Supporting Customised Collaboration over Shared Document Repositories

Claudia-Lavinia Ignat and Moira C. Norrie

Institute for Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
{ignat,norrie}@inf.ethz.ch

Abstract. The development of collaborative environments that not only manage information and communication, but also support the actual work processes of organisations is very important. XML documents are increasingly being used to mark up various kinds of data from web content to data used by applications. Often these documents need to be collaboratively created and edited by a group of users. In this paper we present a flexible solution for supporting collaboration over shared repositories containing both XML and text documents. By adopting hierarchical document models instead of linear representations used in most editing systems, the level of conflict granularity and resolution can be varied dynamically and the semantics of the user operations can be easily expressed. Merging of user work is based on the operations performed rather than the document states which provides a less complex and more appropriate way of handling conflicts.

1 Introduction

Collaboration is a central aspect of any team activity and hence of importance to any organisation - be it business, science, education, administration, political or social. The development of collaborative environments that not only manage information and communication, but also support the actual work processes of organisations is therefore very important. While some collaborative activity may involve shared access to databases, a great deal of information central to the operation of organisations is held in documents and support for collaboration is left to file systems, document editors, revision control systems or the users themselves. At the same time, XML documents are increasingly being used to store all kinds of information including not only application data, but also all forms of metadata, specifications, configurations, templates, web documents and even code. While some XML documents are generated automatically from systems, for example database exports, there are many cases where XML documents are created and edited by users either in raw text format or through special tool support.

Although a number of document management and collaborative editing systems have been developed to support collaboration over documents, they tend to focus on a particular form of document or mode of working. For example, some collaborative editors support only synchronous editing of text documents while revision control systems often support only asynchronous editing with support for merging of versions. However, within a single project, it is often the case that different forms of collaboration are used at different stages of the document life cycle and, depending on the activity and mode of working, it should be possible to support both synchronous and asynchronous collaboration and customise the definition and resolution of potential conflicts, as pointed out in [7].

In [8] we described our approach for maintaining consistency for real-time collaboration over text documents. In [5, 6] we described how our approach has been applied to asynchronous collaboration over a repository, the basic unit for collaboration being the text document. In this paper, we focus on asynchronous working over a shared document repository and show how flexible solutions to collaboration over both text and XML documents can be achieved by adopting hierarchical document models instead of the linear representations used in most editing systems. These models enable the level of conflict granularity and resolution to be varied dynamically and capture more of the semantics of user operations. The handling of conflicts is based on the technique of operational transformations applied to different document levels and the merging of user work is therefore based on the operations that they perform rather than the document states which, as we will show, provides a much more appropriate way of detecting and handling conflicts. Rather than providing a full description of the merging algorithms for consistency maintenance, in this paper, we briefly describe the principles of merging and instead focus on the aspects of customisation achieved by our approach in terms of the types of documents supported, i.e. text and XML, and flexibility in the definition and resolution of conflicts.

The paper is structured as follows. We begin in section 2 by presenting the limitations of existing version control systems for collaboration over documents and give an overview of the existing asynchronous collaborative systems for text and XML documents. Section 3 describes the document model that we adopted. In section 4, we present the set of operations that are used to describe the changes performed by the users. In section 5, we show how an existing linear-based merge approach has been used by our tree-based merging approach recursively over the document levels. We then describe in section 6 how conflicts can be defined and resolved in a flexible way in our system. Concluding remarks are presented in section 7.

2 Collaboration over Documents

We start this section with the description of some scenarios showing the set of requirements of a version control system supporting a group of people collaboratively working on set of XML and text documents and the limitations of existing systems. Afterwards we present existing related approaches for asynchronous collaboration on XML and text documents and highlight the contribution of our approach.

Consider the case of a research team in the field of computational physics that wants to publish the results of their simulations in XML documents and also write scientific papers about their research work. The XML format offers a number of advantages for computational physics: clear markup of input data and results, standardised data formats, and easier exchange and archival stability of data. Concurrent editing of the documents containing the data results should be supported as simulations and the gathering of results can be performed in parallel by the members of the group. Documentation for the simulations can be edited and stored in XML or text documents. However, scientific papers should conform to different formats required for publication. Text documents that include formatting instructions cover most of the formats required for publication. For instance, RTF (Rich Text Format) and LaTex documents are text documents including formatting instructions. For the moment, we only consider collaborative editing on raw text documents, but in the future we are going to extend it to text documents that include different formatting instructions.

