Decentralized Concurrency Control for Real-time Collaborative Editors

Abdessamad Imine
LORIA INRIA Lorraine & Nancy University
Campus Scientifique, 54506 Vandœuvre-Lès-Nancy Cedex, France
imine@loria.fr

ABSTRACT

Real-time Collaborative Editors (RCE) provide computer support for modifying simultaneously shared documents, such as articles, wiki pages and programming source code, by dispersed users. Due to data replication, Operational Transformation (OT) is considered as the efficient and safe method for consistency maintenance in the literature of collaborative editors. Indeed, it is aimed at ensuring copies convergence even though the users's updates are executed in any order on different copies. Unfortunately, existing OT algorithms often fail to achieve this objective. Moreover, these algorithms have limited scalability with the number of users as they use vector timestamps to enforce causality dependency. In this paper, we present a novel framework for managing collaborative editing work in a scalable and decentralized fashion. It may be deployed easily on P2P networks as it supports dynamic groups where users can leave and join at any time.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*Collaborative Computing, Synchronous Interaction*

General Terms

Algorithms, Design, Theory

Keywords

Collaborative editors, Optimistic replication, Consistency, Operational Transformation, P2P, Real-time collaboration.

1. INTRODUCTION

Motivations. Real-time Collaborative Editors (RCE)[1] enable a group of users to edit simultaneously shared documents from physically dispersed sites that are interconnected by a computer network. To improve availability of data, each user has a local copy of the shared documents. In general, the collaboration is performed as follows: each user's updates are locally executed in nonblocking manner and then are propagated to other sites in order to be executed on other copies.

Although being distributed applications, RCE are specific in the sense that they must consider human factors. So, they are characterized by the following requirements:

- *High local responsiveness*: the system has to be as responsive as its single-user editors [1, 15, 16];
- *High concurrency*: the users must be able to concurrently and freely modify any part of the shared document at any time [1, 15];
- *Consistency*: the users must eventually be able to see a converged view of all copies [1, 15];
- *Decentralized coordination*: all concurrent updates must be synchronized in a decentralized fashion in order to avoid a single point of failure;
- *Scalability*: a group must be dynamic in the sense that users may join or leave the group at any time.

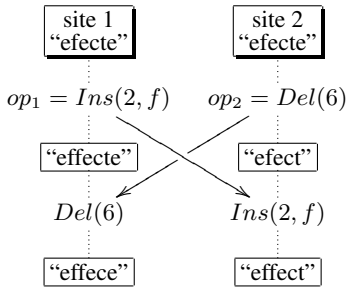
It is very difficult to meet these requirements when deploying RCE in networks with high communication latencies (*e.g.* Internet). Due to replication and arbitrary exchange of updates, consistency maintenance in a scalable and decentralized manner is a challenging problem. Traditional concurrency control techniques, such as (pessimistic/optimistic) locking and serialization, turned out to be ineffective because they may ensure consistency at the expense of responsiveness and loss of updates [1, 6, 15].

To illustrate this problem, consider the scenario in Figure 1.(a) where two users work on a shared document represented by a sequence of characters. These characters are addressed from 1 to the end of the document. Initially, both copies hold the string "efecte". User 1 executes operation $op_1 = Ins(2, f)$ to insert the character 'f' at position 2. Concurrently, user 2 performs $op_2 = Del(6)$ to delete the character 'e' at position 6. When op_1 is received and executed on site 2, it produces the expected string "effect". But, at site 1, op_2 does not take into account that op_1 has been executed before it and it produces the string "effece". The result at site 1 is different from the result of site 2 and it apparently violates the intention of op_2 since the last character 'e', which was intended to be deleted, is still present in the final string. It should be pointed out that even if a serialization protocol [1] was used to require that all sites execute op_1 and op_2 in the same order (*i.e.* a global order on concurrent operations) to obtain an identical result *effece*, this identical result is still inconsistent with the original intention of op_2.

