Flexible Reconciliation of XML Documents in Asynchronous Editing

Claudia Lavinia Ignat, Gérald Oster

FLEXIBLE RECONCILIATION OF XML DOCUMENTS IN ASYNCHRONOUS EDITING*

Claudia-Lavinia Ignat and Gérald Oster
LORIA-INRIA Lorraine, Campus Scientifique, F-54506 Vandœuvre-lès-Nancy CEDEX, France
Claudia.Ignat@loria.fr, Gerald.Oster@loria.fr

Keywords: Collaborative editing, asynchronous communication, XML, conflict management, operational transformation

Abstract: As XML documents are increasingly being used in a wide variety of applications and often people work in teams distributed across space and time, it is very important that users are supported for editing collaboratively XML documents. Existing tools do not offer appropriate support for the management of conflicting changes performed in parallel on XML documents. In this paper we propose a merging mechanism that offers users the possibility to define conflict nodes prevented from integration of concurrent changes. Changes referring to non-conflict nodes are automatically merged, while users are assisted to manually merge changes referring to conflict nodes. Changes are tracked by means of operations associated to the nodes they target and merging relies on an operation-transformation mechanism adapted for hierarchical structures.

1 INTRODUCTION

Collaboration is a key requirement of teams of individuals working together towards some common goal. Computer-supported collaboration is becoming increasingly common, often compulsory in academia and industry where people work in teams and are distributed across space and time. XML is a popular format for marking up various kinds of data, such as application data, metadata, specifications, configurations, templates, web documents and even code. Often XML documents are created and edited by users either in raw text format or through special tool support. In this paper we describe our approach for supporting users in the process of collaboratively editing XML documents.

A typical way of editing XML documents is the asynchronous collaboration where users work in isolation on their copies of the document and synchronise their changes against a shared repository where changes are published. In this paper we describe our approach for this kind of asynchronous collaboration over XML documents.

*The work presented in this paper was done when both authors were members of the GlobIS group at ETH Zurich, Switzerland.

Some state-based approaches for merging XML documents have been proposed in (Wang et al., 2003; Cobena et al., 2002; Fontaine, 2001). State-based approaches use only the information about the states of the documents and no information about the evolution of one state into another. In this way they do not keep 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. On the other side, operation-based merging approaches (Lippe and van Oosterom, 1992) keep information about the evolution of one document state into another in a buffer containing 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 the other copy of the same document. Conflicts might exist between concurrent changes and therefore an important issue is how to provide users support for the definition and resolution of conflicts. State-based merging approaches detect the units of the document where conflicting changes were performed, but they do not offer support on how to recover from a situation of conflict. Usually, users have to manually edit the versions of the document in order to recover from

conflict. Merging based on operations offers good support for conflict resolution by having the possibility of tracking user operations. User operations can be integrated or cancelled in order to recover from situations of conflict.

Some operation based approaches for merging XML documents have been proposed in (Molli et al., 2002) and (Davis et al., 2002). These approaches did not deal with issues related to the definition and resolution of conflicts. They adopt an automatic resolution of conflicts by combining the effects of all concurrent operations, but do not allow users the possibility to include some changes and restrict others.

For instance, consider the following part of XML document:

```
<?xml version="1.0"?>
<movieDB>
  <movie title="21 Grams">
    <actor>Sean Penn</actor>
  </movie>
</movieDB>
```

Assume that two users start working from the above version of the document. Suppose that the first user inserts the director element <director>Alejandro G. Inarritu</director> as the first child of the movie element.

Further, suppose that the second user, concurrently with the first user, inserts a second actor element <actor>Naomi Watts</actor> as child of the movie element.

One of the possibilities for merging is to take into account the changes performed by both users and to obtain:

```
<?xml version="1.0"?>
<movieDB>
  <movie title="21 Grams">
    <director>Alejandro G. Inarritu</director>
    <actor>Sean Penn</actor>
    <actor>Naomi Watts</actor>
  </movie>
</movieDB>
```

Another possibility is to consider that concurrent changes done at the level of the movieDB element should be integrated, but concurrent changes done at the level of the movie element should not be permitted and users should be asked which changes to keep. For instance, it could have been the case that two users concurrently insert a director element and their works cannot be merged as a movie element has at most one director.