First consider the editing of XML documents and the case that two researchers concurrently edit the following part of an XML document.

```
<averages>
  <scalar_average name="Energy">
        <mean>-0.9469</mean><error>0.00362</error>
  </scalar_average>
</averages>
```

Assume they concurrently modify the values of the `mean` and `error` elements, with the values `-0.9336` and `0.00299`. The two changes should both be performed and the final version of the document should be

```
<averages>
  <scalar_average name="Energy">
        <mean>-0.9336</mean><error>0.00299</error>
  </scalar_average>
</averages>
```

In the CVS [1] or Subversion [3] systems 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 and therefore the concurrent modification of the `mean` and `error` elements is detected as conflict. The user has then to manually choose one of the modifications. If conflicts would be defined at the level of elements both changes could be taken into consideration. Another case is when one user adds some spaces between the `mean` and `error` elements for reformatting purposes, while another user in parallel performs some changes to the `mean` element. Version control 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. Again, such situations can be avoided if the document is structured into elements and separators and the resolution conflict is set at the level of the element. Situations may arise where a user would like to work exclusively on part of a document. The possibility of locking parts of an XML document before an update procedure is performed is not offered by existing version control systems.

Let us analyse next 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 from the computational physics group 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 have been 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 early stages of writing the paper, the maximum number of modifications performed in parallel should be kept. In this case, defining the conflict at the word level would be appropriate, i.e. conflict is detected only if modifications have been performed on the same word. But, at a later stage when changes are critical, the conflict granularity can be set at the paragraph level. This means that if two modifications have been performed in the same paragraph, the author committing the changes has to carefully read the two versions of the 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, in the case of conflict, the changes of the professor included in the last version in the repository are considered rather than the changes of the students.

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 version control systems such as CVS and Subversion. We propose an approach that allows conflicts to be defined using semantic units corresponding to the structure of the document, such as paragraph, sentence or word in the case of text documents or elements, attributes, separators, words and characters in the case of XML documents. Moreover, in our approach, we offer not only manual resolution for conflicts, but also other automatic resolution policies, such as to keep the changes in the repository or in the local workspace in the case of conflict.

Another disadvantage of existing version control systems such as CVS and Subversion is the fact that they 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 [15, 10] 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 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 documents to be transferred over the network between the local workspaces and the repository. Moreover, no complex differentiation algorithms for XML [17, 2, 9] or diff [14] for text have to be applied in order to compute the delta between the documents. Therefore, the responsiveness of the system is better in the operation-based approach. Merging based on operations also offers better support for conflict resolution by having the possibility of tracking user operations. 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. 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 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 merging approach based on the transformation of operations representing the changes performed during editing. By adopting a tree model of the document, different semantic units can be associated to the document levels and the approach offers a flexible way of defining and resolving conflicts. Our approach is general for any document conforming to a hierarchical structure and we show how it can be applied to both text and XML documents.

In what follows we are going to give a very short overview of the existing approaches for the merging of both text and XML documents.

An operation-based merging approach that uses a flexible way of defining conflicts has been used in FORCE [15]. 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 [11]. 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.

By using a hierarchical model of documents, not only can conflicts be detected and handled in a flexible way, but also the efficiency in terms of the number of transformations performed is improved compared to approaches that use a linear representation of documents, as shown in [8, 6] 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 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.

A flexible object framework that allows the definition of the merge policy based on a particular application was presented in [13]. The objects subject to the collaboration are structured and therefore semantic fine-grained policies for merging can be specified. A merge matrix defines the merge functions for the possible set of operations. The approach proposes different policies for merging, but does not specify an ordering of concurrent operations, such as the order of execution of two insert operations or an insert and delete. The approach does not describe how the difference between two versions of the hierarchical documents is generated. In our approach, we dealt with both the generation of differences between document versions and the handling of conflicts.

