will have been transformed and placed in the correct list. Finally, the update method is recursively called with each of the previously created lists of operations as remote logs.

Log Compression

We apply a log compression procedure by which we reduce the size of the log by means of transforming several lower level operations into a single higher level operation. The compression procedure is called before an update or commit is performed. For instance, several *InsertChar* operations which insert characters in the same word, can be grouped into one single *InsertWord* operation inserting the word formed by the target characters of the *InsertChar* operations.

Conflict Definition and Resolution

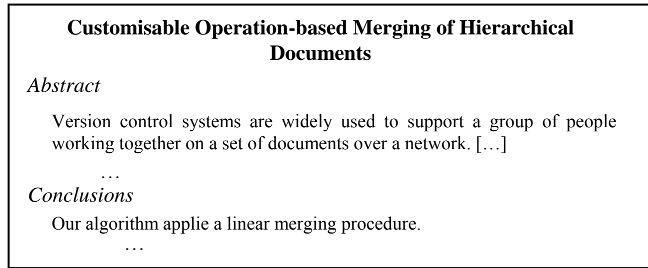
Because of the tree model of the document, the conflicts can be defined at different granularity levels: paragraph, sentence, word or characters. In our current implementation we have defined that two operations are conflicting in the case that they modify the same semantic unit: paragraph, sentence, word or character. The semantic unit is indicated by the working granularity level chosen by the user. The conflicts can be visualised at the chosen granularity levels or at a higher level of granularity. For instance, if the user chooses to work at the sentence level it means that two concurrent operations modifying the same sentence are conflicting. The conflicts can be presented at the sentence level such that the user can choose between the two versions of the sentences. But the user may choose to visualise the conflicts also in the context of the paragraph to which the sentences belong or at an upper level. However, other rules for defining the conflicts can be specified by the implementation of the *semanticConflict* function, such as checking some grammar rules by using the semantic units defined by the hierarchical model.

The conflict resolution policies that we offer in our implementation are automatic or manual, depending on whether conflicts are resolved automatically without the user being prompted for a decision regarding any kind of conflict or manually with the user being asked to choose one version or the other. For automatic resolution we offer two policies for resolving conflicts: to automatically keep only the local operations or only the remote operations in the case of conflict. Concerning the manual resolution policies the user can choose to be presented with each pair

of conflicting operations or he can choose to be presented with the two different effects achieved by applying either all the local operations or all the remote operations pertaining to the conflict unit where the conflict appeared. The user can then choose between the two alternatives.

Example

In what follows we illustrate the asynchronous communication by means of an example. Consider that the repository contains as version V_0 the document illustrated in Figure 4, the document being divided into sections, paragraphs, sentences, words and characters.



Customisable Operation-based Merging of Hierarchical Documents

Abstract

Version control systems are widely used to support a group of people working together on a set of documents over a network. […]

…

Conclusions

Our algorithm applie a linear merging procedure.

…

Figure 4. Example document

Suppose a conflict is defined between two operations concurrently modifying the same word and the policy of merging is that, in the case of conflict, local modifications are kept automatically. Further, assume two users check out version V_0 from the repository into their private workspaces. Assume users are concurrently editing the first paragraph of the Conclusions, namely *"Our algorithm applie a linear merging procedure."* For the sake of simplicity, we will omit the specification of paragraph *p* and section *s* in the following description of operations performed. The first user performs the operations O_{11} and O_{12}, where O_{11}=*InsertChar("d",1,3,7)* and O_{12}=*InsertWord("recursively",1,4).* Operation O_{11} inserts the character *"d"* in the first sentence, third word and operation O_{12} inserts the word *"recursively"* into first sentence, as the fourth word in order to obtain the version

…
Our algorithm applied recursively a linear merging procedure.
…

where […] denotes the other sections of the document that have not been modified. The second user performs the operation O_{21}, O_{21}=*InsertSentence("The approach offers an increased efficiency.",2)* and O_{22}=*InsertChar("s",1,3,7)* in order to obtain

…
Our algorithm applies a linear merging procedure. The approach offers an increased efficiency.
…

Suppose that both users try to commit, but $User_1$ gets access to the repository first, while $User_2$'s request is queued. After the commit operation of $User_1$, the last version in the repository is V_1=*"[…]Our algorithm applied recursively a linear merging procedure.[…]".* DL_{10} representing the difference between V_1 and V_0 in the repository is obtained as a result of the linearisation of the history buffer distributed throughout the tree, $DL_{10}=[O_{12},O_{11}]$.

When $User_2$'s request is processed, the repository sends to $User_2$ a message to update the local copy. Therefore the update procedure is applied, the local tree generated at the site of $User_2$ being traversed in a top-down manner. Firstly the document level history is analysed. But there are no remote operations of section level to be merged. The update is then applied to the section level. Since the operations in our example refer to section *s*, the log referring to section *s* is analysed. But there are no remote operations of paragraph level to be merged. The update is then applied to the paragraph level. Since there are no remote operations of sentence level in paragraph *p*, the processing is applied to the sentence level. The local document contains two sentences, but there are no operations referring to sentence 2, so the merging for sentence 1 will be analysed. Operation O_{12} is of word level, and because there are no local operations of word level, O_{12} will keep its original form. The update procedure will be recursively applied for each of the words belonging to sentence 1. We will analyse only the update applied to the third word of sentence 1, since the remote logs corresponding to the other words in the sentence are empty. The merge procedure will be applied between the list of operations consisting of O_{11} and the list consisting of O_{22}. O_{11} and O_{22} are conflicting and according to the assumed policy the local operation will be kept. As result of this merging, the list of operations to be transmitted to the repository is $[inv(O_{11}),O_{22}]$ and the list of operations to be applied on the local copy of the document is empty. Therefore, the new local version of the document in the workspace of $User_2$ will be *"[...]Our algorithm applies recursively a linear merging procedure. The approach offers an increased efficiency. [..]"* This will also be the new version V_2 of the document in the repository after $User_2$ commits. D_{21} will become $D_{21}=[O_{21}, inv(O_{11}), O_{22}]$.