In order to allow such behaviour, it should be possible to define conflict nodes in the hierarchical structure associated with an XML document. Changes performed on the nodes belonging to the path from the root node to the conflict node will be merged, while changes belonging to the conflict nodes will be considered conflicting and users have to manually solve these conflicts.

For keeping track of the changes performed on nodes of the documents, we use a model of the document where we associate operations with the nodes they refer to. 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.

In this paper we present our merging approach for XML documents and present an asynchronous collaborative editor that supports users in the process of managing conflicts. In section 2 we start by describing some requirements for editing XML documents. We then go on by presenting in section 3 the structure of the document and the set of operations modelling the editing process. Section 4 presents our merging approach for XML documents. We show how our merging approach recursively applies an existing operational transformation algorithm over the document hierarchical structures. Moreover, we describe the extension of the operational transformation mechanism for our set of operations. In section 5 we show how users can use our XML editor and set various policies for reconciliation. We then compare in section 6 our approach with other related works. We end our paper by some concluding remarks.

2 EDITING XML DOCUMENTS

In this section we describe some features present in existing single-user XML editors and that should be offered also by collaborative XML editors. For example, single-user XML editors, such as XML Spy (XMLSpy, 2007), offer features of auto-completion to speed up and make more convenient editing of well-formed XML documents. In collaborative editing a necessary condition for obtaining a well-formed reconciled document is that the two XML documents to be merged are well-formed. Therefore, our goal was to build a collaborative editor that uses auto-completion during editing in order to maintain well-formed documents.

Consider 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. Our editor provides support to insert complete elements, so that the operations can be tracked unambiguously at any time in the editing process. For instance, every time the user inserts a '<' character, the insertion of '<></>' is performed. Of course,

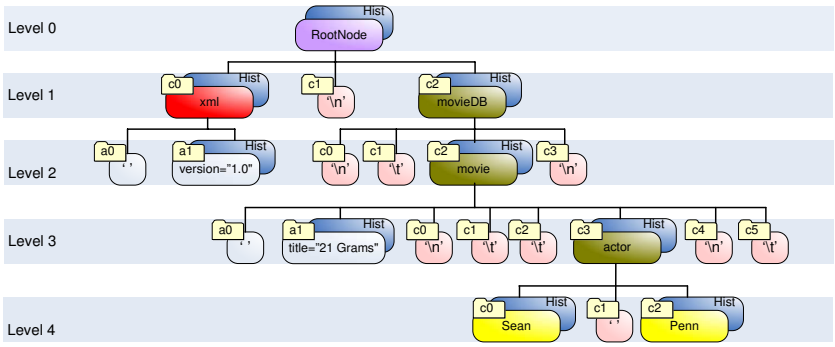

Figure 1: Structure of an XML document

an empty tag such as '<></>' is not a valid XML element, but at least it constitutes a good support for the creation of a new valid element.

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 beginning or closing tag, unless the whole element is deleted. For instance, the user cannot delete '</test>' from an element '<test>hello world</test>'.

Another issue regarding 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 beginning tag ('<test>' $\Rightarrow$ '<test/>') in order to inform the system that the element should be transformed into an 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 beginning and a closing tag.

The editor that we built supports users editing XML documents by automatically validating the content of the documents. The user can format the document, i.e. insert white spaces to make the content more readable, which is not possible using a graphical interface where the user has only a structured view of the content.

3 DOCUMENT MODEL AND OPERATIONS SET

We now present our model for XML documents and the operations used in the editing process of XML documents.

XML documents are based on a tree model. We classified the nodes of the document into *root*, *processing*, *element*, *attribute*, *word* and *separator* in order that various conflict rules can be defined. The *root node* is a special node representing the virtual root of the document that contains the nodes of the document. *Processing nodes* define processing instructions in the XML document. *Element* and *attribute nodes* define elements and attributes of the XML document. *Word nodes* compose the textual content of an XML element. *Separator nodes* are used to preserve XML formatting and they represent *white spaces* and *quotation marks*. 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.

