

SUPPORTING COLLABORATIVE WRITING OF XML DOCUMENTS

Gérald Oster, Hala Skaf-Molli, Pascal Molli

Nancy-Université, LORIA-INRIA Lorraine, Campus Scientifique, BP 239, F-54506 Vandœuvre-lès-Nancy CEDEX, France
{oster, skaf, molli}@loria.fr

Hala Naja-Jazzar

Faculty of Science 3, Lebanese University, Tripoli, Lebanon
hjazzar@ul.edu.lb

Keywords: CSCW, Collaborative Writing, XML, Change Control

Abstract: Synchronisation of replicated shared data is a key issue in collaborative writing systems. Most existing synchronization tools are specific to a particular type of shared data, i.e. text files, calendars, XML files. Therefore, users must use different tools to maintain their different copies up-to-date. In this paper we propose a generic synchronization framework based on the operational transformation approach that supports synchronisation of text files, calendars, XML files by using the same tool. We present how our framework is used to support cooperative writing of XML documents. An implementation is illustrated through the revision control system called So6, which is part of a distributed collaborative technology called LibreSource.

1 INTRODUCTION

Cooperative writing is becoming increasingly common, often compulsory in academic and corporate work. Even the World Wide Web or simply the Web becomes a global read-write information space where multiple authors are interacting in contrast to the traditional model of one author publishing to many readers. People involved in cooperative writing can work across space, time and different organisations. In spite of this need for collaboration, it is surprising to see how poorly computer systems support group activities. Very often, people just send the shared document by mail and use a turn taking strategy to avoid conflicting updates. This is a serious bottleneck for productive work since people cannot work in parallel. Therefore, our initial requirements is that *a good cooperative editor should allow anyone to write any shared data at any time*. Cooperative environments such as Wikis or version control systems such as CVS and Subversion are popular alternatives to the mail approach.

Wiki system is a cooperative writing environment that allows anyone to write at any time the shared documents. However, it restricts editing to a certain type of shared documents, i.e. Wiki pages. A special markup language that offers a simplified alternative to

HTML is used for editing Wiki pages. In case of concurrent modifications, Wikis generally apply the last writer wins rule. Consequently, modifications done by some users may not appear in the last visible page. This is a kind of lost updates.

Our requirements to improve the functionality of cooperative editors can be refined as: *a good cooperative editor should allow anyone to write any shared data at any time without lost updates*.

The existing version control systems such as CVS (Berliner, 1990) avoid lost updates. However, CVS was originally designed to support cooperative software development. Only text files containing code sources such as C file and Java files are considered for merging. In this context, when conflicting changes are performed, conflicts appear inside merged files. A special syntax is used to clearly help programmers to locate the problem. Other shared data types are considered as binary files and concurrent changes on this type of data are not merged. Consequently, CVS allows anyone to write any text files at any time without lost updates.

We want to build a cooperative editor that allows anyone to write any kind of data, not only text files, but also XML files, CAD files, calendar files at any time without lost updates. A generic synchronizer that enables to merge any data type without lost up-

date is required. We propose to build a generic and safe synchronisation framework. This framework allows to synchronise text files, calendars, XML files by using the same tool while ensuring that conflict resolution will not introduce lost updates.

In previous work, we described how the operational Transformation approach (OT) was used as a theoretical foundation to build such a generic and safe synchronizer (Molli et al., 2003). We defined also the specific transformation functions to synchronise linear structure such as text files.

This paper will focus on the transformation functions for XML data and their implementation in an open source collaborative technology called Libre-Source. Our final objective is to build a library for merging blocks of text, strings, trees, graphs. Anyone can use these functions, add new functions or modify existing ones according to their needs.

The paper is structured as follows. Section 2 introduces the operational transformation approach which serves as a theoretical foundation for our generic synchronisation framework called So6. Section 3 presents the architecture and the algorithms used in So6. Section 4 defines the XML transformation functions and demonstrates the use of these functions through an example. Section 5 discusses related work. Section 6 concludes and points out some future work.

