Flexible Definition and Resolution of Conflicts through Multi-level Editing

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

***Abstract*— Version control systems are widely used to support a group of people working together on a set of documents over a network by merging their changes into the same source repository. The existing versioning systems offer limited support concerning conflict resolution and tracking of user activity. In this paper we propose a multi-level editing approach that keeps the editing operations that refer to an element of a hierarchical document associated with that element. In this way, customisable merging is achieved, where the conflicts can be specified and resolved at different granularity levels.**

I. Introduction

Asynchronous collaborative editing systems support a group of people concurrently 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. Versioning systems offer users the possibility to merge their changes into the same source repository.

Well known versioning systems such as CVS [1], RCS [26] and Subversion [3] offer limited support concerning conflict resolution and tracking of user activity. These systems do not offer a flexible means of specifying the possible forms of conflict. 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 regardless of what these changes are. Concurrent changes on different lines are merged automatically. Therefore, these systems cannot handle multiple changes within a single line.

These 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. An operation-based merging approach [13], [22] 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. Merging is done by executing the operations performed on one copy of a document on another copy of the same document. Therefore, complex differentiation algorithms for text such as diff [18] or for XML [2], [11], [27], [28] do not have to be applied in order to compute the delta between documents. Merging based on operations also offers better support for conflict resolution by having the possibility of tracking user operations [13]. State-based approaches take into account just the final and initial states of the document and lose information about the process of transformation from one state to the other, such as the order of execution of the operations. Moreover, there is usually more than one function that can be used to transform an initial state of document into a final one. Operation-based merging records the operations performed, and, therefore, the actual function is well known.

Most of the existing approaches for merging based on operation transformation such as [5], [21], [25], [23], [12] adopt a linear structure of the document. For instance, text documents are seen as a sequence of characters. The definition and resolution of conflicts does not take into account the structure of the document, such as paragraphs, sentences or words.

Some of the operation transformation approaches for merging have been defined for hierarchical documents such as SGML [4] and XML [14]. Even if the structure of the documents is hierarchical, the operational transformation approach is similar to the approach for linear structures and it does not take advantage of the tree structure of the documents. Existing operation-based approaches maintain a single history buffer where a record of the executed operations is kept. Operations are not associated to the structure of the document and therefore it is difficult to select which operations refer to a certain node in the document. The execution of the operations from the history buffer determines the final state of the document. However, the position of the target node of an operation O from the history buffer might have changed due to the execution of the operations that follow O in the history buffer. Therefore, the approach of selection of the operations from the history buffer that refer to a certain node cannot be done by a simple analysis of the operations in the history buffer. This fact has limitations for the definition and resolution of conflicts. Moreover, the approaches described in [4], [14] adopt only automatic resolution of conflicts where the effect of all operations is maintained and they do not offer the user the possibility of defining and resolving conflicts in a flexible way. For instance, it is not possible to define that any operations that refer to the same node are conflicting and let a user choose one of the versions of the node.

In this paper we propose a multi-level editing approach for a hierarchical representation of documents as support for the flexible definition and resolution of conflicts. Multi-level editing involves keeping editing operations that refer to an

1-4244-0429-0/06/$20.00 ©2006 IEEE

element associated with that element. In this way, conflicting operations that refer to the same subtree of the document are easily detected by the analysis of the histories associated with the nodes belonging to the subtree. Therefore, the detection of conflicts is simplified compared to the approach using a single history buffer. Moreover, conflict levels can be dynamically varied and conflict units can be presented in the context in which they occurred or at a higher level. For instance, if conflict was defined at the level of an element, meaning that two operations changing that element are in conflict, the conflict can be presented at the level of the element or at the level of one of the ancestor elements. The proposed 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 as an example. We modeled text documents as consisting of paragraphs, sentences, words and characters. In this way, conflicts can 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.

In [9] we presented our multi-level editing approach for real-time communication where the changes performed by one user are immediately seen by other users. We showed how our multi-level editing approach recursively applies an existing operational transformation linear algorithm over the hierarchical structure of the document. However, when applied for asynchronous communication over a central repository, a linear merging algorithm that offers support for conflict handling has to be recursively applied over the document levels. Therefore, in this paper we show the issues that have to be considered for the adaptation of our multi-level editing approach from real-time to asynchronous collaboration and present our approach for asynchronous communication over a shared repository. We focus on how our approach allows a multi-granularity definition and resolution of conflicts. An asynchronous text editing system has been implemented based on the ideas described in this paper.

The paper is structured as follows. We begin in section II by presenting a motivation of our work. In section III we present an overview of the previous work related to the definition and resolution of conflicts. In section IV we describe the document model that we adopted for supporting multi-level editing. In section V we present the requirements for synchronisation over a shared repository and then in section VI we describe our merging algorithm for hierarchical structures as support for achieving a multi-granularity definition and resolution of conflicts. In section VII we show how conflicts can be defined and resolved from the application interface. Concluding remarks are presented in section VIII.

II. Motivation

In what follows we motivate our approach by means of a scenario, analysing how flexible granularity and policies for the resolution of conflicts could help users in the collaborative editing process. Consider the example of two PhD students writing a research paper together with their professor. At the beginning, they decide on the structure of the paper and divide the work of writing sections. Initially, after writing different sections, their work is easily merged because the parts that they have been working on do not overlap. Even though they were assigned separate parts of the document to work on, some parts of the document such as the bibliography or the introduction may be edited together. Moreover, at a later stage, the sections written by one of the authors will be read by the other authors. In the early stages of writing the paper, the maximum number of modifications performed in parallel should be possible. In this case, it would be appropriate that conflict is detected only if modifications have been performed on the same word.