The set of operations contains *insert* and *delete* operations targeting one of the previously mentioned types of nodes, i.e. INSERTPROCESSING, INSERTELEMENT, INSERTATTRIBUTE, INSERTWORD, INSERTSEPARATOR and respectively DELETEPROCESSING, DELETEELEMENT, DELETEATTRIBUTE, DELETEWORD and DELETESEPARATOR. Additionally we defined operations for the insertion and deletion of *characters* to update *processing* or *element* names, *attributes* and *words*, i.e. INSERTCHAR and DELETECHAR. We also defined operations for the insertion and deletion of *closing tags* of elements, i.e. INSERTCLOSINGTAG and DELETECLOSINGTAG.

The two operations INSERTCLOSINGTAG and DELETECLOSINGTAG were considered as a user may want to keep different forms for the representation of empty elements in a certain document and does not want to have an implicitly established form for the representation of empty elements. Our solution to considering both forms for the representation of an element is more general than the solution of having a single form for the visualisation of empty elements.

Elements of an XML document are ordered and,

therefore, each node in the document is identified by a vector of positions representing the path from the root node to the current element. A node contains as children the child element nodes and the attributes associated with that element. For achieving a uniformity between the representation of elements and attributes, we considered that the attributes of an element are ordered. However, to distinguish between child elements and attributes, an element in the position vector has associated a prefix *'c'* or *'a'* showing whether it refers to a child or an attribute element. Consider the following XML document:

```
<?xml version="1.0"?>
<movieDB>
   <movie title="21 Grams">
      <actor>Sean Penn</actor>
   </movie>
</movieDB>
```

The structure of this document is illustrated in Figure 1. We associated different levels with the nodes of the document corresponding to their heights in the tree. Each node in the document, except separator nodes, has an associated history buffer containing the list of operations associated with its child nodes. For instance, the operation of insertion of a second actor element <actor>Naomi Watts</actor> as child of the movie element has the form INSERTELEMENT('<actor>Naomi Watts</actor>',*c2.c2.c4)* and is kept in the history buffer associated with the movie element.

An operation has as argument the position of the node it targets. When an operation is applied, it has to be transformed against other concurrent operations that might change the position of the target node. The idea of storing operations distributed throughout the structure of the tree was to restrict the searching range of operations that might affect an operation. In the model we used in (Ignat and Norrie, 2003) an operation has to be transformed against all other operations in the histories of the nodes on the path from the root to the target node. The model was applied for text documents, where the operations were insertions and deletions of elements. For XML documents more types of operations have to be defined, and, therefore, a decision has to be done where to store these operations.

Operations targeting child elements or attributes such as INSERTELEMENT, DELETEELEMENT, INSERTWORD and DELETEWORD, INSERTATTRIBUTE and DELETEATTRIBUTE change the structure of the element, while operations targeting the tags of an element such as INSERTCLOSINGTAG, DELETECLOSINGTAG, INSERTCHAR and DELETECHAR change the content of the element. Operations that change the structure of an element have to be kept in the history associated with that element. In the same way, operations INSERTCHAR and DELETECHAR targeting a character of a word are kept in the history buffer associated with that word. The main decision that we faced was where to keep operations that change the name of a tag. We decided to keep these types of operations in the history associated with the node they refer due to the following reason. Consider the case of an empty element containing the beginning and closing tags. Further consider that a user is deleting the closing tag of the element. Consider that a second user inserts a child element to the empty element. Operations of deletion of a closing tag and of insertion of elements as direct children of the element whose closing tag has to be deleted cannot be both applied. As seen in section 2, the execution of one of these operations will make impossible the execution of the other operation. We have chosen to cancel the DELETECLOSINGTAG operation and to keep the inserted elements, due to the fact that a DELETECLOSINGTAG operation means simply to rewrite the form of an empty element. As targeting closing tags have to be transformed against operations targeting child elements and vice-versa, we had to keep these operations in the same history buffer. Moreover, the DELETECLOSINGTAG operation is issued by inserting an '/' at the end of the name of the empty element in the beginning tag of the element. The INSERTCLOSINGTAG operation is issued by deleting the '/' at the end of the name of the empty element. Therefore, the DELETECLOSINGTAG and INSERTCLOSINGTAG are implemented as operations of insertion of characters in the name of the empty element. Consequently, operations targeting closing tags and characters in the name of the element are kept in the history buffer associated with that element.

