Operation-based Merging of Hierarchical Documents

Claudia-Lavinia Ignat and Moira C. Norrie

Institute for Information Systems, ETH Zurich, Switzerland
{ignat,norrie}@inf.ethz.ch

Abstract. Version control systems allow a group of people to work together on a set of documents over a network by merging their changes into the same source repository. The existing version control systems offer limited support concerning conflict resolution and tracking of user activity. In this paper we propose a customisable operational transformation merging approach for hierarchical documents that offers the possibility to specify and resolve the conflicts at different granularity levels. Our proposed approach also achieves better efficiency compared to existing approaches for merging documents with linear structures.

1 Introduction

Asynchronous collaborative editing systems have been developed to support a group of people editing a document collaboratively over a network by allowing the users to modify in isolation the copies of the document and afterwards synchronise their copies in order to reestablish a common view of the data.

Well known versioning systems such as CVS [1], RCS [9] and Subversion [2] offer limited support concerning conflict resolution and tracking of user activity. In these systems there is no flexible way of specifying the conflicts, the basic conflict unit being the line. The changes performed by two users are in conflict if they refer to the same line, meaning that multiple changes within a single line cannot be handled by these systems. The merging approaches adopted by these systems are *state-based*, i.e. the merging process uses only the states of the documents. In this way conflicts are presented in the line order in which they appear in the final document. On the other side, *operation-based* merging [5, 8] keeps the information about the evolution of one state of the document into another. Therefore, it provides the possibility of tracking the user operations and of presenting the conflicts in the context in which they were generated.

The FORCE [8] operation based merging approach uses a flexible way of defining conflicts, but it assumes a linear representation of the document, the operations do not taking into account the structure of the document.

In this paper we propose a customisable operation-based merging algorithm that works on a hierarchical representation of documents, allowing the possibility of defining and resolving the conflicts by using different semantic units corresponding to the document levels. An important advantage of our algorithm

based on the tree representation is its improved efficiency compared to other merging approaches that use a linear representation of the documents. Our merging algorithm recursively applies over the different document levels any existing merging algorithm relying on the linear structure of the document.

The paper is structured as follows. In the next section we present an existing linear based merge algorithm that has been used by our tree-based merging algorithm recursively over the document levels. We describe our tree-based approach in Section 3. In Section 4 we compare our approach with some related work. Concluding remarks are presented in the last section.

2 Operational Transformation Linear-based Merging

In this section we are going to briefly introduce the operational transformation mechanism and the way this mechanism has been applied for linear-based merging approaches.

The basic operations supplied by a configuration management tool are checkout, commit and update. A *checkout* operation creates a local working copy of the document from the repository. A *commit* operation creates in the repository a new version of the document based on the local copy of the document, given that the repository does not contain a more recent version of the document to be committed than the local copy. An *update* operation performs the merging of the local copy of the document with the last version of that document stored in the repository.

In order to explain the basic operational transformation approach [3] let us consider the following scenario. Suppose the repository contains the document consisting of one sentence *"He saw the movie."* and two users check-out this version of the document and perform some operations in their workspaces. Further, suppose $User_1$ performs the operation $O_{11} = InsertWord$ *("yesterday",5)*, i.e. intending to insert the word *"yesterday"* as the 5th word, in order to obtain *"He saw the movie* ***yesterday.****"* 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("actually",2)* in order to obtain *"He* ***actually*** *saw the movie."* Before performing a commit, $User_2$ needs to update the local copy of the document. The operation O_{11} has to be transformed to include the effect of the operation O_{21} when it is executed in the workspace of $User_2$. This operation transformation is called *inclusion transformation.* Because the operation O_{21} inserted a word before the word to be inserted by operation O_{11}, operation O_{21} will become an insert operation of the word *"yesterday"* as the 6th word, the result being *"He* ***actually*** *saw the movie* ***yesterday.****"* The inverse operation of inclusion transformation is *exclusion transformation* which transforms an operation O_a against the operation O_b that precedes O_a such that the effect of O_b is excluded from O_a.

In what follows we are going to briefly describe the merging algorithms applied for the linear representation of documents as implemented in [8].

In the commit phase, the repository simply executes sequentially the operations performed in the local workspace. 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.