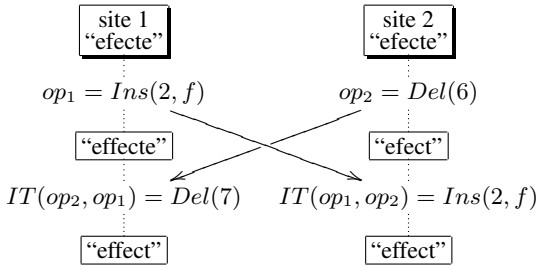
[1] *e.g.* Google Docs at `http://docs.google.com/`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NOTERE 2008 June 23-27, 2008, Lyon, France
Copyright 2008 ACM 978-1-59593-937-1/08/0003 ...$5.00.

Operational Transformation (OT). To maintain consistency, an OT approach has been proposed in [1]. In general, it consists of application-dependent transformation algorithm, called IT, so that for every possible pair of concurrent updates, the application programmer has to specify how to integrate these updates regardless of reception order. In Figure 1.(b), we illustrate the effect of IT on the previous example. At site 1, op_2 needs to be transformed in order to include the effects of op_1: $op_2' = IT((Del(6), Ins(2, f)) = Del(7)$. The deletion position of op_2 is incremented because op_1 has inserted a character at position 1, which is before the character deleted by op_2.



(a) Incorrect integration.



(b) Correct integration.

Figure 1: Serialization of concurrent updates

OT aims at ensuring consistency in a decentralized way without the need of any global order. It allows users to concurrently modify the shared document and exchange their updates in any order since the convergence of all copies must be ensured in all cases. Unfortunately, we have discovered that the most existing OT algorithms fail to guarantee consistency, because they contain bugs [2, 4]. Moreover, to our knowledge, the scalability requirement has never been dealt with in OT community research. All proposed OT frameworks rely on a fixed number of users during collaboration sessions. This is due to the fact that they use vector timestamps (to preserve causality relation) that do not scale well.

Contributions. In this paper, we propose a new framework for collaborative editing to address the weakness of previous OT works and to satisfy all requirements mentioned above. Our contributions are as follows:

1. Our framework supports an unconstrained collaborative editing work (without the necessity of central coordination). Using optimistic replication scheme, it provides simultaneous access to shared documents.

2. Instead of vector timestamps, we use a simple technique to preserve causality dependency. Our technique is minimal because only direct dependency information between updates is used. It is independent of the number of users and it provides high concurrency in comparison with vector timestamps.

3. Using OT approach, reconciliation of divergent copies is done automatically in a decentralized fashion.

4. Our framework can scale naturally thanks to our minimal causality dependency relation. In other words, it may be deployed easily in Peer-to-Peer (P2P) networks.

Outline. This paper is organized as follows: Section 2 gives the ingredients of our consistency model. Section 3 illustrates two kinds of OT algorithms that we use for transforming updates. Section 4 presents our concurrency control algorithm for managing collaborative editing works and an illustrative example on a collaboration session. Section 5 discusses related work and Section 6 summarizes contributions and future works.

2. CONSISTENCY MODEL

2.1 Model of Collaborative Editor

It is known that collaborative editors manipulate shared objects that own a linear structure [1, 14, 16]. This structure can be modelled by the abstract data type *list*. A list is a sequence of elements from a data type $\mathcal{E}$. This data type is only a parameter and can be instantiated by each type needed. For instance, an element may be regarded as a character, a paragraph, a page, an XML node, etc. In [16], it has been shown that this linear structure can be easily extended to a range of multimedia documents, such as MicroSoft Word and PowerPoint documents.

We assume that the list state can only be modified by the following primitive operations:

- $Ins(p, e, \omega)$ where p is the insertion position, e the element to be added at position p and ω is the sequence of positions that contains all different positions occupied by e during the transformation process (see Section 3.1);
- $Del(p)$ which deletes the element at position p.

Hence the set of operations is defined as follows:

$$\mathcal{O} = \{Ins(p, e, \omega) | e \in \mathcal{E}, p \in \mathbb{N} \text{ and } \omega \in \mathbb{N}^*\} \cup \{Del(p) | p \in \mathbb{N}\} \cup \{Nop\}$$

where Nop is the idle operation that has null effect on the list state. Each user's site has a local state l that is altered only by insertion and deletion operations. The initial state, denoted l_0, is the same for all sites. Let $\mathcal{L}$ be the set of list states. We use the function, $Do : \mathcal{O} \times \mathcal{L} \rightarrow \mathcal{L}$, for computing the resulting state l' when a user applies operation o to state l: $Do(o, l) = l'$.