4 MERGING PROCESS

The basic methods supplied by an asynchronous collaborative editing 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.

In the commit phase, the operations executed lo-

cally and stored in the local log distributed throughout the tree have to be saved in the repository. The hierarchical representation of the history of the document is linearised using a breadth-first traversal of the tree, first the operations of level 0, then operations of level 1 and so on. In the checkout phase, the operations from the repository are executed in the local workspace.

In the rest of this section we describe the update stage involving the merging process of XML documents. In the update phase we recursively applied over the different document levels an operational transformation (Ellis and Gibbs, 1989) algorithm for merging lists of operations. Therefore, in what follows we present the basic principles of operational transformation mechanism and then the FORCE (Shen and Sun, 2002) operational transformation algorithm for merging linear structured documents. We then present how we adapted FORCE for merging hierarchical structured documents. We also show that FORCE does not work for the case of dependent operations. We present our solution to adapt the algorithm for our set of dependent operations.

4.1 Operational Transformation

For merging we used the operational transformation approach (Ellis and Gibbs, 1989). We first illustrate the basic mechanism of the operational transformation, 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 whose structure is represented in Figure 1 and two users checkout this version of the document and perform some operations in their workspaces. Further, suppose $User_1$ performs the operation O_1=INSERTELEMENT('<actor>Naomi Watts</actor>',*c2.c2.c4)* to add the '<actor>Naomi Watts</actor>' element on the path $/c2/c2$ as the 4th child of the movie element. Afterwards, $User_1$ commits the changes to the repository and the repository stores the list of operations performed by $User_1$ consisting of O_1. Concurrently, $User_2$ executes operation O_2=INSERTELEMENT('<director>Alejandro G. Inarritu</director>',*c2.c2.c3)* of inserting the element director on the path $/c2/c2$, as the 3rd child of the movie element before the existing actor element in the document. Before performing a commit, $User_2$ needs to update the local copy of the document. O_1 stored in the repository needs to be transformed in order to include the effect of O_2. O_2 and O_1 have the same path from the root element to the parent node and they are operations of the same level. As O_2 inserts an element before the insertion position of O_1, O_1 needs to increase its position of insertion by 1. In this way, the transformed operation of O_1 becomes O_1'=INSERTELEMENT('<actor>Naomi Watts</actor>',*c2.c2.c5)* and it can be executed on the local copy of $User_2$.

4.2 FORCE merging algorithm

In this subsection we describe the FORCE algorithm. Suppose that a user started to work in their local workspace on a copy of version V_n in the repository and executed the list *LL* of operations in their workspace. Suppose that at a later time, the user wants to commit their changes to the repository. Consider that concurrently the version in the repository was updated to version V_{n+1}, and therefore the user has to update their local copy of the document. The merging has to be done between the list *LL* of operations executed by the user in their local workspace and the list *DL* of operations representing the delta between versions V_{n+1} and V_n. 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 to version V_{n+1}. 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 stored in the repository.

From the list of operations in the list *DL* not all of them can be executed in the local workspace as some of these operations may be in conflict with some of the operations from *LL*. If an operation O_{di} from *DL* is in conflict with at least one operation from *LL*, it cannot be executed in the local workspace. Moreover, all operations following it in the list *DL* need to exclude the effect of O_{di} from their context, i.e. they have to be transformed to a form that does not include the effect of O_{di}. In order to exclude the effect of operation O_{di} from the context of all the operations following it in the list *DL*, operation O_{di} has to be transposed towards the end of the list *DL*. The transposition *transpose* (O_a, O_b) between operations O_a and O_b changes the execution order of O_a and O_b and transforms them such that the same effect is obtained as if the operations were executed in their initial order and initial form. When the new delta is saved to the repository it has to include the inverse of the transposed O_{di}

in order to reflect the fact that operation O_{di} was cancelled.