2 BACKGROUND

This section describes the Operational Transformation approach (OT) that is the theoretical foundation of the generic and safe synchroniser So6. OT (Ellis and Gibbs, 1989) is an optimistic replication model used in real-time group editors domain. OT considers n sites, each site owns a copy of shared data. When a site performs an update, it generates a corresponding operation, which is first executed locally and then broadcasted to other sites. Every operation is processed in four steps: (a) generated on one site, (b) broadcasted to other sites, (c) received by other sites, (d) executed on other sites.

The execution context of a received operation op_i may be different from its generation context. In this case, the integration of op_i by other sites may lead to inconsistencies between replicas. For instance, we consider two sites $site_1$ and $site_2$ working on a shared data of type *string of characters* initially equal to the string “effect”. A string of characters can be modified with the operation $ins(p,c)$ for inserting a character c at position p in the string. We assume the position of the first character in a string is 0. $user_1$ and

$user_2$ generate and execute two concurrent operations $op_1=ins(2,f)$ and $op_2=ins(5,s)$, respectively. When op_1 is received and executed on $site_2$, it produces the expected string “effects”. But, when op_2 is received on $site_1$, since it does not take into account that op_1 has been executed before it, its execution leads to the state “effectst”. Finally, the copies of $site_1$ and $site_2$ do not converge.

In the operational transformation (OT) approach, before being executed, received operations are transformed regarding concurrent operations that were already executed on the local copy. This transformation is performed by calling transformation functions.

Definition A transformation function T takes two concurrent operations, op_1 and op_2 , must be defined on a same state S . The function computes a new operation op'_1 equivalent to op_1 – i.e. has the same effects – but defined on the state $S' = S \odot op_2$. S' is the state resulting from the execution of op_2 on state S .

Using OT approach, our previous example is now executed as follows. When op_2 is received on $site_1$, op_2 needs to be transformed regarding op_1 . The integration algorithm calls the transformation function $T(op_2=ins(5,s),op_1=ins(2,f)) = ins(6,s) = op'_2$. The insertion position of op_2 is incremented since op_1 has inserted an f before s in state “effect”. After the execution of op'_2 , the state of $site_1$ becomes “effects”. On the contrary, when op_1 is received on $site_2$, the transformation does not modify op_1 's parameters since f is inserted before s . Thus, op_1 is executed as-is and the state of $site_2$ is “effects”. On this scenario, OT approach has ensured that both copies converge to the same value.

The OT approach distinguishes two main components: an *integration algorithm* and a set of *transformation functions*. The integration algorithm is in charge of reception, diffusion and execution of operations. When necessary, it calls transformation functions. This algorithm does not depend on type of replicated data. The transformation functions merge concurrent modifications by serializing two concurrent operations. These functions are specific to a particular type of replicated data such as string of characters, XML documents, calendars or file system.

OT approach aims to achieve *convergence* of copies.

Convergence As every optimistic replication algorithm, OT approach aims to ensure eventual consistency. This means that if no updates are performed for a long period of time, all updates will eventually propagate through the system and all the copies will converge towards a same value. In other words, when

the system is idle (no operation in pipes), all copies are identical.

To ensure convergence, it has been proved (Suleiman et al., 1998) that the underlying transformation functions must satisfy two properties:

Definition The TP_1 property defines a *state equivalence*. The state generated by the execution of op_1 followed by $T(op_2, op_1)$ must be the same as the state generated by the execution of op_2 followed by $T(op_1, op_2)$: $op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$

Definition The TP_2 property ensures that the transformation of an operation regarding a sequence of concurrent operations does not depend on the order in which operations of this sequence were transformed: $T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2))$

The operational transformation approach could be used to design a reconciliation framework able to reconcile divergent copies of any type of data. In order to build such a framework, the following task have to be completed. First, an integration algorithm must be chosen ; regarding this algorithm, TP_2 property may be required on underlying transformation functions. Second, operations which could be performed on shared data types must be defined. Finally, the required transformation functions for all combination of operations have to be provided. In the next sections, we are going to describe our framework.