Suppose that the two students concurrently edit a version of a document consisting of a set of paragraphs. For simplicity we are going to analyse the concurrent work performed on the following paragraph of the document: *"In the case of operation merging, when a conflict occurs, the operation causing the conflict is presented in context in which it was performed. CVS and Subversion present the conflict in line order of the document."*

Suppose that the first student modifies the word *"operation"* to *"operational"* and adds the article *"the"* before the word *"context"* in order to obtain the following version of the document: *"In the case of operation**al** merging, when a conflict occurs, the operation causing the conflict is presented in **the** context in which it was performed.[...]"*

Concurrently, the second user modifies the word *"operation"* to *"operation-based"* and adds the word *"originally"* in order to obtain the following version of the document: *"In the case of operation**-based** merging, when a conflict occurs, the operation causing the conflict is presented in context in which it was **originally** performed.[...]"*

Suppose that the first user commits the changes to the repository. When the second user wants to commit their changes to the repository, they have to first update the local version of the document. Before performing an update, a user should be able to specify the conflict level in the updating process, i.e. the level of granularity where conflicts should be detected. As previously mentioned, in the early stages of writing a paper, it should be appropriate that a user defines conflict at the word level. This means that changes performed in the same word are detected as conflicting and changes performed in different words are not considered conflicting. Concerning the resolution policies for merging, the user should be able to specify whether their changes or the changes in the repository should be kept in the case of conflict. Another possibility for the resolution of conflicts is to let users choose between the two modified conflicting units of the document. Suppose that in our example the second user chooses the word granularity unit for conflict and to manually decide on the version of the document to be kept in the case of conflict. In this case the second user should be asked to choose between the conflicting words *"operation-based"* and *"operational"*. The other operations executed by the two users are not conflicting and they can all be integrated in the merged version of the

document. Suppose that the second user chooses the local modification when presented with the two versions of the conflicting word. Therefore, the local version of the document of the second user becomes: *"In the case of operation-**based** merging, when a conflict occurs, the operation causing the conflict is presented in **the** context in which it was **originally** performed.[...]"*. Further, if the first user updates the local copy of the document, this will be the version of the document in the local workspace.

Note that systems such as CVS and Subversion that use the diff tool for merging will detect the conflicts between the two versions of the document as each version spans a single line and the conflict unit defined in these systems is the line. The user has then to choose one of the two versions. In the case that a combined effect of the changes that have been performed is desired, the user has to manually add the changes performed on the version that was not selected.

Let us continue with our example and the requirements of a versioning tool supporting users collaboratively editing text documents over a shared repository. In a later stage of writing the paper, when changes are critical, the conflict resolution could be set at the sentence or paragraph level. If two modifications have been performed in the same sentence or paragraph respectively, the author committing the changes has to carefully read the two versions of the sentence or paragraph and decide which version to keep. Suppose that each version in the repository is associated with the user who committed that version. In the case that the last version from the repository was committed by the professor, the students might choose to synchronise their local workspaces in accordance with the automatic policy of keeping the changes from the repository in the case of a conflict. In this way, if conflict is detected, the changes of the professor included in the last version in the repository are taken rather than the changes of the students.

In order to illustrate how merging is done at the sentence level, let us continue with our example. Suppose that the two users continue to concurrently edit the last committed version of the document. Suppose that the first user changes the second sentence of the paragraph *"CVS and Subversion present the conflict in line order of the document."* to *"State-based merging systems such as CVS and Subversion present the conflict in the line order of the final document, the state of a line possibly incorporating the effect of more than one conflicting operation."* Concurrently, the second user changes the same sentence in the paragraph to *"For instance, CVS and Subversion present the conflict in line order of the final document."* Assume that the first user commits their changes to the repository first. When the second user updates the changes from the repository, suppose that they choose the sentence as the granularity level for the definition of conflicts and the conflict unit comparison for the resolution of conflicts. The user will then be presented with the two versions of the modified sentence and can choose one of these versions.

Further, consider the case of collaborative editing of XML documents when one user adds some spaces before some element for reformatting purposes, while another user in parallel performs some changes to this element. Versioning systems such as CVS and Subversion will detect conflict since the same line of the document has been modified, even though there is no semantic conflict. Such situations can be avoided if the resolution conflict is set at the level of the element.

As seen from the above examples, there is a need to adopt a flexible means of defining conflicts, as opposed to the fixed unit of conflict (the line) adopted by versioning systems such as RCS [26], CVS [1] and Subversion [3]. Users should be allowed to define conflicts using semantic units such as paragraph, sentence, word or character in the case of text documents. Concerning resolution policies, not only manual resolution for conflicts should be offered, but also other automatic resolution policies, such as, if conflict is detected, to keep the changes in the repository or in the local workspace.

III. Related Work

Due to advantages of operation-based over state-based merging, we adopted a merging approach based on operations.

An operation-based merging approach that uses a flexible way of defining conflicts has been used in FORCE [22]. 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.