4.3 Merging XML Documents

The *update* procedure updates the local version of the hierarchical document with the changes that have been committed by other users to the repository. It aims to compute a new delta to be saved in the repository, i.e. the transformation of the local operations associated with each node against the non-conflicting operations from the remote log in the repository. Moreover, a modified version of the remote log has to be executed 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. A detailed description of the update procedure applied for text documents represented as a hierarchical structure is presented in (Ignat and Norrie, 2005). For merging XML documents we applied the same principles. In what follows we want to report on the special issues for the adaptation of FORCE linear merging algorithm for XML documents.

As we saw in section 4.2, if an operation O has to be cancelled and it has to be removed from the log, the operation has to be transposed at the end of the log. The FORCE approach considers that a transposition between two operations O_1 and O_2 can always be performed. For operations applied on strings this fact holds true if the two operations do not have overlapping ranges. In FORCE operations in the log are transformed into non-overlapping operations by a compression procedure.

However, generally, ordering constraints between operations exist which do not allow operations to be executed in reverse order. This general case cannot be resolved by a compression procedure. In what follows we present the cases that we encountered in the editing of XML documents that restrict the change of order between operations and the solutions that we adopted.

Between the set of defined operations relations of *dependency* or *before* constraints exist. These constraints restrict the possibility of changing the order between the operations. Consider the case of an empty element of the form '<elem/>'. In order that a child element of this element is inserted, an operation INSERTCLOSINGTAG has to be issued. The operation INSERTELEMENT of insertion of a child element, such as '<subelem></subelem>', in order to obtain '<elem><subelem></subelem></elem>', is said to *depend* on operation INSERTCLOSINGTAG. This means that operations INSERTELEMENT could not have been issued if INSERTCLOSINGTAG would not have been executed before.

Further, consider that the element '<subelem></subelem>' is deleted by issuing the operation DELETEELEMENT and the operation DELETECLOSINGTAG is issued for the element '<elem></elem>' in order to obtain '<elem/>'. Between the operations DELETEELEMENT and DELETECLOSINGTAG there is a *before* constraint, meaning that the order between the operations should be maintained, i.e. the DELETEELEMENT operation and any other operation referring to a child element of a certain node should be ordered before the DELETECLOSINGTAG operation applied on that node.

Therefore, the operations of INSERTELEMENT, DELETEELEMENT, INSERTWORD, DELETEWORD, INSERTPROCESSING, DELETEPROCESSING executed between INSERTCLOSINGTAG and DELETECLOSINGTAG operations and targeting child nodes of the element targeted by INSERTCLOSINGTAG and DELETECLOSINGTAG cannot be transposed to a position outside the range defined by INSERTCLOSINGTAG and DELETECLOSINGTAG.

The solution that we adopted was to detect the cases when the removal of an operation would result into making the transposition of that operation violate existing ordering constraints between operations. If an operation is cancelled, all its dependent and before operations have to be cancelled, too.

5 USER INTERFACE SUPPORT FOR MERGING

Our asynchronous editor for XML documents lets users edit the document from a textual interface, with an overview over the tree structure of the document visualised alongside where the user can define the policies for merging. Two policies were adopted for merging, namely automatic and manual. Automatic policies merge the operations performed locally with the operations from the repository without the intervention of the user. Manual policies for merging involve the intervention of user for conflict resolution.