3 THE SO6 FRAMEWORK

So6 framework is based on SOCT4 integration algorithm (Vidot et al., 2000). Originally, SOCT4 has been designed for real-time group editors and we adapted it for asynchronous interaction (Molli et al., 2003). SOCT4 integration algorithm requires only TP_1 property on transformation functions. It is based on a continuous global order of operations. Shared data are replicated on different sites (workspaces). Each operation generated on a local site is sent with a unique global timestamp to other sites. An operation from a site with a given timestamp can be sent to other sites only if all its preceding operations based on the timestamp order have been received and executed. In this way, SOCT4 ensures that concurrent operations will not be transformed following different transformation paths. This leverages the need for transformation functions to satisfy TP_2 property. Moreover, this mechanism works similarly to the *Copy-Modify-Merge* paradigm widely used in version control systems such as CVS. Regarding this paradigm, a user

can publish her modifications only if she integrated all previously published modifications.

The So6 framework has the following components: one central timestamper also called So6 queue, and several So6 workspaces connected to a timestamper.

3.1 So6 Queue

A So6 queue Q is a timestamper that stores a sequence of operations. An operation is timed when a user sends it to the queue. A queue maintains a timestamp *lastTicket* equal to the last delivered timestamp. When a user creates a queue, the timestamp *lastTicket* is initialized to zero and the sequence of operations is empty. The *publish* procedure assigns a new timestamp to the operation op and stores it in Q .

```
int publish(Operation op) {
    lastTicket++
    Q[lastTicket] = op
    return lastTicket
}
```

3.2 So6 Workspace

Users can work insulated in their workspaces which we are going to refer to as So6 workspaces. A So6 workspace stores all documents shared by the user. This workspace is generally connected to a So6 queue. When users modify a document, they generate corresponding operations. Workspace has the following data structure:

A timestamp *siteTicket*. It memorises the timestamp of the last operation published to or retrieved from the So6 queue.

Two states *currentState* and *referenceState*. They are used to compute the sequence of operations that have been performed locally. *currentState* is the state on which the user works. *referenceState* is the state resulting from the execution of all operations integrated by the site.

A sequence of operations *Hg*. It stores all the operations integrated by the site. This sequence contains all operations published by the site and those retrieved from the timestamper. The operations are ordered according to their timestamps. If the operations contained in the sequence Hg are executed on an empty state, then it obviously computes the state *referenceState*.

Inside a workspace, the following procedures are defined:

A Commit procedure. During this procedure, the system detects local operations generated since last commit. Then, it sends each operation to the So6 queue in order to be time stamped and stored.

```

commit() {
  if (timestamper.lastTicket > siteTicket)
    abort "uptodate_check_failed"
  Operation[] locals =
    computeDifference(referenceState,
                      currentState)

  int ticket;
  for (int i=0; i<locals.length; i++) {
    ticket=timestamper.publish(locals[i])
    execute(locals[i], referenceState)
    Hg[ticket] = locals[i]
  }
  siteTicket = timestamper.lastTicket
}

```

An Update procedure. Through this procedure, the system retrieves unconsumed operations from the So6 queue and merge them with local operations corresponding to unpublished changes.

```

update() {
  Operation[] remotes
  int i=0
  while (siteTicket < timestamper.lastTicket){
    siteTicket++
    i++
    remotes[i] =
      timestamper.retrieve(siteTicket)
  }
  Operation[] locals =
    computeDifference(referenceState,
                      currentState)
  merge(remotes, locals)
}

```

The update procedure calls two other sub-procedures `computeDifference` and `merge`. The `computeDifference` uses a differentiation algorithm to compute the sequence of operations that were executed on the state $state_1$ to obtain the state $state_2$. For instance, in the case of an XML document, any XML differentiation algorithm can be used. For our prototype, we used XyDiff (Cobena et al., 2002). The merge procedure integrates two sequences of concurrent operations using the set of transformation functions T .

```

merge(Operation[] remotes, Operation[] locals) {
  for (int i=0; i<remotes.length; i++) {
    Operation opr = remotes[i]
    int ticket = remotes[i].ticket
    for (int j=0; j<locals.length; j++) {
      Operation opl = locals[j]
      locals[j] = T(opl, opr)
      opr = T(opr, opl)
    }
  }
}