The hierarchical structure is a general model for a large class of documents and it allows flexible means of defining and resolving conflicts. Moreover, the algorithms for maintaining consistency in collaborative editing based on the tree representation of documents achieve an improved efficiency compared to other approaches that use a linear representation of the documents, as shown in [9] and shortly explained in what follows. 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 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 scanned 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 in the repository. Our merging algorithm recursively applies over the different document levels any existing merging algorithm relying on the linear structure of the document. In this paper, we show how the histories of operations associated to the nodes in the tree can be used to support a flexible solution for the definition and resolution of conflicts.

At the same time, we have extended our approach to cater for XML documents as well as text documents [8]. Some state-based approaches for merging XML documents have been

proposed in [2], [11], [27], [28]. A state-based approach for merging text documents is the flexible diff [19] approach that finds and reports differences between two versions of text. The PREP writing environment which relies on the diff approach is flexible, allowing users to indicate the granularity of the changes they want to find, the choices being word, phrase, sentence or paragraph. Moreover, the users can control the granularity of how the changes are shown when they are reported, the choices being again word, phrase, sentence or paragraph. If the user chooses a pinpointing granularity of sentence, then any word differences are shown as an old sentence deleted and a new sentence inserted. In contrast to the above mentioned approaches, our approach is operation-based and, as previously described, this yields a number of advantages over state-based merging.

An operational transformation approach for merging file systems was proposed in [15]. File systems have a hierarchical structure, however, for merging of text documents, the authors proposed using a fixed working unit, i.e. the block unit consisting of several lines of text. Other operational transformation approaches for merging hierarchical documents, such as SGML [4] or XML [14] documents, have been proposed. The approaches offer an automatic solution for merging and do not allow the user to customise the definition and resolution of conflicts. We offer both automatic and manual solutions for merging and a multi-level granularity for the definition and resolution of conflicts. Moreover, our approach achieves better efficiency than the approaches described in [15], [4], [14] since the log is distributed throughout the tree rather than being linear.

A flexible object framework that allows the definition of the merge policy based on a particular application was adopted by the Suite collaboration system [16]. Merging can be automatic, semi-automatic or interactive. The objects subject to collaboration are structured and therefore semantic fine-grained policies for merging can be specified. A merge matrix defines merge functions for the possible set of operations. The paper provides as an example the merge matrix for a sequence object, where the operations considered are the deletion of an element, the insertion of an element after another element in the sequence and the modification of an element. The work described in [16] presents how conflicts are handled between two versions of a document, but does not provide a solution for an n-way merging, such as a synchronisation mechanism against a repository.

A follow-up work of [16] is the Sync application [17] that provides high-level primitives in the form of predefined classes that enable programmers to create synchronised, replicated data objects. The Sync approach for merging is based on the merge-model described in [16]. An application of Sync for a drawing-based collaborative tool has been proposed in [17]. Sync allows a merge of a user's change with the server, but requires that the server's change always wins in a conflict. In our approach, various policies for merging can be specified for resolving conflicts, such as executing the local changes, or the changes on the repository or let the user decide on the changes to be kept.

Our approach follows the same idea of a flexible definition and resolution of conflicts as described in [16], [17]. However, we implemented the flexibility for the definition and resolution of conflicts in the context of an operational transformation algorithm for maintaining the order between the semantic units of the document. We identified the semantic units of the document by their position in the tree and we provide exact algorithms for maintaining the order between the semantic units in the presence of concurrent operations. Moreover, we proposed complete merging algorithms for the synchronisation against a shared repository. The approaches proposed in [16], [17] do not provide any correctness criteria that should be satisfied by the proposed merging matrices when used for n-way merging. The criteria for correctness of the transformation functions which have the same meaning as the merging matrices are well defined [21], [23]. In our approach we can use any existing transformation functions that satisfy the correctness properties.

IV. Multi-granularity Document Model

Structuring the document into different semantic units offers users the possibility to define and resolve conflicts in a natural way.

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. The hierarchical structure is created when the document is checked out from the repository and modified while changes are performed locally. Each node (excluding leaf nodes) keeps a history of insertion or deletion operations associated with its child nodes. The structure of the document is illustrated in Figure 1.

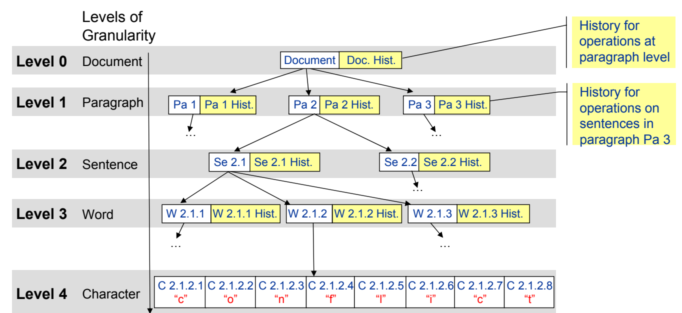

Fig. 1. Structure of the document

The set of operations that can be performed are insertion and deletion of the semantic units composing the document, i.e. paragraphs, sentences, words or characters. For instance, the operation *InsertWord("CollaborateCom",2,2,3)* denotes the insertion of word *"CollaborateCom"* into the 2nd paragraph, 2nd sentence, as the 3rd word. This operation is kept in the history of sentence *Se 2.2*. We are going to refer to the ordered

list of positions composing the path from the root to the target element of the operation as the position vector.

As previously mentioned, our approach is general and can be applied to any document conforming to a hierarchical structure. For example, it can be applied to 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, as shown in [7].