In what follows we are going to illustrate the merging policies by means of some examples. Assume the two users start working from the same version of the document illustrated in Figure 1. Suppose that the first user inserts a director element as child of the movie element, as shown in the left client window in Figure 2 and afterwards commits the changes to the repository. Suppose that the second user concurrently inserts a second actor element as child of the movie

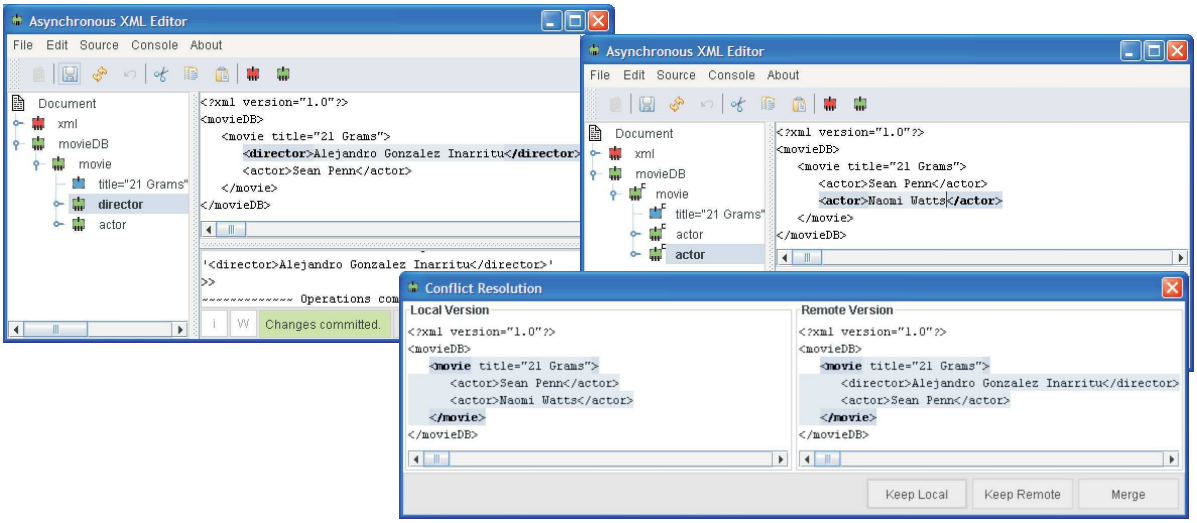

Figure 2: Conflict Resolution for Merging

element, as shown in the right client window in Figure 2. In order to commit their changes, the second user has to update the local version of the document with the changes from the repository.

Assume first that the second user chooses the default merging policy, i.e. the automatic policy for resolution where no rules for the definition or resolution of conflicts are set. The merged version of the document combines the changes performed locally with the remote ones, as explained also in the example presented in section 1.

Alternatively, assume the second user does not want to automatically merge changes, but prefers to set the detection of concurrent modifications targeting the movie element. Concerning the resolution of conflict, suppose that the user wants to manually choose between the conflicting versions of the movie element and, therefore, the user can specify the semantic detection for the node movie on the hierarchical document view. In the case of an update, due to the fact that the node was concurrently modified, the user is then presented with the two versions of the movie element, as shown in Figure 2. The user can then choose to keep either the local or remote version of the document or to perform an automatic merging of the changes.

Another feature provided by our editor is locking of elements. In the case that users want to keep their local modifications referring to parts of document by ignoring concurrent changes on those parts, they can lock the corresponding elements. For instance, in the above example, before performing an update, the second user can choose to lock node movie and therefore keep the local changes performed on the element.

6 RELATED WORK

Some approaches for detection of changes in XML documents have been proposed in (Wang et al., 2003; Cobena et al., 2002; Fontaine, 2001). However, no mechanisms for the reconciliation of detected concurrent changes have been proposed. Moreover, these approaches are state-based and the changes are detected by computing a difference between versions of XML documents. Computing the difference between two versions of hierarchical structures each time a reconciliation has to be performed is very complex and it does not reflect the exact user changes. Our approach is operation-based and changes done by users are easily tracked. We proposed also flexible ways for the reconciliation of documents where users can specify if XML elements should be merged automatically, semi-automatically or manually.

Other operation-based merging approaches have been proposed, but these mechanisms were mainly designed for linear structures, such as (Shen and Sun, 2002). Operation-based approaches for merging XML documents have been proposed in (Molli et al., 2002) and (Davis et al., 2002). As opposed to our approach, these approaches perform an automatic merging of changes and do not let users to flexibly define rules for the reconciliation. Moreover, the set of operations considered in our approach is larger and offers support for the definition of various rules for conflict handling.