Some state-based approaches for merging XML documents have been proposed in [17, 2, 9]. In contrast, our approach is operation-based and we previously highlighted the advantages of merging based on operations compared to state-based merging.

Another operational-transformation approach for merging hierarchical documents, such as XML and CRC (Class, Responsibility, Collaboration) documents, has been proposed in [12]. The environment provides the user with a graphical interface which allows operations to be performed such as the creation and deletion of a new node, the creation and deletion of a certain attribute and the modification of an attribute. By using the graphical interface, no customised formatting for the elements can be used. Modification of a node involves the deletion of the node and the insertion of a new node containing the modified value. Moreover, for text nodes, a lower granularity such as words or characters does not exist. Our approach offers a more natural way of editing XML documents, as we provide a text interface. We have chosen to rather add some additional logic to the editor to ensure well-formed documents than limit the user with a graphical interface. Moreover, our approach achieves better efficiency since the log of operations is distributed throughout the tree rather than being linear.

3 Model of the Document

We now present our model for text and XML documents and the particular issues concerning consistency maintenance during the editing of well-formed XML documents as defined by W3C. We mention that we did not consider issues of checking the validity during collaborative editing of XML documents according to DTD (Document Type Definition) or XML Schema.

We model a text document as being composed of a set of paragraphs, each paragraph containing a set of sentences, each sentence being formed by a set of words and each word containing a set of characters. In this way, the conflicts can be defined and resolved at different granularity levels, corresponding to the document levels (paragraph, sentence, word and character). For instance, a conflict can be defined at the level of sentence, and, in this way, if two users concurrently modify a sentence, a conflict will be detected. Books, a more general form of text documents, also conform to a hierarchical model being formed by chapters, sections, paragraphs, sentences, words and characters.

XML, the popular format for marking up various kinds of data from web content to data used by applications, is also based on a tree model. We classified the nodes of the document into root nodes, processing nodes, element nodes, attribute nodes, word nodes and separator nodes in order that various conflict rules can be defined. A conflict could then be defined, for example, for the case that two users perform operations on the same word node or for the case that users concurrently modify the same attribute node.

When editing XML content, we encounter problems which do not occur when working with text. Consider the case that a user edits an XML document, e.g. by adding the line '`<test>hello world</test>`' character by character. In this way, the XML document will not be well-formed until the closing tag is completed. The editor should provide support to insert complete elements, so that the operations can be tracked unambiguously at any time in the editing process. Our editor offers auto-completeness of elements. For instance, every time the user inserts a '`<`' character, the insertion of '`<></>`' is performed. Of course an empty tag, such as '`<></>`' is not a valid XML element, but at least it allows the desired operation of creating a new element to be addressed in a valid way.

Additional rules for the deletion of characters have to be provided. A user should be prevented from deleting parts of the structure of an element, such as the begin or end tag, unless the whole element is deleted. For instance, the user cannot delete '`</test>`' from an element '`<test>hello world</test>`'. Another issue regarding the editing of elements are the two different forms that an element can take: the form containing both the opening and closing tags such as '`<test></test>`', or the form of an empty element such as '`<test/>`' containing only the closing tag meaning that no further child elements are defined. The user is prevented from directly deleting the closing tag ('`</test>`'). Instead the user can insert a '`/`' character at the end of the starting tag ('`<test>`' $\Rightarrow$ '`<test/>`') in order to tell the system that the element should be transformed into an empty element containing only a closing tag. The operation is not performed if the element contains other *child nodes*. On the other hand, the deletion of the '`/`' character in an empty element leads to the creation of an element containing a begin and end tag.

In the remainder of this section, we discuss detailed handling of each type of node that we used to structure XML documents. The *root node* is a special node representing the virtual root of the document that contains the nodes of the document. The user cannot perform operations on this node.

Processing nodes can be used to define processing instructions in the XML document such as '`<?xml version="1.0"?>`'. In order to keep the XML content valid and to allow insertions of whole elements, the insertion of processing nodes is restricted to complete processing nodes, i.e. '`<??>`' and the deletion of elements referring to the structure of a processing node can be done only if the whole processing node is deleted.