We define an $update$ as a quadruple (c, r, a, o) where c is the identity of the collaborator (or the user) issuing the update[2], $r \in \mathbb{N}$ is its serial number, a is the identity of the preceding update[3], and finally $o \in \mathcal{O}$ is the operation to be executed on the list state. Note that if a is $null$ then the update does not depend on any other update. The projections $u.c$, $u.r$, $u.a$, and $u.o$ will be used to denote the corresponding components of update u. Let $\mathcal{U}$ denote the set of updates and we use u, u', u_1, u_2, ..., to denote updates. The concatenation of $u.c$ and $u.r$ is defined as the update identity of u. Notation $[u_1; u_2; \ldots; u_n]$ represents an update sequence. Applying an update sequence to a

[2]The set of collaborator identities is assumed totally ordered by relation $\leq$.

[3]According to the dependency relation described in Section 2.2.

list l is recursively defined as follows: (i) $Do([], l) = l$ where [] is the empty sequence and; (ii) $Do([u_1.o; u_2.o; \ldots; u_n.o], l) = Do(u_n.o, Do(\ldots, Do(u_2.o, Do(u_1.o, l))))$. Two update sequences seq_1 and seq_2 are *equivalent*, denoted by $seq_1 \equiv seq_2$, iff $Do(seq_1, l) = Do(seq_2, l)$ for all lists l.

A *log* buffer is an update sequence which is maintained on every site in order to keep all executed updates. Given a log L, $L[i]$ denotes the i-th update of L; $|L|$ is the length of L; $L[i, j]$ is the sub-log of L ranging from its i-th to j-th updates with $0 < i \leq j \leq n - 1$ such that $n = |L|$. Let $\mathcal{H}$ be the set of logs.

Furthermore, we suppose that sites are interconnected by a reliable network. The propagation of updates is based on an epidemic style of communication.

2.2 Causal Dependency Relation

Given a log, an update may *depend on* previous updates according to the execution order. In other words, the effect of an update may be "influenced" by some previous updates. Tracking this dependency inside a log enables us to identify updates that must be executed on all sites according to the same order. As a long established convention in collaborative editors [1, 14], the vector timestamps are used to determine the happened-before and concurrent relations between updates. Unfortunately, these techniques do not scale well, since each timestamp is a vector of integers with a number of entries equal to the number of sites. Instead we propose a minimal dependency relation which is not dependent of the number of users, and accordingly, allows for dynamic groups. Studying the semantics of linear object (modified by insertion and deletion operations) allows us to provide the following dependency relation:

DEFINITION 2.1. ***(Causal dependency relation)*** *Let L be a log where $L[i] = u_i$ and $L[j] = u_j$ with $j = i + 1$. We define the transitive relation $\xrightarrow{s}$ on L as follows. We say that $u_i \xrightarrow{s} u_j$ iff one of the following conditions holds:*

1. $u_i.o = Ins(p, e, \omega)$, $u_j.o = Ins(p, e', \omega')$ *and* $u_i.c \leq u_j.c$; $(\mathbf{df_1})$
2. $u_i.o = Ins(p, e, \omega)$, $u_j.o = Ins(p + 1, e', \omega')$ *and* $u_j.c \leq u_i.c$; $(\mathbf{df_2})$
3. $u_i.o = Ins(p, e, \omega)$, *and* $u_j.o = Del(p)$. $(\mathbf{df_3})$

Otherwise, u_i and u_j are independent *(or* concurrent*) and we denote them by $u_i \parallel u_j$.*

Two updates of a log are independent means that they can be executed out-of-order. According to conditions df_i $(i = 1, 2, 3)$, all updates may only depend on insertion updates. When the added elements are adjacent (at positions p and/or $p + 1$), the insertion updates may be dependent. Deleting an element depends on the update that has inserted this element. Thus, there is no dependency between delete updates and they can be executed among them in any order. It is easy to show that our causal relation builds a dependency tree. In this case, each update has only to store the update identity whose it directly depends on.