V. Synchronisation over a Shared Repository

In this section we present the synchronisation mechanism over a shared repository by describing the basic methods that should be offered by a version control system. We then present the basic operational transformation mechanism in order to understand the basic steps of the update procedure. We also describe the requirements of an algorithm for dealing with conflicts.

A. Checkout, Commit, Update

The three basic methods supported by a version control system are: checkout, commit and update. A checkout method creates a local working copy of an object from the repository. A commit method creates in the repository a new version of the corresponding object by validating the modifications done on the local copy of the object. The condition of performing this method is that the repository does not contain a more recent version of the object to be committed than the local copy of the object. An update method performs the merging of the local copy of the object with the last version of that object stored in the repository.

In what follows we present the requirements of an operation-based implementation of the commit, checkout and update methods in asynchronous communication with a shared repository.

In the commit phase, a check is first performed as to whether the user can commit the changes to the repository. 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, a commit can be performed. Otherwise, an update is necessary before committing the data. In the case that a commit is allowed, the repository should simply store the operations that were performed in the local workspace.

In the checkout phase, a request should be sent to the repository to specify the version of the document that is intended to be checked out. Using the set of operations stored in the repository as delta, the system should be able to provide to the local workspace either the state of the required version of the document or the set of operations that are the support for computing the state of the required version.

In the updating phase, the repository should send to the local workspace 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 should perform a merging algorithm to update the local version of the document. Consider the scenario where 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 their 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 on the local copy of the user in order to update the local document by integrating the changes included in 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 resulting list of transformed local operations represents the new delta to be saved in the repository.

B. Operational Transformation

In this subsection we present the general operation transformation mechanism that has been used to maintain consistency in real-time collaborative editing [5], [21], [25], [23], [12] as well as in asynchronous collaborative editing [22], [14].

Firstly, we present the notion of context [22] 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, $CT_{O_a} = CT_{O_b}$, 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 $CT_{O_b} = CT_{O_a} \cdot O_a$, 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.

Next, we explain one of the basic mechanisms of the operational transformation approach, called inclusion transformation. 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. Suppose the repository contains the document consisting of one sentence *"We present the merge."* and two users checkout this version of the document and perform some operations in their workspaces. Further, suppose $User_1$ performs the operation O_{11} =*InsertWord ("procedure",5)*. It is an operation to insert the word *"procedure"* at the end of the sentence, as the 5th word, in order to obtain *"We present the merge* ***procedure****."* Note that due to the fact that the document contains only one sentence, we simplified the form of an operation targeting this sentence by skipping the number of the paragraph and of the sentence. 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}. Further assume that, 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 their 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 $O_a =_{CT} O_b$.

Another form of operation transformation used in the process of updating a local copy of the document is exclusion 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. The condition of performing $ET(O_a, O_b)$ is that $O_b \rightarrow_{CT} O_a$.

C. Dealing with Conflicts

Not all operations belonging to DL can be executed in the local workspace as some of these operations may be in conflict with some of the operations from LL. Let us consider that $DL = [O_{d1}, ..., O_{d(i-1)}, O_{di}, O_{d(i+1)}, ..., O_{dm}]$. In the case that O_{di} is in conflict with at least one operation from LL and O_{di} cannot be executed in the local workspace, a mechanism for undoing O_{di} should be provided such that the effect of O_{di} is excluded from the operations that follow it in DL. The effect of undoing O_{di} should be reflected on the repository by storing a new operation after the operations from DL that cancels the effect of O_{di}. The reason that O_{di} should be excluded from the operations O_{dj} that follow O_{di} in DL is that when O_{dj} has to be transformed against the list LL, the form of O_{dj} has to be adapted to illustrate the fact that O_{di} was cancelled. The fact that the undoing of O_{di} should be obtained by executing some new operations following the operations from DL is required by the fact that the list DL is already stored in the repository and cannot be modified as it represents the delta between two versions in the repository. The effect of cancelling O_{di} in the repository due to a conflicting local operation can be made visible in the repository only after the local user commits their changes to the repository. When a commit is performed, the new delta should contain the operations whose effect cancel O_{di}.

In the case that a conflict between O_{di} and an operation in the local log LL occurs and the local operation has to be cancelled, the cancelled local operation should be excluded from the operations that follow it in LL. Then when these local operations are transformed against operations in DL they reflect the fact that an operation preceding them was cancelled.

Therefore, a mechanism for performing the integration of an operation into the log and the cancellation of an operation from the log in the way described above has to be provided.

In [20], [24], [6] mechanisms for performing undo have been proposed, so one of these mechanisms could be used for the cancellation of an operation. For the integration of an operation into a log, one of the algorithms working for real-time communication such as SOCT2 [23] or GOTO [25] could be applied. However, specialised algorithms for asynchronous communication such as FORCE [22] achieve a better performance as shown below.

