# Flexible Collaboration over XML Documents

Claudia-Lavinia Ignat and Moira C. Norrie

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

**Abstract.** XML documents are increasingly being used to mark up various kinds of data from web content to scientific data. 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 XML documents. Merging of user work is based on the operations performed. A key to achieving flexibility for the definition and resolution of conflicts was to keep the operations distributed throughout the tree model of the document associated with the nodes to which they refer.

## 1 Introduction

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. 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. Our goal was to investigate concurrent editing over XML documents. A key issue in collaboration is merging versions of XML documents produced by different users.

Some state-based approaches for merging XML documents have been proposed in [12, 2, 9]. As opposed to state-based merging, where only the information about the states of the documents is used, the operation-based approaches keep 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 the document on the other copy of the document. Conflicts might exist between concurrent operations and an important issue is how to provide users support for the definition and resolution of conflicts. Merging based on operations offers good support for conflict resolution by having the possibility of tracking user operations.

Operation based approaches for merging XML documents have been proposed in [10] and [4]. These approaches did not deal with issues related to the definition and resolution of conflicts. They maintain a single history buffer where the executed operations are kept and it is difficult to select the operations that refer to a particular node in the document. This fact has limitations for the definition and resolution of conflicts referring to a unit of the document. These approaches adopt an automatic resolution of conflicts by combining the effects of concurrent operations, but do not allow users in the case of conflict to manually select between versions of elements.

In this paper, we propose a flexible merging approach for XML documents where the history buffer containing the performed operations is distributed throughout the tree rather than being linear. We previously applied the distribution of history throughout a tree document model for the case of hierarchical text documents in the real-time [8] and asynchronous [6, 7] modes of collaboration. In this paper we show how the association of operations with the structure of the document is used for the asynchronous collaboration over XML documents with a shared repository. The resolution of conflicts is simplified compared to the approaches that use a single history buffer, as 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. Moreover, conflicts can be dynamically defined at different levels of granularity corresponding to the different levels of the document. The policies for merging can be automatically or manually defined by users.

The paper is structured as follows. In section 2 we describe the requirements for merging and editing XML documents. Section 3 presents the model of the document and the set of operations that we adopted. We then go on to describe in section 4 our approach for merging. In section 5 we present the flexibility of our approach for the definition and resolution of conflicts. We present concluding remarks in section 6.

## 2 Collaboration over XML Documents

In this section we are going to motivate the need for change management over XML documents and give a brief description of the issues to be considered for the editing of XML documents.

Consider the case of a research team in the field of computational chemistry that wants to publish the results of their experiments in XML documents. The XML format offers a number of advantages for computational chemistry, such as clear markup of input data and results, standardised data formats, and easier exchange and archival stability of data.

The results of the simulations may be modified several times by members of the team and various versions may be required at different times. Therefore, a version management system should offer support for the archiving of all versions and for the retrieval of past versions. Members may need to query not only the current value of data, but also past values of some elements of the simulation and process change information.

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.

Consider the case that two researchers concurrently edit the following part of an XML document:

```
<substance role="reagent" title="tert-Butyl hydroperoxide">
  <amount>40</amount> <concentration>5</concentration>
</substance>
```

Assume they concurrently modify the values of the `amount` and `concentration` elements, with the values `38` and `6` respectively. The two changes should both be performed and the final version of the document should be:

```
<substance role="reagent" title="tert-Butyl hydroperoxide">
  <amount>38</amount> <concentration>6</concentration>
</substance>
```

A solution for collaborative editing over XML documents is to represent the XML documents as simple text documents and use existing version systems such as Subversion [3] for performing merging. These systems perform merging 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 modifications of the `amount` and `concentration` elements are detected as conflict. The user has then to manually choose one of the modifications. If the structure of the document is considered and conflicts are defined at the level of elements both changes could be taken into consideration.

We propose an approach that allows conflicts to be defined using semantic units corresponding to the structure of the document, such as elements, attributes, separators, words and characters. Moreover, in our approach, we offer not only manual resolution for conflicts, but also other automatic resolution policies, such as keeping the changes in the repository or in the local workspace in the case of conflict targeting a given node or to lock some parts of the document.