2.3 Criteria Consistency

A stable state in a RCE is achieved when all generated updates have been performed at all sites. Our replication scheme must ensure the following criteria:

DEFINITION 2.2. ***(Consistency Model)*** *A RCE is* consistent *iff it satisfies the following properties:*

1. Dependency preservation: *if $u_1 \xrightarrow{s} u_2$ then u_1 is executed before u_2 at all sites.*
2. Convergence: *when all sites have performed the same set of updates, the copies of the shared document are identical.*

To establish a causal dependency between updates, we use the relation given in Definition 2.1. This relation is minimal because every update has to know only the identity of the update it depends on directly. Nevertheless, it remains one problem to be solved: how to serialize concurrent updates in order to achieve the convergence? The solution of this problem is given in the next section.

3. OPERATIONAL TRANSFORMATION

The basic idea of OT is to perform any local update as soon as it is generated for high local responsiveness. Remote updates are transformed against concurrent updates that have been executed locally before its execution. Many collaborative applications are based on the OT approach such as Joint Emacs [11] (a groupware based on text editor Emacs), CoWord [16] (a collaborative Microsoft word processor) and CoPowerPoint [16] (a real-time collaborative multimedia slides creation and presentation system).

In our work, we use three kinds of transformation [12, 14]: Inclusive Transformation (IT), Exclusive Transformation (ET) and Permutation ($PERM$). In the following, we will present our IT and ET algorithms. For more details, see [2, 5].

3.1 Inclusive transformation

Principle. Intuitively, the *inclusive transformation* $IT(u, u')$ transforms u against u' in order to *include the effect* of u' in u. The transformed form of u is then executed after u'. As in [15], $IT(u, u')$ is defined iff u and u' are *concurrent* and *defined on the same* list state. To detect concurrency between updates we use the relation given in Definition 2.1. It should be noted that we have redefined the insertion operation by adding a new parameter (*i.e.* ω) [3,5]. In fact, this parameter is used as a *stack* to store all different positions occupied by an element during the transformation process. Each time an insert update is transformed we push the last position before transformation in the ω parameter. On the other hand, when an insert update is generated its ω is empty. For instance, consider u_1 and u_2 such that $u_1.o = Ins(3, x, \epsilon)$ and $u_2.o = Del(1)$ where ϵ denotes the empty stack: then $IT(u_1, u_2) = u_1'$ with $u_1'.o = Ins(2, x, [3])$.

Given an insert update i_j and a delete update d_j such that $i_j.o = Ins(p_j, e_j, \omega_j)$ and $d_j.o = Del(p_j)$ for $j \in \{1, 2\}$. All cases for our IT are given in Algorithm 1. Instead of single positions, we compare position words using lexicographic order $\prec$ to transform a pair of insert updates (lines 4-6). When two updates insert two elements at the same word position (they are in conflict), a choice has to be done: which element must be inserted before the other? The solution that is generally adopted consists in associating a priority to each insert update (*e.g.*, the user identifier [11, 14]). In our IT function, when a conflict occurs, the element of an insertion update whose user identifier c is the highest is inserted before the other. On the other hand, when two updates delete at the same position, IT returns the idle update whose the operation parameter is Nop[4]. The remaining cases of IT are quite simple. For more details on our IT algorithm, see [5].

Transformation properties. Using an IT algorithm requires us to satisfy two properties $TP1$ and $TP2$ in order to ensure convergence [11]. For all u, u_1 and u_2 pairwise concurrent updates with $u_1' = IT(u_1, u_2)$ and $u_2' = IT(u_2, u_1)$:

[4]If $u.o = Nop$ then $IT(u, u') = u$ and $IT(u', u) = u'$.

```
1: IT(u1, u2) = u1'
2: u1' ← u1
3: Choice of u1 and u2
4:    Case: u1 = i1 and u2 = i2
5:       if (p2ω2 ≺ p1ω1 or (p2ω2 = p1ω1 and u2.c < u1.c))
         then u1'.o ← Ins(p1 + 1, e1, p1ω1)
6:       end if
7:    Case: u1 = i1 and u2 = d2
8:       if (p2 < p1) then u1'.o ← Ins(p1 − 1, e1, p1ω1)
9:       else if (p2 = p1) then u1'.o ← Ins(p1, e1, p1ω1)
10:      end if
11:   Case: u1 = d1 and u2 = i2
12:      if (p2 ≤ p1) then u1'.o ← Del(p1 + 1)
13:      end if
14:   Case: u1 = d1 and u2 = d2
15:      if (p2 < p1) then u1'.o ← Del(p1 − 1)
16:      else if (p2 = p1) then u1'.o ← Nop
17:      end if
18: end choice
19: return u1'
```

Algorithm 1: Inclusive transformation.