```

```

execute(remotes[i], referenceState)
Hg[ticket] = remotes[i]
execute(opr, currentState)
siteTicket = ticket
}
}

```

This procedure relies on the SOCT4 integration mechanism. Each operation $remote[i]$ must be transformed to an operation opr regarding the whole sequence of local operations. Then, this operation can be executed on the current state $currentState$ of the site. Additionally, the original operation $remote[i]$ is executed on the state $referenceState$.

4 XML DOCUMENTS SUPPORT

In the previous section, we presented our generic framework for reconciling divergent copies of data. In this section, we describe how this framework could be instantiated to support collaboration over XML documents. In (Molli et al., 2003), we instantiated our framework to reconcile a file system and also text documents.

As usual, the XML document is modeled as a node-labelled ordered tree, and each XML element, be it leaf or non-leaf, corresponds to a node of that tree. Since we suppose that the tree is ordered, the children of every node are ordered. Therefore, each node is uniquely identified by its path. This path is defined as the sequence of child number starting from the root. The path of the root node is denoted $[]$. For instance, the XML document presented in Figure 1 is mapped to the tree depicted by Figure 2. And, the path $[0, 1, 0]$ leads to the leaf labelled with the value *The abstract is...*

```

<?xml version="1.0" encoding="UTF-8"?>
<article>
  <sect1>
    <title>Abstract</title>
    <para>The abstract is...</para>
  </sect1>
  <sect1>
    <title>Introduction</title>
    <para>Optimistic replication...</para>
  </sect1>
</article>

```

Figure 1: An example of XML document.

We assume that the tree representation of an XML document can be changed by the following two operations:

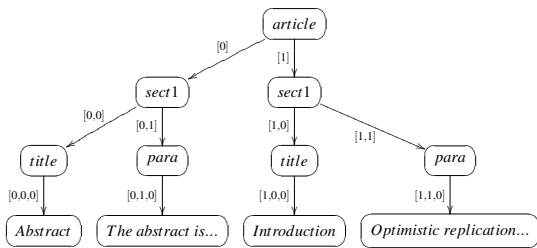


Figure 2: Mapping document of Fig. 1 to an ordered tree

- $addNode(parent, n, val)$ adds a new node as a child of the node identified by the path $parent$. This node is added as n th child and its value – or label – is val .
- $delNode(parent, n)$ deletes the n th child of the node identified by the path $parent$.

For the sake of simplicity, a move operation on a node of the tree is considered equivalent to the deletion of this node from its old location followed by the insertion of this node to its new location.

In order to work with node paths, the following functions are defined. The function $length(p)$ returns the length of the path p , i.e. the number of nodes in this path. The predicate $childOf(p_1, p_2)$ is *true* if the node identified by the path p_1 is a descendant of the node identified by the path p_2 . The function $getPos(p, n)$ returns the $(n + 1)$ th value of the path p , i.e. $getPos([3,2,1,4], 2) = 1$. The function $incPos(p, n)$ computes a new path by incrementing the $(n + 1)$ th value of the path p , i.e. $incPos([3,2,1,4], 2) = [3,2,2,4]$. In the same manner, the function $decPos(p, n)$ computes a new path by decrementing the $(n + 1)$ th value of the path p , i.e. $decPos([3,2,1,4], 2) = [3,2,0,4]$. Finally, the function $codeInf(val_1, val_2)$ allows to compare two values val_1 and val_2 . Such a function can always be defined. For example, for text nodes, $codeInf()$ is defined on the lexicographical order between the values, i.e. $codeInf("Abstract", "Introduction") = true$.