Suppose that $DL = [O_{d1}, ..., O_{d(i-1)}, O_{di}, O_{d(i+1)}, \ldots, O_{dm}]$ and $LL = [O_{l1}, ..., O_{l(i-1)}, O_{li}, O_{l(i+1)}, ..., O_{ln}]$. The operations in DL and in LL are contextually preceding and O_{d1} and O_{l1} have the same initial context and all operations in DL are concurrent with the operations in LL. Let us analyse the number of transformations that have to be performed to integrate each operation belonging to DL into LL and each operation belonging to LL into DL using the SOCT2 [23] algorithm. Let us analyse first the integration of the operations belonging to DL into LL. When O_{d1} is transformed against all operations in LL, n inclusion transformations will be performed, the result being operation O'_{d1}. When O_{d2} has to be integrated into the transformed local log $LL' = [O_{l1}, \cdots, O_{ln}, O'_{d1}]$, the operations in the log have to be reordered such that the first part of the log contains the operations that precede O_{d2} and the last part of the log contains the operations that are concurrent with O_{d2}. Therefore, O'_{d1} has to be transposed at the beginning of the history buffer. Each step of the transposition involves the computation of an inclusion and exclusion transformation and, therefore, the transposition process requires $2 * n$ transformations. Afterwards, O_{d2} has to be transformed against the concurrent operations and, in this case, n inclusion transformations will be performed. Therefore, the integration of O_{d2} requires $3*n$ transformations. The integration of all operations in DL into LL requires therefore $n + 3*n*(m-1) = 3*n*m - 2*n$ transformations to be performed. Similarly, the integration of all operations belonging to LL into DL requires $3*n*m - 2*m$ operations to be performed. Therefore, the total number of transformations are $6*n*m - 2*m - 2*n$.

The FORCE [22] approach transforms each operation O_{di} in DL in turn with respect to each operation O_{lj} in LL and, after such a transformation is performed, the symmetric transformation of O_{lj} with respect to O_{di} is also performed. The approach requires $2*n*m$ transformations to be performed and the logs have to be traversed only once.

We therefore applied the FORCE algorithm for our merging approach recursively over the document levels.

VI. Our approach

In this section, we describe our merging algorithm for dealing with conflicts applied to hierarchical documents.

The commit phase in the case of the tree representation of 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. In this way, the first operations in the log will be the ones belonging to paragraph logs, followed by operations belonging to sentence logs and finally operations belonging to word logs.

In the checkout phase the operations from the repository are executed in the local workspace.

The update procedure presented below achieves the actual update of the local version of the hierarchical document with

the changes that have been committed to the repository by other users 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 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 in the repository. The update procedure is now presented.

```
Algorithm update(CN, RL, ConfLevel, KeepLocal, Policy) {
  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));

  if(Policy = noMerge)
    if (KeepLocal)
      if (isEmpty(LLL)) NLL = inverse(RLL);
      else (NRL, NLL):=merge(RLL, LLL);
    else
      if (isEmpty(RLL)) NRL = inverse(LLL);
      else (NLL, NRL):=merge(LLL, RLL);
  else
    if(Policy = automatic or Policy = conflictChoice)
      if(KeepLocal) (NRL, NLL):=merge(RLL, LLL);
      else (NLL, NRL):=merge(LLL, RLL);

  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++) {
    Childi = getChildAt(CN, i);
    RLi = ChildRL[i];
    if(level(Childi)! = Level)
      update(Childi, RLi, ConfLevel, KeepLocal, Policy);
    else
      if(RLi = [])
        update(Childi, RLi, ConfLevel, true, noMerge);
      else
        if(isEmpty(getLog(Childi)))
          update(Childi, RLi, ConfLevel, false, noMerge);
        else
          if(Policy = automatic or Policy = noMerge)
            update(Childi, RLi, ConfLevel, KeepLocal,
              noMerge);
          else {
            update(Childi, RLi, ConfLevel, true, noMerge);
            V1 = getChildAt(CN, i);
            setChildAt(CN, i, Childi);
            update(Childi, RLi, ConfLevel, false, noMerge);
            V2 = getChildAt(CN, i);
            setChildAt(CN, i, Childi);
            if(chosen(V1, V2) = V1)
              update(Childi, RLi, ConfLevel, true, noMerge);
            else
              update(Childi, RLi, ConfLevel, false, noMerge)
          }
  }
}
```

The CN argument of the update procedure represents the current node in the tree traversal, and is equal to the root of the document tree in the initial call. The parameter RL represents the remote log. $ConfLevel$ is the conflict level set by the user. For text documents composed of paragraphs, sentences, words and characters, the conflict level can be paragraph, sentence or word. Two operations are in conflict if they refer to the same subtree whose root has been defined to be a conflict unit. By defining a conflict level, all nodes belonging to that level are considered conflict units. $KeepLocal$ is a boolean indicating if the local operations or the remote operations are kept in the case of a conflict. The $Policy$ argument indicates the type of the chosen resolution policy, i.e. automatic or manual, the corresponding values being *automatic* and *conflictChoice*. The policy is propagated down the tree for each recursive call. Different actions have to be taken if processing is done inside or outside a conflict subtree, i.e. a subtree whose root is a conflict unit. We introduced a third policy called *noMerge* used when processing is done inside a conflict subtree. The *noMerge* policy chooses the local or remote operations depending on the decision taken at the root of the conflict subtree and cancels the remote or the local operations respectively.

The local level log LLL and the remote level log RLL contain the parts of the local and remote logs referring to the current node. 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, an *InsertSentence* operation is of level 1, an *InsertWord* operation is of level 2 and an *InsertChar* operation is of level 3. The $bInd$ variable stores the index of the first operation that refers to a lower level than the level of the current node.