Element nodes represent XML element structures and they consist of an *element name*, as well as some optional *attribute* and *child nodes*. For the following element node '`<test att="val">hello world</test>`', the string '`test`' is the *element name*, '`att="val"`' is an *attribute node* and '`hello world`' is composed of three *child nodes*, namely two *word nodes* and one *separator node.* Similar to processing nodes, in order to ensure well-formed XML documents, only complete element nodes having the form '`<></>`' are allowed to be inserted, and the deletion of characters modifying the structure of the element node is restricted. Element nodes present a further issue as the element name in the opening and closing tag must be the same. As the update of the element name is an atomic operation, the editor alters the element names automatically whenever the user adds/removes some characters to/from the tag name.

The *attribute nodes* can be used either by the processing nodes or the element nodes and they basically consist of a single attribute string. To support the user, the editor will insert the '`=""`' characters automatically whenever the user adds a new attribute.

The *separator nodes* are used to preserve the formatting of the *XML* document and they represent *white spaces* and *quotation marks.*

4 The Set of Operations

In this section, we present the set of operations used to describe the actions performed by users during the editing process of text and XML documents. The set of operations has been chosen to be as minimal as possible, but to allow the flexible definition and resolution of conflicts. Even if the sets of operations for describing the changes performed in the text and XML documents are different, the mechanism for consistency maintenance is the same, as we will show later.

For text editing, the set of operations that can be performed on the model of the document are *insert* and *delete* a semantic unit, such as paragraph, sentence, word or character.

For XML editing, the set of operations contains various forms of insert and delete operations. *INSERT_PROCESSING* inserts a new processing node. *INSERT_ELEMENT* inserts a new element node that can either be a child of the root node or a child of another element node. *INSERT_ATTRIBUTE* inserts a new attribute node that can either be added to a processing or element node. *INSERT_WORD* inserts a new word node that can be added to any element node. *INSERT_SEPARATOR* inserts a new separator node. In order to maintain well-formed documents, the user is not allowed to split the names of processing nodes, elements or attributes by means of separators. *INSERT_CHAR* inserts a character that can be added to update processing or element names, attributes and words. *INSERT_CLOSING_TAG* adds a closing tag. *DELETE_PROCESSING*, *DELETE_ELEMENT*, *DELETE_ATTRIBUTE*, *DELETE_WORD*, *DELETE_SEPARATOR*, *DELETE_CHAR* and *DELETE_CLOSING_TAG* are the delete operations corresponding to the set of insert operations.

5 Operational Transformation Approach

The operational transformation approach [4] is a suitable approach for merging that has been adopted for text documents conforming to a linear structure, such as a sequence of characters. The advantages for merging based on operations compared to state-based merging are, as already pointed out, improved responsiveness and the possibility of tracking the activity of the users.

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, assuming that the repository does not contain a more recent version of the document 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.

We first illustrate the basic operation of the operational transformation mechanism, called inclusion transformation, by means of an example. 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. 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)*. It is an operation 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."*

In what follows, we outline an existing operational transformation approach working on linear structures of documents. Afterwards we present the extension of the linear-based approach working for a hierarchical document structure.

5.1 Linear-based Merging

First we describe the merging algorithm applied to a linear representation of documents as implemented in [15].

In the commit phase, the repository simply executes sequentially the operations performed in the local workspace in order to generate the state of the latest version from the repository. The list of operations sent to the repository represents the difference between the latest two versions of the document. In the checkout phase, in the case that the requested version number of the document equals the latest version number in the repository, the state of the latest version of the document is sent to the local workspace. In the case that the requested version number is less than the latest version number from the repository, the state of the document that is sent to the local workspace has to be computed. It is obtained by executing on the state of the latest version of the document the inverses of the operations that represent the deltas between the latest version in the repository and the requested version.

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 that 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. These transformed operations are then executed on the local 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 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.

In the case that, a user wants to commit the local changes to the repository after performing an update, but in the meantime another user committed his changes to the repository, the first user has to perform a new update.

5.2 Hierarchical-based Merging

The merging approach presented in the previous section works for a linear representation of documents, the operations being defined on strings, without taking into account the structure of the document. Structuring the document into different semantic units allows the possibility for the user to define and resolve the conflicts in a natural way. The approach that we present is a generalisation of the merging mechanism for a linear structure applied to a hierarchical structure.