- **TP1**: $[u_1 ; u_2'] \equiv [u_2 ; u_1']$.
- **TP2**: $IT(IT(u, u_1), u_2') = IT(IT(u, u_2), u_1')$.

Property $TP1$ defines a *state identity* and ensures that if u_1 and u_2 are concurrent, the effect of executing u_1 before u_2 is the same as executing u_2 before u_1. This property is necessary but not sufficient when the number of sites is greater than two. Property $TP2$ defines an *update identity* and ensures that transforming u along equivalent and different update sequences will give the same update.

Properties $TP1$ and $TP2$ are sufficient to ensure the convergence for *any number* of concurrent operations which can be executed in *arbitrary order* [9,11]. Accordingly, by these properties, it is not necessary to enforce a global total order between concurrent updates because data divergence can always be repaired by operational transformation.

TP2 puzzle. However, although in theory the OT approach is able to achieve convergence in the presence of arbitrary transformation orders, linear objects (such as text or ordered XML tree) still represent a serious challenge for the application of the OT approach. Indeed, all proposed IT algorithms [2,4] for these datatypes fail to meet the property $TP2$, leading inevitably to data divergence situations. The "killer" scenario for these algorithms always consists of two insertion updates and a delete update, like the one depicted in Figure 2 (where for the simplicity we show only the operation parameter of updates and the origin position is 0). At site 2 (resp. site 3), u_1 is recursively transformed against the sequence $[u_2; u_3']$ (resp. $[u_3; u_2']$). This scenario is termed $TP2$ *puzzle* [14]. In Figure 2 we show how our IT algorithm achieves data convergence even though it fails to satisfy $TP2$ (*e.g.* $u_1'.o \neq u_1''.o$ as stacks [2.3] and [4.3] are different).

In [3, 5], we proposed a weakened form for $TP2$, called $C'2$[5], which is necessary when using only n concurrent updates defined on the same state ($n \geq 2$). But there are some cases involving causality relation based on vector timestamps where $C'2$ fails to ensure convergence. As a matter of fact we have showed that $C'2$ is sufficient for achieving convergence, if we use our causal dependency relation (see Definition 2.1) and we maintain logs in a canonical form (see Definition 4.2).

[5]It does not impose update identity like $TP2$. For more details, see [5].

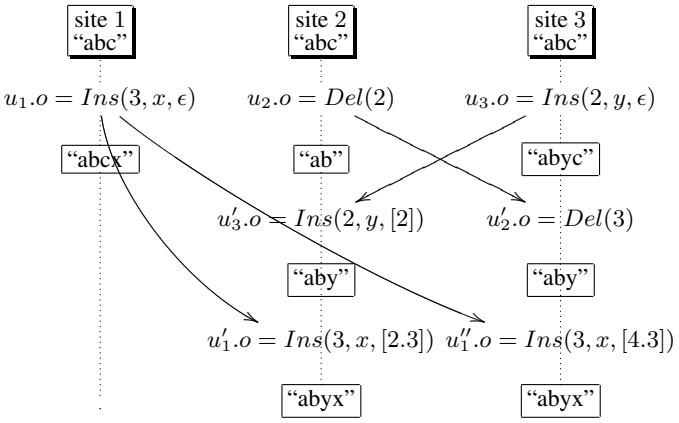


Figure 2: Correct execution of $TP2$ puzzle.