As we explained in section 2, a transformation function computes the result of the integration of two concurrent operations. So, for one XML tree, all possible combinations of operations defined on that XML tree have to be considered. Thus, the following transformation functions for each couple of operations have to be defined: $(addNode(), addNode())$, $(delNode(), delNode())$, $(addNode(), delNode())$ and $(delNode(), addNode())$. Due to space limitations, we are going to describe in details only the transformation function $T(addNode(), addNode())$.

Figure 3 indicates the complete definition of the transformation function T for two concurrent $addNode$ operations. This function transforms $op_1 = addNode(p_1, n_1, v_1)$ regarding

```

T(addNode(p1, n1, v1), addNode(p2, n2, v2)) =
  if (p1 = p2) then
    if (n1 < n2) then addNode(p1, n1, v1)
    elseif (n2 < n1) then addNode(p1, n1 + 1, v1)
    elseif (codeInf(v1, v2)) then addNode(p1, n1, v1)
    elseif (codeInf(v2, v1)) then addNode(p1, n1 + 1, v1)
    else Id()
  endif
elseif (childOf(p1, p2)) then
  if (n2 ≤ getPos(p1, length(p2))) then
    addNode(incPos(p1, length(p2)), n1, v1)
  else addNode(p1, n1, v1)
  endif
else addNode(p1, n1, v1)
endif

```

Figure 3: Transformation function for $addNode$ - $addNode$

$op_2 = addNode(p_2, n_2, v_2)$. The main idea of this function is to compare the insertion position of two concurrent additions of nodes in the XML tree. The following cases have to be considered:

- If the two additions operate on the same parent node, then T compares their insertion positions.
 - If op_1 inserts a child at a position after the insertion position of op_2 then the insertion position of op_1 has to be shifted one position to right. Therefore, its insertion position is incremented.
 - If op_1 inserts a child before the insertion position of op_2 , then the insertion position of op_1 remains the same.
 - If op_1 and op_2 try to insert at the same position, T must decide the serialisation order. In the above definition, the decision of T is based on the $codeInf()$ function, which compares the lexicographic value of nodes. If lexicographic values are equal, then op_1 and op_2 try to insert the same node at the same position, consequently, the function disables the effect of op_1 by transforming it into an *identity* operation. Of course, this is an arbitrary choice and other solutions are possible such as the insertion of both nodes.
- If the two additions operate on different parent nodes, then the previous execution of op_2 might move the parent node of op_1 . This situation occurs when the parent node of op_1 is a child of the parent node of op_2 .

To illustrate, consider the initial XML tree given in the figure 4 and two concurrent operations $op_2 = addNode([1], 1, X)$ and $op_1 = addNode([2], 1, Y)$. The execution of the operation op_2 moves the parent node on which

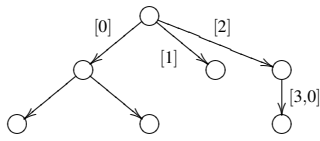


Figure 4: Initial tree

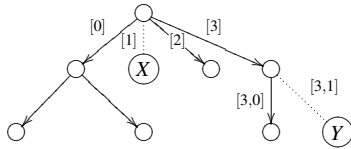


Figure 5: Concurrent additions on different parents

the operation op_1 has to be executed. Therefore, the transformation of the operation op_1 regarding of the operation op_2 must give the operation $op'_1 = addNode([2+1], 1, Y)$. Comparing and updating the position in the path of op_1 is achieved by using $getPos()$ and $incPos()$ functions. The resulting tree is depicted in figure 5.