The disadvantage of the linear-based merging approach is that all operations in the repository and in the local workspace are kept in a single buffer and, when an operation has to be integrated into one of these buffers, a large number of transformations have to be performed. In our approach, the history buffer is distributed throughout the tree, thereby making the merge more efficient as only certain paths in the tree have to be spanned and few transformations are performed. Using the same model, we were also able to improve the efficiency of real-time collaborative editing as reported in [8]. The model of the document is therefore extended by associating to each node in the hierarchical structure (excluding leaf nodes) a history buffer containing operations associated with its children nodes.

The structure of a text-based document is illustrated in Fig. 1. Each internal node of the tree has an associated history containing operations of insertion or deletion of child nodes.

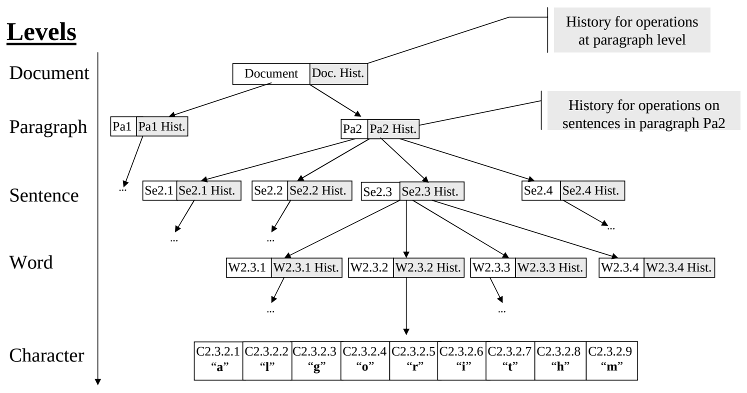


Fig. 1. Structure of a text document

For the XML document below, its tree representation is illustrated in Fig. 2. The attributes of a node are considered to be children of that node.

```
<?xml version="1.0"?>
<addressBook>
    <person id="p001">
        <name>Smith, John< /name>
    < /person>
< /addressBook>
```

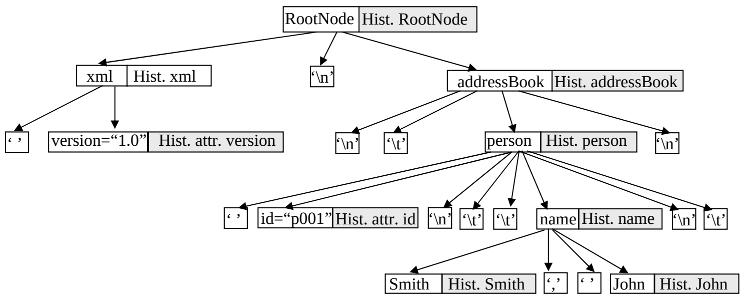


Fig. 2. Structure of an XML document

Operations referring to processing nodes, elements, attributes, words and separators are added to the history associated with the parent node. Operations referring to characters are added to the history associated to the processing target, element names, attributes or words to which they belong. The operations referring to the closing tags are added to the history associated with the element to which they belong.

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 using a breadth-first traversal of the tree. For instance, in the case of text editing, 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.

We now describe the update procedure that we apply for the case of text editing. 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 linear order in the remote log. It has as its 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 to each level of the document starting from the document level. First the *remote level log* is constructed to contain 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. 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. A detailed description of the update procedure is presented in [5].

The same basic ideas underlying the merging of text documents have been applied to the merging of XML documents. While in the case of text editing, transformation functions have been defined between the operations of *insert* and *delete*, in the case of XML editing, transformation functions have been defined for all types of operations targeting processing nodes, elements, attributes, words, characters and separators.

6 Conflict Definition and Resolution

In this section, we show how our approach can be used to define and resolve conflicts in a flexible way.