3.2 Exclusive transformation

Let $[u_1; u_2]$ be an update sequence so that u_2 is defined on the state produced by u_1. The *exclusive transformation* $ET(u_2, u_1)$ enables us to *exclude the effect* of u_1 from u_2 as if u_2 had not been executed after u_1. As a simple example, consider the scenario in Figure 3. Given u_1 defined in state "abc" and u_2 defined in state "ac" (produced by u_1): $ET(u_2, u_1) = u_2'$ with $u_2'.o = Ins(1, y, \epsilon)$ which is exactly the form of u_2 as defined relative to the state "abc". It should be noted that in our work $ET(u_2, u_1)$ is defined iff $u_1 \not\xrightarrow{s} u_2$. Using ET algorithm enables updates to be executed out-of-order.

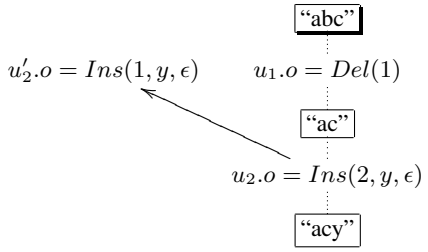


Figure 3: Exclusive transformation

Let i_j and d_j be two updates such that $i_j.o = Ins(p_j, e_j, \omega_j)$ and $d_j.o = Del(p_j)$ with $j \in \{1, 2\}$. All different cases of our ET are given in Algorithm 2. In case where u_1 and u_2 have the same insertion positions, then the element added by u_1 must precede (or is before) the one added by u_2. If the relation between their position words or user identities reflects this precedence, $ET(u_1, u_2)$ returns u_1 as u_2 has no effect on the insertion position of u_1. On the other hand, when u_2 is before u_1 and their word positions (or their user identities) give the same order, then we decrement the insertion position of u_1 for excluding the u_2's effect (lines 8-15). To exclude the effect of a delete update u_2 from another delete update u_1, we increment the position of u_1 (as if u_2 had not been executed before u_1) when it has erased an element after the one of u_2 (lines 26-27). Functions Tp and Tl are used to return respectively the top position and the resulting stack without the top position for every stack. Note that ET returns "Undefined" when there is a causal dependency (*i.e.* $u_2 \xrightarrow{s} u_1$).

In [2], we have showed that our algorithms IT and ET are reversible, in the sense that transforming in both directions preserves the data convergence.

```
1: ET(u1, u2) = u1'
2: u1' ← u1
3: Choice of u1 and u2
4:   Case: u1 = i1 and u2 = i2
5:     α1 ← p1.ω1
6:     α2 ← p2.ω2
7:     if (p1 = p2) and ((α1 ≻ α2) or
         (α1 = α2 and u1.c ≥ u2.c))
       then return "Undefined"
8:     else if (p1 = p2 + 1)
9:           then if (ω1 ≠ ε)
10:                then if (ω1 ≻ α1) or (ω1 = α1
                          and u1.c > u2.c)
                        then
                         u1'.o ← Ins(p1 − 1, e1, Tl(ω1))
11:                     else return "Undefined"
12:          else if (u1.c > u2.c)
                  then u1'.o ← Ins(p1 − 1, e1, ω2)
13:                else return "Undefined"
14:          else if (p1 > p2 + 1)
                  then u1'.o ← Ins(p1 − 1, e1, Tl(ω1))
15:    endif

16:  Case: u1 = i1 and u2 = d2
17:    if (ω1 ≠ ε)
18:    then if (p1 = p2 and p1 = Tp(ω1))
            then u1'.o ← Ins(p1, e1, Tl(ω1))
19:          else if (p1 > p2)
                  then u1'.o ← Ins(p1 + 1, e1, Tl(ω1))
20:    else if (p1 > p2) then u1'.o ← Ins(p1 + 1, e1, ω1)
21:    endif

22:  Case: u1 = d1 and u2 = i2
23:    if (p1 ≥ p2 + 1) then u1'.o ← Del(p1 − 1)
24:    else if (p1 = p2) then return "Undefined"
25:    endif

26:  Case : u1 = d1 and u2 = d2
27:    if (p1 ≥ p2) then u1'.o ← Del(p1 + 1)
28: end choice
29: return u1'
```

Algorithm 2: Exclusive transformation.