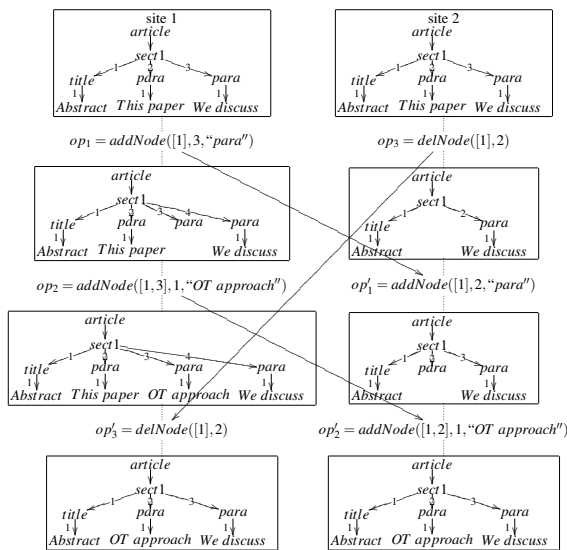


Figure 6: Collaborative Editing Scenario

Transformation functions for couples of operations $delNode-delNode$, $addNode-delNode$ are defined in the same way. Among the pairs of transformation functions, there is a critical case to consider: what to do when an operation removes a subtree while another concurrent one appends a node to this subtree? This is clearly a case of conflict. The solution we chose is to remove the subtree even if in this case the concurrent changes performed on this subtree are lost. This solution allows to ensure data convergence.

To avoid this lost update, we assume that the system should provide an undo feature in order to restore lost changes if the convergent state is not suitable for users. This undo feature is subject to many research efforts (Sun, 2002).

Writing *correct* transformation functions regarding the TP_1 property is not an easy task. The safety of the operational transformation approach relies on the correctness of transformation functions. If transformation functions do not satisfy TP_1 then the integration algorithm cannot ensure convergence of copies. Proving TP_1 property is error prone, time consuming and part of an iterative process. It is nearly impossible to do this by hand. In order to achieve this task, we used our VOTE environment (Imine et al., 2006) which is based on an automatic theorem prover. The input of this environment is exactly the definition of the transformation functions given in this paper. Describing our environment for verifying correctness of transformation functions is out of the scope of this paper, a more detailed description is available in (Imine et al., 2006; Imine et al., 2003).

In the following, a scenario illustrating how the So6 framework works is presented. It considers two users who are cooperating for writing an XML document. They are working in their own workspaces called $site_1$ and $site_2$, respectively. Each workspace contains a copy of the shared XML document. At the beginning both copies are identical.

The different steps of this scenario are summarized as follows:

$site_1$	$site_2$
op_1	op_3
op_2	
	commit (send op_3)
update (compute op'_3, op'_1, op'_2)	
commit (send op'_1, op'_2)	
	update (exec. op'_1, op'_2)

Users work concurrently to edit the document. The first user performs operations op_1 and op_2 while the second performs the operation op_3 . The states of the copies of the document taking into account these modifications are depicted by the Figure 6. After that, the second user *commits* their modifications i.e. the operation op_3 is sent to the timestamp. Later, the first user *updates* their workspace in order to integrate modifications published by the second user. During the *update*, the transformed operations op'_3, op'_1, op'_2 are calculated. At this step, only the operation op'_3 is locally executed. Then, the first user *commits* their modifications. During this step, op'_1 and op'_2 are sent to the timestamp. When the second user calls the *update* procedure, op'_1 and op'_2 are executed as-is on the local copy of the workspace $site_2$ (remember that

this user does not perform new operation). At the end of the execution, both copies of the document converge towards a unique value.

5 RELATED WORK

Configuration Management (CM) tools (Berliner, 1990) are widely used for asynchronous collaborative editing. Users work in parallel, produce data divergence and reconcile later using the Copy-Modify-Merge paradigm. Reconciliation is performed by tight cooperation between version manager and merge tools. When a reconciliation is required, i.e. usually when a user updates their workspace, the version manager provides those versions required by merge tools (Munson and Dewan, 1994). Merge is performed locally in the user workspace. Merge tools extract from different versions concurrent logs of operations using differentiation algorithms (Chawathe and Garcia-Molina, 1997). These differentiation algorithms are specific to data types. Finally, concurrent operations are merged using ad-hoc algorithms specific to data types. An XML merge tool such as DeltaXML (Fontaine, 2002) or XyDiff (Cobena et al., 2002) can be used in conjunction with CM tools for supporting collaboration on XML data. However, in this approach, several merge tools are used: one for file systems, another one for text files and another one for XML files. Each merge tool has its own merge algorithm. They might not be consistent together if they do not apply the same strategy. For example, in CVS, the merge tool used for text files relies on compensation contrary to the merge tool used at the file system level. Thus, whatever are the changes performed on a text file, they will always be merged into the new file version ; even conflicting changes are put in the text file – they are delimited with special mark-ups –. After the merging, a user can compensate what has been performed by the merge tool by editing the content of the text file. On the other side, the merge tool used at the file system level does not apply this principle. In the case it detects a conflict, the reconciliation process is stopped and the user is asked to solve the conflict. The operational transformation (OT) model is more general, more uniform and safer than the model used in CM tools. In the OT approach, the merge algorithm is shared by all transformation functions. It ensures convergence if underlying transformation functions ensure the TP_1 property. In this way, we can extend the reconciliation engine by adding new transformation functions without violating consistency.