The next step is the update of the indices of all the operations in LLL so that they correspond to the current position in the tree of the node to whose log they belong. 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 positions have changed will no longer have valid indices. For example, if the local level

log contains the operation *DeleteChar("d",1,3,4,5)*, denoting an insertion of character *"d"* into paragraph 1, sentence 3, word 4, at position 5 in the word, 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)*.

Depending on the chosen policy, the merge procedure used in FORCE [22] is called in order to merge RLL and LLL and generate two new logs, the new remote log NRL and the new local log NLL, each of which is modified to include the effects of the operations in the other log. If the policy is *noMerge* with the decision to keep local operations and cancel remote operations, the merge procedure is called in its original form. If the local log associated with the current node is empty, remote operations have to be cancelled. This is achieved by keeping in NLL inverses of the operations in the remote log. The case when $KeepLocal$ is false is symmetric to the previously described case. In this case, the merge procedure is called with the local and remote log arguments switched, such that, if conflict is detected, remote operations are considered. Note that for a *noMerge* policy, all local operations are in conflict with the remote operations. If the policy is *automatic* or *conflictChoice*, the FORCE merge procedure is called in its original form.

Afterwards, the operations in NRL are applied to the local copy of the document in order to update it and the local log of the current node is then replaced with NLL. We mention that for our merging algorithm we can use any existing linear approach for merging two lists of operations. However, in our current implementation, we have used the FORCE merging algorithm.

Next, the remaining part of the remote log, i.e. the operations from $bInd$ on, needs 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 $bInd$ on, will be transformed in order to include the effects of all operations in NLL. This is necessary as operations in the new local log are of higher level than the ones remaining in the remote log and thus can influence the context of the remote operations. For instance, suppose that when merging is performed at the paragraph level, NLL contains an operation of insertion of a paragraph as the first paragraph in the document and the remote log contains the operation $O = InsertWord(\text{“}conflict\text{”}, 3, 2, 2)$ of insertion of word *"conflict"* into the third paragraph, second sentence as the second word. In this case, O should be transformed to reflect that word *"conflict"* should be inserted into the fourth paragraph and not into the third paragraph.

Afterwards, the transformed remote operations will be added to the corresponding $ChildRL$ lists chosen by analysing the modified index corresponding to the level of the current node. The function *getInd(O,L)* returns the index of operation O corresponding to the level L. 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 for each child of the current node and its corresponding previously created remote log. However, the resolution method propagated to children depends on the resolution method applied on CN and on the level of the child nodes. If the level of a child of the current node is not equal with the level of conflict, the call of update applied on CN is propagated to its child. If the level of the child node is equal with the level of conflict, the following checking is performed. If the remote list associated to the child node is empty or if there are no local operations targeting the subtree rooted at the child node, it means that no conflicting operations are targeting the child node. The update procedure is then called with the resolution policy of *noMerge*, with the indication to keep only the local or the remote operations, respectively. The *noMerge* policy, as well as the *automatic* policy applied on a node whose level is equal with the conflict level is propagated as a *noMerge* policy. If the resolution policy is *conflictChoice*, two versions of the subtree rooted at *CN* have to be computed, one containing the execution of only the local operations and the other one containing the execution of only the remote operations. The two versions are then presented to the user and the one selected is recomputed.

Transformation functions adapted for linear structures can be used in our approach as explained in what follows. As shown in the update procedure, when merging is performed at the level of the current node of granularity i, the local level log LLL contains operations of granularity level i and the remote log contains both operations of granularity level i from the remote level log RLL and other operations that refer to the current node but are of a finer granularity than i. Transformations between the operations in LLL and RLL have to be performed. As these operations are of different levels of granularity, the inclusion and exclusion transformation functions have to deal with operations of different levels of granularity. However, in what follows we explain how transformation functions for operations of the same level of granularity are used in our approach.

Operations from LLL and RLL are of the same granularity level i, so the transformations of the operations from LLL against the operations from RLL and conversely modify the ith index of the position vector of the operations.

Each operation from LLL and RLL can be transformed into a simple operation of level i, the position parameter of the simple operation representing the ith index of the position vector of the corresponding original operation. We refer to operations characterised by a position vector as composite operations and to operations referring to a certain level in the tree as simple operations. The same idea of transforming a composite operation into a simple one has been applied for our algorithm for the real-time communication [9], [10]. Inclusion and exclusion transformation functions for simple operations can then be applied to compute the transformed positions for level i of the composite operations. The transformed position of the simple operations will represent the ith index of the position vector of the transformed form of the corresponding composite operation.

The operations in RL that follow the operations in RLL are of a finer granularity than i and they have to include the effect of the operations in the new local log NLL. The operations in NLL are of granularity i and they might affect only the position of the operations in RL corresponding to level i. The operations in RL are transformed into simple operations corresponding to level i. Transformation functions working for linear structures can be then applied to find the transformed form of the simple operations. The simple operations can be then reverted to their corresponding complex operations by modifying the ith index in the position vector of the composite operation with the position parameter of the transformed form of the simple operation.

Therefore, the same transformation functions working for linear structures, such as the ones in the SOCT2 [23] or GOTO [25] algorithms, can be recursively applied in our approach. Our approach is general and can apply any merging algorithm and any transformation functions working for linear structures of the document recursively over the hierarchical structure of the document.