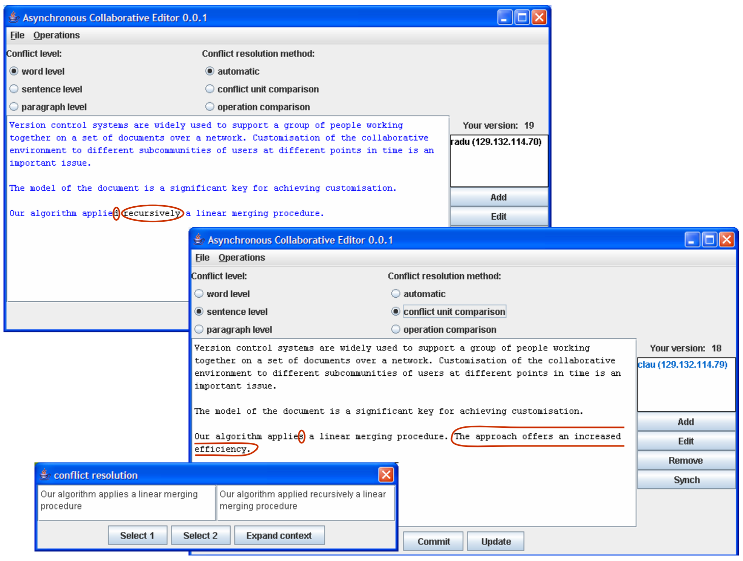
Due to the tree model of the document, for the case of text editing, the conflicts can be defined at different granularity levels: paragraph, sentence, word or character. In our current implementation, we have defined that two operations are conflicting in the case that they modify the same semantic unit: paragraph, sentence, word or character. The semantic unit is indicated by the 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. Other rules for defining the conflicts could be implemented such as to check if some grammar rules are satisfied. This testing can easily be implemented using the semantic units defined by the hierarchical model.

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 user can choose between the operation comparison and the conflict unit comparison policies. The operation comparison policy means that when two operations are in conflict, the user is presented with the effects of both operations and has to decide which of the effects to preserve. In the conflict unit comparison policy, the user has 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.

In order to better understand how conflicts are defined and resolved, we are going to provide the following scenario. Suppose that two users are concurrently editing a document where the last paragraph consists of the sentence: *"Our algorithm applie a linear merging procedure"*. For simplicity, we are going to analyse the concurrent editing performed on this paragraph. The first user adds the character *"d"* at the end of the word *"applie"* and inserts the word *"recursively"* as illustrated in Fig. 3. The second user adds the character *"s"* at the end of the word *"applie"* and the new sentence *"The approach offers an increased efficiency."* as also shown in the figure.

Suppose that, after performing their modifications, the first user commits their changes to the repository. In order to commit to the repository, $User_2$ has to update their local version. In the case that $User_2$ has chosen the conflict level to be sentence and the policy for merging to be conflict unit comparison, the user is presented with the two sentences that are in conflict, as illustrated in Figure 3. Suppose that they choose the variant corresponding to their local version. After the second user performs a commit, the last paragraph of the new version of the document in the repository becomes: *"Our algorithm applie**s** a linear merging procedure. **The approach offers an increased efficiency.**"*

In the case that the second user would have chosen the word level granularity, the conflict would have been detected for the word *"applie"*. The two



Conflict level: [✓ word level] sentence level paragraph level

Conflict resolution method: [✓ automatic] conflict unit comparison operation comparison

```
Version control systems are widely used to support a group of people working
together on a set of documents over a network. Customisation of the collaborative
environment to different subcommunities of users at different points in time is an
important issue.

The model of the document is a significant key for achieving customisation.

Our algorithm applied recursively a linear merging procedure.
```

Your version: 19

radu (129.132.114.70)