Some propositions have been done in the OT model to work with XML data. Davis and al. (Davis et al., 2002) defined some transformation functions

for SGML. These functions present some similarities with our transformations for XML. However, Davis and al.'s functions do not verify the TP_1 property. Thus, using these transformation functions in our framework will not ensure convergence of copies of shared data.

In (Shen and Sun, 2002), Shen et al. proposed a framework similar to our So6 framework. The main difference is when a conflict occurs between two concurrent operations, the operation coming from the repository is cancelled, and the local operation is preserved. Firstly, this choice is not acceptable since cancelling an operation means losing some previously published work. Secondly, the authors do not provide any information concerning the editing of a tree structure such as an XML document. In parallel to our work, Ignat et al. (Ignat and Norrie, 2006) extended the Shen et al.'s approach to a tree structured document. The main idea is to distribute the log of operations through the tree. Thus, each node is associated with a log containing the operations performed on its content, insertion and deletion of child nodes. Using this model, they are able to use transformation functions defined for a linear structure such as the one proposed for a string of characters by Ressel et al. (Ressel et al., 1996). Their proposition constitutes an alternative to our approach.

IceCube (Kermarrec et al., 2001) is a generic approach for reconciling divergent copies of documents. It handles reconciliation as a constraints optimisation problem: the one of executing an optimal combination of concurrent changes. IceCube uses semantic constraints between operations that the reconciliation algorithm has to preserve. Basically, IceCube explores all possible combinations of concurrent operations and rejects all combinations violating defined constraints. This approach is interesting because, IceCube is looking for the combinations of concurrent operations that minimize conflicts of reconciliation. Maybe, on this point, the operational transformation approach will not find the optimal reconciliation. On the other hand, IceCube has some intrinsic drawbacks: Combinatorial explosion can occur during the first stage of reconciliation.

The Harmony project (Foster et al., 2005) is a generic framework for reconciling divergent copies. In this framework, the reconciliation process exploits schema of the structures being synchronized to achieve a better accuracy. This framework relies on a state-based approach which means three copies of the document – the two divergent copies and the common ancestor document – are required for reconciliation. As most state-based synchronisers, the goal of the reconciliation engine is to reduce

divergence between copies. However, convergence of copies is not achieved in all cases. If conflicting changes are detected between two copies, the conflicts are marked but the copies remain divergent. On the contrary, our framework will always ensure convergence of copies. Simply, in a case of conflicting changes, these changes will be transformed to be integrated as conflicting changes in the copies. This allows every participant to later resolve the conflict. We think sharing conflicts is useful, because sometimes the user informed about the conflict has no knowledge to resolve it.

6 CONCLUSION

We have presented the SO6 framework for supporting cooperative writing over documents. This framework relies on a theoretical model called operational transformation approach. Our framework is generic in the sense that it could be instantiated to manage multiple types of document. In order to illustrate these features, we explained how to enable cooperative writing of XML documents. This framework and the presented transformation functions are integrated in the SO6 revision management tool included in the LibreSource (<http://www.libresource.org/>) collaborative platform. This tool is able to reconcile copies of a file system containing text documents and XML documents.