VII. Defining and Resolving Conflicts from the Application Interface

In this section, we show how our approach can be used to define and resolve conflicts in a flexible way. Due to the tree model of the document, the conflicts can be defined at different granularity levels: paragraph, sentence, word or character. In our current implementation, we 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 conflict level chosen by the user from the graphical interface. The conflicts can be visualised at the chosen granularity levels or at a higher level of granularity. For example, 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 sentence. It may happen that in order to choose the right version, the user has to read the whole paragraph to which the sentence belongs, i.e. the user can choose to visualise the conflicts also in the context of the paragraph or at an upper level.

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. Concerning manual resolution policies, the conflict unit comparison policy gives a user the possibility to choose between the set of all local operations and the set of all remote operations affecting the selected conflict unit (word, sentence or paragraph). The user is therefore presented with the two units that are in conflict. The policies for the resolution of conflicts can be specified in the graphical interface. The rules for the definition of conflict and the policies for conflict resolution can be specified by each user before an update is performed and they do not have to be uniquely defined for all users. Moreover, for different update steps, users can specify different definition and resolution merge policies.

Consider the example described in section II where two users concurrently modify the document: *"In the case of operation merging, when a conflict occurs, the operation causing the conflict is presented in context in which it was performed. CVS and Subversion present the conflict in line order of the document.[...]"* The concurrent modifications performed by the two users as described in the example in section II are illustrated in Figure 2.

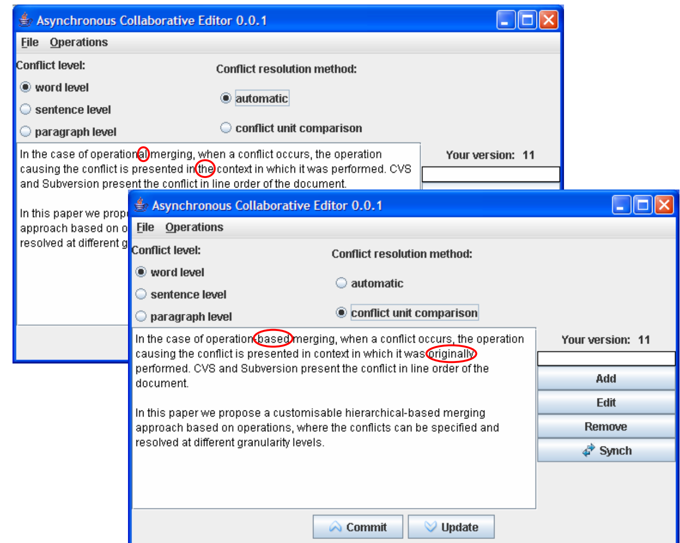

Fig. 2. Concurrent changes of two users

Suppose that the first user commits the changes to the repository. When the second user updates the local version of the document, a conflict level and resolution method have to be specified. Suppose that the user chooses word as the granularity level for the definition of conflicts, meaning that changes performed in the same word are conflicting. Further suppose that the user chooses conflict unit comparison as the resolution method, meaning that when a conflict occurs, the user is asked which of the two versions to keep, the one from the repository or the one from the local workspace.

When an update is performed by the second user, as shown in Figure 3, the non-conflicting changes from the repository are executed and the user is presented with a dialog box to choose between the conflicting words *"operation-based"* and *"operational"*. The user can choose to expand the context for conflict, i.e. to visualise the context where the conflict occurred, both in the local workspace and the repository. In this way, the user can visualise the sentences, the paragraphs or even the whole document where the changes occurred. Suppose that the local modification is chosen, and, therefore, after a commit to the repository, the form of the paragraph for the last version of the document is: *"In the case of operation-based merging, when a conflict occurs, the operation causing the conflict is presented in the context in which it was originally performed. CVS and Subversion present the conflict*

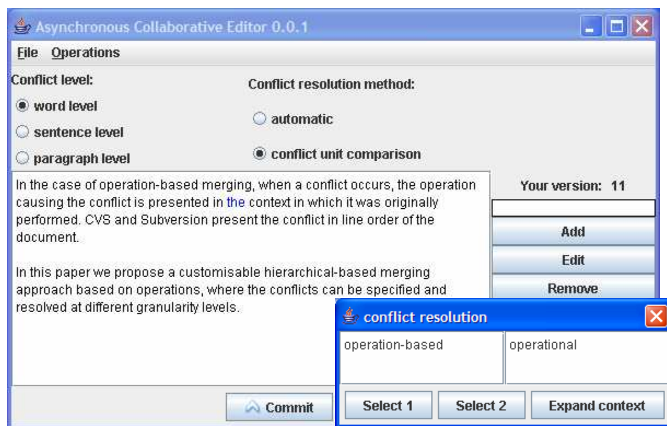

Fig. 3. Conflict resolution for word granularity

in line order of the document.[...]" If the first user updates the local document, this will be the version of the document in their local workspace.

VIII. Conclusions

In this paper we proposed a multi-level editing approach for hierarchical documents that offers support for the definition and resolution of conflicts at different granularity levels corresponding to the document levels. The proposed algorithm recursively applies the same basic mechanism as the existing operation-based merging algorithms working for linear structures over the different document levels.

Our future plans include developing a multi-level undo operation to enable users to undo operations referring to a particular node of the hierarchical document structure.

An asynchronous collaborative editor working for text documents has been implemented in our group. We also implemented an XML editor based on the same ideas described in this paper [7].