In the updating phase, the merging algorithm has to be performed between the list of operations executed in the local workspace LL and the list of operations DL representing the delta between the most recent version from the repository and the version the local user started working on. Two basic steps have to be performed. The first step consists of transforming the remote operations from DL in order to include the effect of the local operations. 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 the case that operation O_i belonging to DL is in conflict with an operation from LL, O_i cannot be executed in the local workspace and it needs to be included into the delta as its inverse in order to cancel the effect of O_i. Moreover, all operations following it in the list DL need to exclude its effect.

3 Merging of Hierarchical Documents

In this section we are going to present a generalisation of the previously described linear-based merging algorithm, that works for a hierarchical structure of the document.

We model the text document as consisting of the following levels of granularity: document, paragraph, sentence, word and character, document being the highest granularity level and character being the lowest granularity level.

Each workspace stores locally a copy of the hierarchical structure of the document. Each node (excluding leaf nodes) will keep a history of insertion or deletion operations associated with its children nodes. The structure of the document is illustrated in Figure 1.

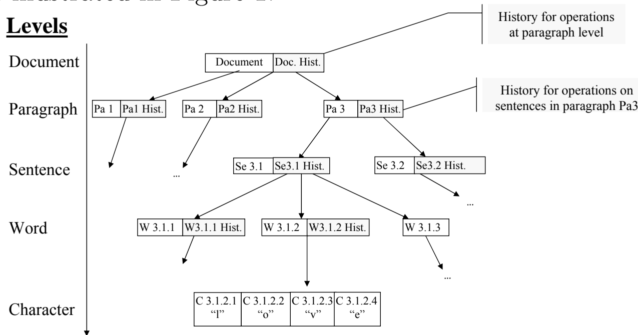

Fig. 1. The structure of the document

Our approach is general and can be applied for any document conforming to a hierarchical structure. For example, it can be applied on documents representing

books, the hierarchical structure consisting of chapters, sections, paragraphs, sentences, words and characters or on XML documents.

The *commit* and *checkout* phase follow the same principles as described for the linear representation of the documents, with the addition that, in the commit phase, the hierarchical representation of the history of the document is linearised by using a breadth-first traversal of the tree. In this way, the first operations in the log will be the ones belonging to the paragraph logs, followed by the operations belonging to the sentence logs and finally the operations belonging to the word logs.

The *update* procedure 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 a linear order into the remote log. The merging procedure has as objective the computing of a new delta to be saved in the repository, 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 repeatedly applied for each level of the document starting from the document level. The first step of the merging algorithm consists in the selection of the *remote level log* containing those operations from the remote log that have the level identical with the level of the operations from the history buffer of the current node. The operations belonging to the *remote level log* are eliminated from the remote log. Since some nodes of the tree might get deleted or inserted during the update process applied for the previous upper levels of the document, the operations in the local log of the current node need to adapt their positions to correspond to the current position in the tree. Afterwards, the basic merging procedure for linear structures is applied to merge the local log of the current node with the *remote level log*. As a result of the merging procedure, the *new remote log* representing the operations that need to be applied on the local document and the *new local log* representing the operations to be saved on the repository are computed. After the *new remote log* is applied locally, the operations from the remote log are transformed against the operations in the local log and are divided among the children of the current node. Afterwards, the merging procedure is recursively called for each child.

As in the case of the real-time collaborative text editing [4], due to the hierarchical structure of the document, only few transformations need to be performed when an operation is integrated into the log as described above, because the operations in the log are distributed throughout the tree model of the document. Only those histories that are distributed along a certain path in the tree are spanned and not the whole log as in the case of a linear model of the document. Moreover, conflicts can be easily expressed using the semantic units (paragraphs, sentences, words, characters), such as rules interdicting the concurrent insertion of two different words into the same sentence. We allow different policies for conflict resolution, such as automatic resolution where the local changes are kept

in the case of a conflict or manual resolution, where the user can choose the modifications to be kept.