One possibility for editing XML documents is to edit them as simple text documents and the users have then to take care of maintaining well-formed documents, such as the consistency between the names of a begin and end tag of an element. Another possibility for editing XML documents is to use a graphical interface for performing operations of creation and deletion of elements and attributes and of modification of attributes. However, in this way, the user is not allowed to customise the formatting for the elements, such as the use of separators between the elements, as an implicit formatting of the nodes is used.

In our approach, we offer users the possibility of editing XML documents by using a text interface. We added some logic to the editor to ensure well-formed documents, such as the auto-completion of the elements or the consistency between the begin and close tags of an element, in the same way as support is offered to users in existing single-user XML editors, such as XMLSpy [1]. For instance, consider the case that a user edits an XML document, e.g. by typing the element `<conf>CDVE</conf>` 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.

## 3 The Model of the Document and the Set of Operations

We now present our model for XML documents and the particular issues concerning consistency maintenance during the editing of well-formed XML documents.

XML documents are 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. 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. The *separator nodes* are used to preserve the formatting of the *XML* document 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. Additionally we defined operations for the *insertion* and *deletion* of *characters* to update *processing* or *element names*, *attributes* and *words.* We also defined operations for the *insertion* and *deletion* of *closing tags* of 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 to that element. For achieving 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 an index *‘c’* or *‘a’* showing whether it refers to a child or an attribute element. Each node in the document, except separator nodes, has an associated history buffer containing the list of operations associated to its child nodes. For instance, the history buffer associated with an element contains the insertions and deletions of child elements, that can be other nodes, separators or words, of attributes, of characters that change the name of the element and operations that insert or delete the closing tag of the 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 to the nodes of the document corresponding to the heights of the nodes in the tree.

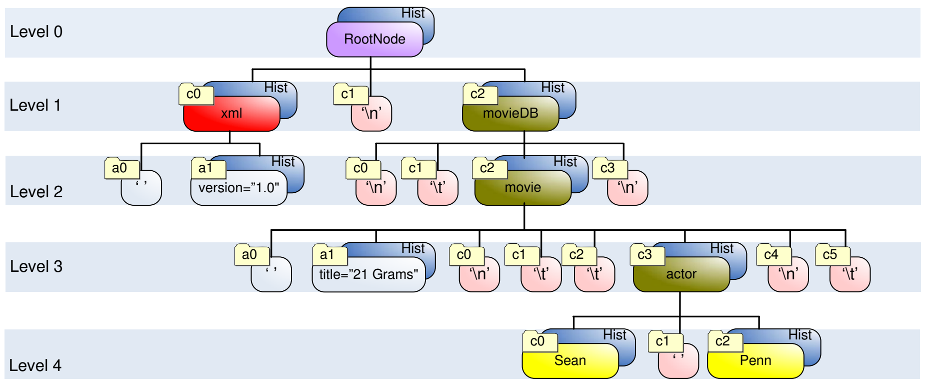


**Fig. 1.** Structure of an XML document

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.

## 4 Operational Transformation Approach

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.

For the merging process we used the operational transformation approach [5]. 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 whose structure is represented in Figure 1 and two users check-out this version of the document and perform some operations in their workspaces. Further, suppose $User_1$ performs the operation $O_{11}$=*InsertElement(* ‘`<actor>Naomi Watts</actor>`’, *c2.c2.c4)* to add the `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 $O_{11}$. Concurrently, $User_2$ executes operation

$O_{21}$=*InsertElement(*'`<director>Alejandro Gonzalez 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. The operation $O_{11}$ stored in the repository needs to be transformed in order to include the effect of operation $O_{21}$. $O_{21}$ and $O_{11}$ have the same path from the root element to the parent node and they are operations of the same level. As operation $O_{21}$ inserts an element before the insertion position of $O_{11}$, $O_{11}$ needs to increase its position of insertion by 1. In this way, the transformed operation of $O_{11}$ becomes $O_{11}$=*InsertElement(* '`<actor>Naomi Watts</actor>`', *c2.c2.c5).*

In the commit phase, the operations executed locally 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.