If our framework ensures convergence, the convergence state may violate the DTD. For example suppose two users add concurrently a “title” element in an XML document. From the point of view of an ordered tree, two title nodes can appear under the root. However, from the point of view of the DTD, only one title is allowed. Finally, the SO6 framework is able to compute a convergence state, but this state may violate the DTD. This is clearly an open issue for the SO6 framework and for XML merge tools.

ACKNOWLEDGEMENTS

We wish to thank Claudia-Lavinia Ignat for her very valuable comments and suggestions which helped us to improve the presentation of this article.

REFERENCES

Berliner, B. (1990). CVS II: Parallelizing Software Development. In *Proceedings of the USENIX Winter Technical Conference*, pages 341–352.

Chawathe, S. S. and Garcia-Molina, H. (1997). Meaningful Change Detection in Structured Data. In *Proceedings of the ACM SIGMOD’97*, pages 26–37.

Cobena, G., Abiteboul, S., and Marian, A. (2002). Detecting Changes in XML Documents. In *Proceedings of the IEEE ICDE 2002*, pages 41–52.

Davis, A. H., Sun, C., and Lu, J. (2002). Generalizing Operational Transformation to the Standard General Markup Language. In *Proceedings of the ACM CSCW 2002*, pages 58–67.

Ellis, C. A. and Gibbs, S. J. (1989). Concurrency Control in Groupware Systems. 18:399–407.

Fontaine, R. L. (2002). Merging XML Files: A New Approach Providing Intelligent Merge of XML Data Sets. In *Proceeding of XML Europe 2002*.

Foster, J. N., Greenwald, M. B., Kirkegaard, C., Pierce, B. C., and Schmitt, A. (2005). Exploiting Schemas in Data Synchronization. In *Proceedings of DBPL 2005*, volume 3774 of *LNCS*.

Ignat, C.-L. and Norrie, M. C. (2006). Supporting Customised Collaboration over Shared Document Repositories. In *Proceedings of CAiSE 2006*, volume 4001 of *LNCS*.

Imine, A., Molli, P., Oster, G., and Rusinowitch, M. (2003). Proving Correctness of Transformation Functions in Real-Time Groupware. In *Proceedings of ECSCW 2003*, pages 277–293.

Imine, A., Rusinowitch, M., Oster, G., and Molli, P. (2006). Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence. *Theoretical Computer Science*, 351(2):167–183.

Kermarrec, A.-M., Rowstron, A., Shapiro, M., and Druschel, P. (2001). The IceCube Approach to the Reconciliation of Divergent Replicas. In *Proceedings of the ACM PODC 2001*, pages 210–218.

Molli, P., Oster, G., Skaf-Molli, H., and Imine, A. (2003). Using the Transformational Approach to Build a Safe and Generic Data Synchronizer. In *Proceedings of the ACM GROUP 2003*, pages 212–220.

Munson, J. P. and Dewan, P. (1994). A Flexible Object Merging Framework. In *Proceedings of the ACM CSCW’94*, pages 231–242, New York, NY, USA.

Ressel, M., Nitsche-Ruhland, D., and Gunzenhäuser, R. (1996). An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *Proceedings of the ACM CSCW’96*, pages 288–297.

Shen, H. and Sun, C. (2002). Flexible Merging for Asynchronous Collaborative Systems. In *Proceeding of the CoopIS 2002*, volume 2519 of *LNCS*, pages 304–321.

Suleiman, M., Cart, M., and Ferrié, J. (1998). Concurrent Operations in a Distributed and Mobile Collaborative Environment. In *Proceedings of the IEEE ICDE’98*, pages 36–45.

Sun, C. (2002). Undo as Concurrent Inverse in Group Editors. *ACM Transactions on Computer-Human Interaction*, 9(4):309–361.

Vidot, N., Cart, M., Ferrié, J., and Suleiman, M. (2000). Copies Convergence in a Distributed Real-Time Collaborative Environment. In *Proceedings of the ACM CSCW 2000*, pages 171–180.