A flexible object framework that allows the definition of a merge policy based on a particular application was adopted by the Suite collaboration system (Shen and Dewan, 1992). Merging can be automatic, semi-automatic or interactive. The objects sub-

ject to collaboration are structured and therefore semantic fine-grained policies for merging can be specified. Our approach was driven by the same motivation as Suite of obtaining a flexibility for merging and it was applied for text documents as shown in (Ignat and Norrie, 2006) and for XML documents as shown in this paper. The framework proposed in (Shen and Dewan, 1992) is a general framework where a merge matrix defines merge functions for the possible set of operations. For two concurrent operations it is specified if one of these operations should be executed, if the intervention of users is needed to decide which operation to execute or if both operations should be executed. In the case that both operations should be executed, it is not specified how the two operations should be executed, such as a certain order of execution. In our approach we have an exact mechanism of executing two operations such that their intentions are preserved. In fact, our approach could be seen as a combination of the general merging approach used in Suite and the operational transformation mechanism.

7 CONCLUSIONS

In this paper we proposed a mechanism for the reconciliation of XML documents where users can specify various ways of merging changes referring to an element - automatically, semi-automatically and manually. Our merging approach is based on operations that track user changes performed on different units of the document. We extended the operational transformation mechanism for merging hierarchical structures. An asynchronous collaborative editor over XML documents was build based on the approach described in this paper.

Our approach is currently based on a central repository. We plan to extend collaboration over XML documents for decentralised environments.

REFERENCES

Cobena, G., Abiteboul, S., and Marian, A. (2002). Detecting changes in XML documents. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 41–52, San Jose, California, USA.

Davis, A. H., Sun, C., and Lu, J. (2002). Generalizing operational transformation to the standard general markup language. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work (CSCW '02)*, pages 58–67, New Orleans, Louisiana, USA.

Ellis, C. A. and Gibbs, S. J. (1989). Concurrency control in groupware systems. *SIGMOD Record*, 18(2):399–407.

Fontaine, R. L. (2001). A delta format for XML: Identifying changes in XML files and representing the changes in XML. In *Proceedings of XML Europe*, Berlin, Germany.

Ignat, C.-L. and Norrie, M. C. (2003). Customizable collaborative editor relying on treeOPT algorithm. In *Proceedings of the 8th European Conference on Computer-supported Cooperative Work (ECSCW'03)*, pages 315–334, Helsinki, Finland.

Ignat, C.-L. and Norrie, M. C. (2005). Flexible merging of hierarchical documents. *Seventh International Workshop on Collaborative Editing, GROUP'05, IEEE Distributed Systems online*.

Ignat, C.-L. and Norrie, M. C. (2006). Flexible definition and resolution of conflicts through multilevel editing. In *Proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'06)*, Georgia, Atlanta, USA.

Lippe, E. and van Oosterom, N. (1992). Operation-based merging. In *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 78–87, Tyson's Corner, Virginia, USA.

Molli, P., Skaf-Molli, H., Oster, G., and Jourdain, S. (2002). SAMS: Synchronous, asynchronous, multi-synchronous environments. In *Proceedings of the Conference on Computer-supported Cooperative Work in Design (CSCWD'02)*, pages 80–85, Rio de Janeiro, Brazil.

Shen, H. and Dewan, P. (1992). Access control for collaborative environments. In *Proceedings of the 1992 ACM conference on Computer-supported Cooperative Work (CSCW'92)*, pages 51–58, Toronto, Ontario, Canada.

Shen, H. and Sun, C. (2002). Flexible merging for asynchronous collaborative systems. In *Proceeding of the Conference on Cooperative Information Systems (CoopIS'02)*, pages 304–321, Irvine, California, USA.

Wang, Y., DeWitt, D. J., and Cai, J. (2003). X-diff: An effective change detection algorithm for XML documents. In *Proceedings of the International Conference on Data Engineering (ICDE'03)*, pages 519–530, Bangalore, India.

XMLSpy (2007). Altova XMLSpy. Available online. http://www.altova.com/products_ide.html.