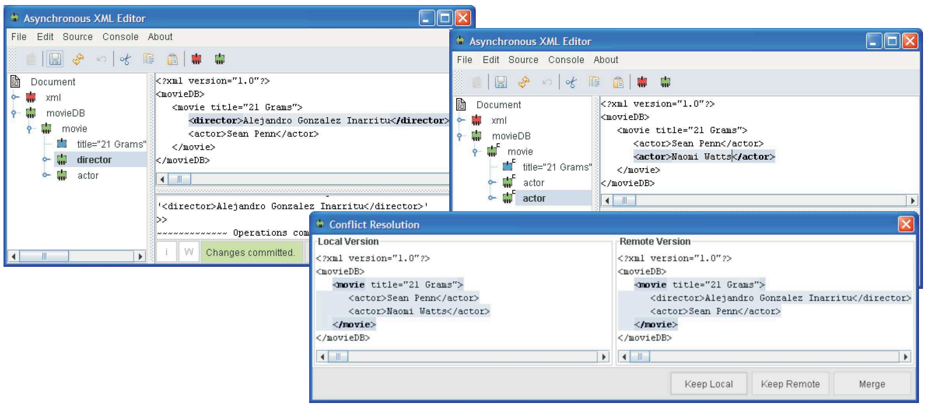
In the update phase we recursively applied over the different document levels the FORCE [11] operational transformation algorithm for merging linear lists of operations. The *update* procedure achieves the actual update of the local version of the hierarchical document with the changes that have been 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 transformation of the local operations associated with each node against the 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. A detailed description of the update procedure applied for text documents represented as a hierarchical structure is presented in [6]. The principles of the update procedure as presented in [6] have been applied also for the XML documents.

## 5 Conflict Definition and Resolution

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. The client interface presents the user with a log showing the operations performed locally and the operations checked out, committed or updated from the repository.

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 the user for the resolution of conflicts.

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.



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

'<director>Alejandro Gonzalez Inarritu</director>'
>>
~~~~~~~~~~~~~~ Operations com
```

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

Local Version

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

Remote Version

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

**Fig. 2.** Conflict Resolution for Merging

Suppose that the second user, concurrently with the first user, inserts a second `actor` element as child of the `movie` 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. The document obtained after merging is given below:

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

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, in the hierarchical representation of the document from the client interface, they can define the semantic detection for the node `movie`. 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.

In the case that the user wants to keep the local modifications if concurrent changes were performed on some elements of the document, they can lock those elements. For instance, in the above example, before performing an update, the second user can choose to lock node `movie` and keep the local changes performed on the element.

## 6 Conclusions

We have presented a customised approach for supporting collaboration over XML documents. We have shown that, by associating operations to the nodes that they target, 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 XML documents has been implemented in our group based on the ideas described in this paper.

## References

1. http://www.altova.com/products_ide.html
2. Cobena, G., Abiteboul, S., Marian, A.: Detecting changes in xml documents. Proc. of the Intl. Conf. on Data Engineering, San Jose, California, USA (2002) 41-52
3. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion. O'Reilly, ISBN: 0-596-00448-6 (2004)
4. Davis, A. H., Sun, C., Lu, J.: Generalizing operational transformation to the standard general markup language. Proc. of CSCW, New Orleans, Louisiana, USA (2002) 58-67
5. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. Proc. of the ACM SIGMOD Conf. on Management of Data, Portland, Oregon, USA (1989) 399-407
6. Ignat, C.-L., Norrie, M.C.: Flexible Merging of Hierarchical Documents. Intl. Workshop on Collaborative Editing. GROUP'05, Sanibel Island, Florida (2005)
7. Ignat, C.-L., Norrie, M.C.: Operation-based Merging of Hierarchical Documents. Proc. of the CAiSE'05 Forum, Porto, Portugal (2005) 101-106
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, Berlin, Germany (2001)
10. Molli, P., Skaf-Molli, H., Oster, G., Jourdain, S.: Sams: Synchronous, asynchronous, multi-synchronous environments. Proc. of CSCWD, Rio de Janeiro, Brazil (2002)
11. Shen, H., Sun, C.: Flexible merging for asynchronous collaborative systems. Proc. of CoopIS/DOA/ODBASE, Irvine, California, USA (2002) 304-321
12. Wang, Y., DeWitt, D.J., Cai, J.Y.: X-Diff: An Effective Change Detection Algorithm for XML Documents. Proc. of ICDE, Bangalore, India (2003)