In what follows we will illustrate the asynchronous communication by means of an example. Consider that the repository contains as version V_0 the document consisting of only one paragraph with one sentence: *"Absence increase great loves."* Further suppose a conflict is defined between two operations concurrently inserting a character in the same position of the 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. The first user performs the operations O_{11}=*InsertChar("d",1,1,2,9)* of inserting the character *"d"* in the first paragraph, first sentence, second word as the last character and $O_{12} = InsertWord(\text{"the"}, 1, 1, 3)$ in order to obtain the version *"Absence increased the great loves."* The second user performs the operation O_{21}, O_{21} =*InsertSentence("And diminishes small ones.",1,2)* and O_{22} =*InsertChar("s",1,1,2,9)* in order to obtain *"Absence increases great loves. And diminishes small ones."* Suppose that both users try to commit, but $User_1$ gets access to the repository first, while $User_2$'s request will be queued. After the commit operation of $User_1$, the last version in the repository will be V_1 =*"Absence increased the great loves."* The difference between V_1 and V_0 in the repository $DL_{10} = [O_{12}, O_{11}]$ is obtained as a result of the linearisation of the history buffer distributed throughout the tree.

When $User_2$'s request is processed, $User_2$ has to update the local copy, and therefore the update procedure is applied. First the document level history of the local document is analysed. Because no remote operations of paragraph level have to be merged, the update is then applied for the paragraph level by analysing the history of paragraph 1. There are no remote operations of sentence level, so the processing is applied for the sentence level. There are no operations referring to sentence 2, so we will analyse the merging for sentence 1. 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 for the second 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 *newLocalLog*, i.e. the list of operations to be transmitted to the repository, will be $[inv(O_{11}), O_{22}]$ and the *newRemoteLog*, i.e. the list of operations to be applied on the local copy of the document will be empty. Therefore, the new local version of the document in the workspace of $User_2$ will be *"Absence increases the great loves. And diminishes small ones."* This will be also the new version V_2 of the document into the repository after $User_2$ commits. D_{21} will become $D_{21} = [O_{21}, inv(O_{11}), O_{22}]$.

In order to highlight the fact that operations of higher level granularity do not need to be transformed against the operations of lower level granularity, consider the case that $User_1$ updates his local version of the document with

version V_2. Operation O_{21} of sentence level does not need to be transformed against any of the local operations in the workspace of $User_1$.

4 Related Work

Due to the hierarchical structure of the document, in our approach the conflicts can be defined at different semantic levels, as opposed to the RCS [9], CVS [1] and Subversion [2] systems and to the FORCE [8] approach.

Other research works looked at the tree representation of documents for asynchronous collaborative editing, such as working with XML documents. But the merging approach described in [10] is state-based and the one proposed in [7] uses a linear log rather than a distributed log.

In [6], an operational transformation approach has been used for synchronising file systems and file contents. The file systems have a hierarchical structure, however, for the merging of the text documents the authors proposed using a fixed working unit, i.e. the block unit consisting of several lines of text.

5 Conclusions

In this paper we proposed a tree based approach for maintaining the consistency in the case of the asynchronous collaborative text editing that offers also the possibility of defining and resolving the conflicts at different granularity levels corresponding to the document levels.

References

1. Berliner, B.: CVS II: Parallelizing software development. Proceedings of USENIX, Washington D.C., 1990.
2. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion. O'Reilly, 2004, ISBN: 0-596-00448-6
3. 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.
4. Ignat, C.L., Norrie, M.C.: Customizable Collaborative Editor Relying on treeOPT Algorithm, Proc. of ECSCW, Helsinki, Finland, Sept. 2003, pp. 315-334.
5. Lippe, E., van Oosterom, N.: Operation-based merging. Proc. of SIGSOFT Symposium on Software development environments, 1992, pp.78-87
6. 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, Nov. 2003.
7. Molli, P., Skaf-Molli, H., Oster, G. and Jourdain, S. Sams: Synchronous, asynchronous, multi-synchronous environments. Proc. of CSCWD, Rio de Janeiro, Brazil, Sept. 2002
8. Shen, H., Sun, C.: Flexible merging for asynchronous collaborative systems. Proc. of CoopIS/DOA/ODBASE 2002, pp. 304-321.
9. Tichy, W.F.: RCS- A system for version control. Software - Practice and Experience, 15(7), Jul. 1985, pp. 637-654.
10. Torii, O., Kimura, T., Segawa, J.: The consistency control system of XML documents. Symposium on Applications and the Internet, Jan. 2003.