Conflict level: word level [✓ sentence level] paragraph level

Conflict resolution method: automatic [✓ conflict unit comparison] operation comparison

```
Version control systems are widely used to support a group of people working
together on a set of documents over a network. Customisation of the collaborative
environment to different subcommunities of users at different points in time is an
important issue.

The model of the document is a significant key for achieving customisation.

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

Your version: 18

clau (129.132.114.79)

Our algorithm applies a linear merging procedure

Our algorithm applied recursively a linear merging procedure

Fig. 3. Conflict resolution

words in conflict would be *"applie**d**"* and *"applie**s**"*. Suppose that the variant corresponding to their local version is chosen. After performing a commit, the last paragraph of the new version of the document in the repository becomes: *"Our algorithm applie**s** **recursively** a linear merging procedure. **The approach offers an increased efficiency.**"*

In this example, we see that it is easy to define generic conflict rules involving different semantic units. We mention that, in the case of version control 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 an increased"* from the workspace will be detected, as well as the addition of the line *"efficiency."* $User_2$ would have to manually choose between the two conflicting lines and to add the additional line. Most probably, $User_2$ will decide to keep their changes and choose the line they 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.

For the case of XML documents, as for the case of text documents, the editor provides a *conflict resolution dialogue* when concurrent changes have been performed on the same granular unit, such as attribute, word or element. The user then needs to decide whether they want to keep the local or the remote version. In the case that a user wants to keep the local version of some parts of the XML document when a merging is performed, they might use the functionality to *lock* nodes of the document.

7 Conclusions

We have presented a customised approach for supporting collaboration over shared repositories containing both text and XML documents. We have shown that, by adopting a hierarchical model of the document, different semantic units can be associated to the document levels and therefore conflicts can be defined and resolved in a flexible way. Our merging approach is operation-based rather than state-based and therefore provides a less complex and more appropriate way of detecting and handling conflicts.

An asynchronous collaborative editor application that allows the editing of both text and XML documents has been implemented in our group based on the ideas described in this paper.

References

1. Berliner, B.: CVS II: Parallelizing software development. Proc. of USENIX, Washington D.C. (1990)
2. Cobena, G., Abiteboul, S., Marian, A.: Detecting changes in xml documents. Proc. of the Intl. Conf. on Data Engineering (2002)
3. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion. O'Reilly, ISBN: 0-596-00448-6 (2004)
4. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. Proc. of the ACM SIGMOD Conf. on Management of Data (1989) 399-407
5. Ignat, C.-L., Norrie, M.C.: Flexible Merging of Hierarchical Documents. Intl. Workshop on Collaborative Editing. GROUP'05, Sanibel Island, Florida (2005)
6. Ignat, C.-L., Norrie, M.C.: Operation-based Merging of Hierarchical Documents. Proc. of the CAiSE'05 Forum, Porto, Portugal (2005) 101-106
7. Ignat, C.-L., Norrie, M.C.: CoDoc: Multi-mode Collaboration over Documents. Proc. of the CAiSE'04, Riga, Latvia (2004) 580-594
8. Ignat, C.-L., Norrie, M.C.: Customisable Collaborative Editor Relying on treeOPT Algorithm. Proc. of ECSCW'03, Helsinki, Finland (2003) 315-334
9. La Fontaine, R.: A Delta Format for XML: Identifying Changes in XML Files and Representing the Changes in XML. XML Europe (2001)
10. Lippe, E., van Oosterom, N.: Operation-based merging. Proc. of the 5th ACM SIGSOFT Symposium on Software development environments (1992) 78-87
11. Molli, P., Oster, G., Skaf-Molli, H., Imine, A.: Using the transformational approach to build a safe and generic data synchronizer. Proc. of Group'03 (2003)
12. Molli, P., Skaf-Molli, H., Oster, G., Jourdain, S.: Sams: Synchronous, asynchronous, multi-synchronous environments. Proc. of CSCWD, Rio de Janeiro, Brazil (2002)
13. Munson, J.P., Dewan, P.: A flexible object merging framework. Proc. of ACM Conf. on CSCW (1994) 231-242
14. Myers, E.: An O(ND) difference algorithm and its variations. Algoritmica, 1(2) (1986) 251-266
15. Shen, H., Sun, C.: Flexible merging for asynchronous collaborative systems. Proc. of CoopIS/DOA/ODBASE (2002) 304-321
16. Vidot, N., Cart, M., Ferrié, J., Suleiman, M.: Copies convergence in a distributed real-time collaborative environment. Proc. of CSCW (2000) 171-180
17. Wang, Y., DeWitt, D.J., Cai, J.Y.: X-Diff: An Effective Change Detection Algorithm for XML Documents. Proc. of ICDE (2003)