When $User_1$ updates his local version of the document, the update procedure will be called in order to merge the history buffers distributed along the local tree with the corresponding operations from D_{21}. We are not going to describe the steps of the update procedure in detail, but just remark that, according to our algorithm, the operations of higher level granularity do not need to be transformed against the operations of lower level granularity. For instance, in our example, the operation O_{21} of sentence level does not need to be transformed against any of the local operations in the workspace of $User_1$.

This example illustrated the fact that only a small number of transformations have to be performed using a tree-model of the text document where the local log is distributed

throughout the tree. The operations of a specific granularity do not need to be transformed against the operations of lower level granularity. The performance gain obtained by using a tree representation compared to using the linear representation of the text documents increases with the number of operations to be merged. In this example we have also seen that it is easy to define generic conflict rules involving different semantic units, such as specifying that concurrent insertions in the same word are conflicting.

We mention that in the case of versioning systems such as CVS and Subversion, when $User_2$ is updating the local copy a conflict between the line *"Our algorithm applied recursively a linear merging procedure."* from the repository and the line *"Our algorithm applies a linear merging procedure. The approach offers"* from the workspace is detected, as well as the addition of the line "*an increased efficiency."* $User_2$ has to manually choose between the two conflicting lines and to add the additional line. Most probably $User_2$ will decide to keep his changes and choose the line he edited, as well as adding the additional line. In order to obtain a combined effect of the changes, $User_2$ has to add manually the word *"recursively"* in the local version of the workspace.

RELATED WORK

An operation-based merging approach that uses a flexible way of defining conflicts has been used in FORCE [11]. However, the FORCE approach assumes a linear representation of the document, the operations being defined on strings and not taking into account the structure of the document.

Another approach that uses the principle of transformation of the operations has been proposed in [9]. Although file systems have a hierarchical structure, for the merging of text documents, the authors proposed using a fixed working unit, i.e. the block unit consisting of several lines of text.

Even though other approaches for merging hierarchical documents, such as XML and CRC (Class, Responsibility, Collaboration) documents, have been proposed [8] using the operational transformation approach, our approach achieves a better efficiency since the log is distributed throughout the tree rather than being linear.

CONCLUSIONS

In this paper we proposed a tree based mechanism for maintaining consistency in the case of asynchronous collaborative text editing. Our approach offers an increased efficiency compared to the existing approaches that use a linear structure for representing the document and the possibility of defining and resolving the conflicts at different granularity levels corresponding to the document levels. The proposed algorithm applies the same basic mechanism as the existing operation-based merging algorithms working for linear structures but it is recursively applied over the different document levels.

REFERENCES

1. Berliner, B. CVS II: Parallelizing software development. Proc. of USENIX, 1990.
2. G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In Proc. Int. Conf. on Data Engineering, 2002.
3. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M. *Version Control with Subversion*, O'Reilly, 2004
4. Ellis, C.A., Gibbs, S.J. Concurrency control in groupware systems. *Proc. of the ACM SIGMOD Conf. on Management of Data*, May 1989, pp. 399-407.
5. Ignat, C.-L., Norrie, M.C. Customizable Collaborative Editor Relying on treeOPT Algorithm. *Proc. of ECSCW'03*, Helsinki, Finland, Sept. 2003, pp. 315-334.
6. Li, D., Li, R. Ensuring Content and Intention Consistency in Real-Time Group Editors. *Proc. of ICDCS*, March 2004.
7. Lippe, E., van Oosterom, N. Operation-based merging. *Proc. of the 5th ACM SIGSOFT Symposium on Software development environments*, 1992, pp. 78-87.
8. Molli, P., Skaf-Molli, H., Oster, G. and Jourdain, S. Sams: Synchronous, asynchronous, multi-synchronous environments. *Proc. of CSCW in Design*, 2002.
9. Molli, P., Oster, G., Skaf-Molli, H., and Imine, A. Using the transformational approach to build a safe and generic data synchronizer. *Proc. of Group'03*, 2003.
10. Myers, E. An O(ND) difference algorithm and its variations. Algoritmica, 1(2), pp. 251-266, 1986
11. Shen, H., Sun, C. Flexible merging for asynchronous collaborative systems. *Proc. of CoopIS*, 2002
12. Sun, C., Ellis, C. Operational transformation in real-time group editors: Issues, algorithms, and achievements. *Proc. of CSCW*, Seattle, Nov. 1998, pp. 59-68.
13. Tichy, W.F. RCS- A system for version control. *Software - Practice and Experience*, 15(7), Jul. 1985, pp. 637-654.
14. Vidot, N., Cart, M., Ferrie, J., Suleiman, M. Copies convergence in a distributed real-time collaborative environment. *Proc. of CSCW*, 2000