References

[1] B. Berliner, "CVS II: Parallelizing software development", *Proc. of USENIX*, Washington D.C., USA, 1990, pp. 341-352

[2] G. Cobena, S. Abiteboul, and A. Marian, "Detecting changes in XML documents", *Proc. of the Intl. Conf. on Data Engineering*, San Jose, California, USA, Feb. 2002, pp. 41-52

[3] B. Collins-Sussman, B.W. Fitzpatrick, and C.M. Pilato, *Version Control with Subversion*, O'Reilly, 2004

[4] A. H. Davis, C. Sun, and J. Lu, "Generalizing operational transformation to the standard general markup language", *Proc. of CSCW*, New Orleans, Louisiana, USA, Nov. 2002, pp. 58-67

[5] C.A. Ellis and S.J. Gibbs, "Concurrency control in groupware systems", *Proc. of the ACM SIGMOD Conf. on Management of Data*, Portland, Oregon, USA, May 1989, pp. 399-407

[6] J. Ferrié, N. Vidot, and M. Cart, " Concurrent Undo Operations in Collaborative Environments Using Operational Transformation", *Proc. of CoopIS*, Larnaca, Cyprus, Oct. 2004, pp. 155-173

[7] C.-L. Ignat and M.C. Norrie, "Flexible Collaboration over XML Documents", *Proc. of CDVE*, Mallorca, Spain, Sept. 2006, pp. 267-274.

[8] C.-L. Ignat and M.C. Norrie, "Supporting Customised Collaboration over Shared Document Repositories", *Proc. of CAiSE*, Luxembourg, Grand-Duchy of Luxembourg, June 2006, pp. 190-204.

[9] C.-L. Ignat and M.C. Norrie, "Customizable Collaborative Editor Relying on treeOPT Algorithm", *Proc. of ECSCW*, Helsinki, Finland, Sept. 2003, pp. 315-334

[10] C.-L. Ignat and M. C. Norrie, "Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems", *Workshop on Collaborative Editing, CSCW 2002*, New Orleans, Louisiana, USA, Nov. 2002

[11] La Fontaine, R., "A Delta Format for XML: Identifying Changes in XML Files and Representing the Changes in XML", *XML Europe*, Berlin, Germany, May 2001

[12] D. Li and R. Li, "Preserving Operation Effects Relation in Group Editors", *Proc. of CSCW*, Chicago, Illinois, USA, Nov. 2004, pp. 457-466

[13] E. Lippe and N. van Oosterom, "Operation-based merging", *Proc. of the 5th ACM SIGSOFT Symposium on Software development environments*, 1992, ACM SIGSOFT Softw. Eng. Notes, Vol. 17, No. 5, pp. 78-87

[14] P. Molli, H. Skaf-Molli, G. Oster, and S. Jourdain, "Sams: Synchronous, asynchronous, multi-synchronous environments", *Proc. of CSCWD*, Rio de Janeiro, Brazil, Sept. 2002, pp.80-85

[15] P. Molli, G. Oster, H. Skaf-Molli, and A. Imine, "Using the transformational approach to build a safe and generic data synchronizer", *Proc. of Group*, Sanibel Island, Florida, USA, Nov. 2003, pp. 212-220

[16] J.P. Munson and P. Dewan, "A flexible object merging framework", *Proc. of CSCW*, Chapel Hill, North Caroline, USA, 1994, pp. 231-242

[17] J. P. Munson and P. Dewan, "Sync: A Java Framework for Mobile Collaborative Applications", *Computer*, 30(6), 1997, pp. 59-66

[18] E. Myers, "An O(ND) difference algorithm and its variations", *Algoritmica*, 1(2), 1986, pp. 251-266

[19] C. M. Neuwirth, R. Chandhok, D. S. Kaufer, P. Erion, J. Morris, and D. Miller, "Flexible diff-ing in a collaborative writing system", *Proc. of CSCW*, Toronto, Ontario, Nov. 1992, pp. 147–154

[20] A. Prakash and M. J. Knister, "A framework for undoing actions in collaborative systems", *ACM Transactions on Computer-Human Interaction*, 1(4), 1994, pp. 295-330

[21] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser", "An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors", *Proc. of CSCW*, Boston, Massachusetts, Nov. 1996, pp. 288-297

[22] H. Shen and C. Sun, "Flexible merging for asynchronous collaborative systems", *Proc. of CoopIS*, California, Irvine, USA, Nov. 2002, pp. 304-321

[23] M. Suleiman, M. Cart and J. Ferrié, "Serialization of Concurrent Operations in a Distributed Collaborative Environment", *Proc. of GROUP*, Phoenix, Arizona, USA, Nov. 1997, pp. 435-445

[24] C. Sun, "Undo as concurrent inverse in group editors", *ACM Trans. Comput.-Hum. Interact.*, 9(4), 2002, pp. 309-361

[25] C. Sun and C.Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements", *Proc. of CSCW*, Seattle, Washington, USA, Nov. 1998, pp. 59-68

[26] W.F. Tichy, "RCS- A system for version control", *Software - Practice and Experience*, 15(7), Jul. 1985, pp. 637-654

[27] O. Torii, T. Kimura, and J. Segawa, "The consistency control system of XML documents", *Symposium on Applications and the Internet*, Orlando, Florida, USA, Jan. 2003, pp. 102-110

[28] Y. Wang, D.J. DeWitt, and J.Y. Cai, "X-Diff: An Effective Change Detection Algorithm for XML Documents", *Proc. of ICDE*, Bangalore, India, March 2003, pp. 519- 530