3.3 Permutation of updates

Sometimes it is necessary to *reorder* (or *permute*) updates in an log without affecting the resulting state of this log [10, 12]. So we use function $PERM$ that enables an update to be moved back to the past in order to simulate it as being executed first.

DEFINITION 3.1. ***(Permuting updates)*** *We define function* $PERM : \mathcal{U} \times \mathcal{U} \rightarrow \mathcal{H}$ *as follows: if* $u_2 \overset{s}{\not\rightarrow} u_1$ *then* $PERM(u_1, u_2) = [u_1'; u_2']$ *such that (i)* $u_1' = ET(u_1, u_2)$*; and, (ii)* $u_2' = IT(u_2, u_1')$.

Note that the function $PERM$ is not defined when $u_2 \overset{s}{\rightarrow} u_1$. Given the reversibility of IT and ET, replacing $[u_2; u_1]$ by $PERM(u_1, u_2)$ means that we can get an equivalent sequence where another form of u_1 could be executed first. As a simple example, consider the scenario in Figure 4: $PERM(u_2', u_1) = [u_2; u_1']$ with $u_2 = ET(u_2', u_1)$ and $u_1' = IT(u_1, u_2)$ and so that $[u_2; u_1'] \equiv [u_1; u_2']$. By using the ω parameter, we ensure that our ET algorithm "goes backward" in a deterministic way as ω contains the previous positions.

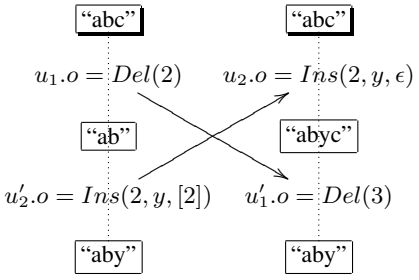


Figure 4: Permutation of updates.

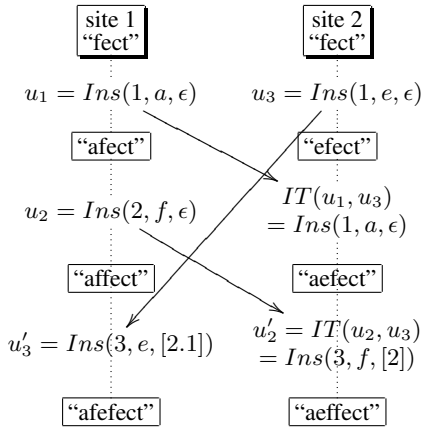
3.4 Partial concurrency situation

Two concurrent updates u_1 and u_2 are said to be *partially concurrent* iff u_1 and u_2 are generated from two different states [12]. In case of partial concurrency situation, naively applying the inclusive transformation IT may lead to data divergence.

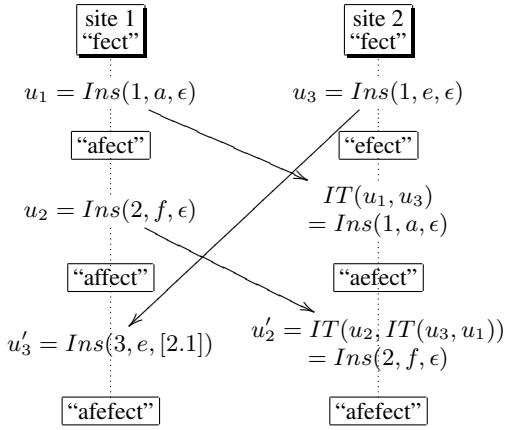
Consider two users trying to correct the word "*fect*" as in Figure 5.(a). User 1 generates u_1 and u_2. User 2 concurrently generates u_3. We have $u_1 \overset{s}{\rightarrow} u_2$ and $u_1 \parallel u_3$ (as u_1 did not see the effect u_3 and vice-versa). However, u_2 and u_3 are partially concurrent as they are generated on different text states. At site 1, u_3 has to be transformed against the sequence $[u_1; u_2]$, *i.e.* $u_3' = IT(IT(u_3, u_1), u_2) = Ins(3, e, [2.1])$. The execution of u_3' gives the word "*afefect*". At site 2, transforming u_1 against u_3 gives $u_1' = u_1 = Ins(1, a, \epsilon)$ and transforming u_2 against u_3 results in $u_2' = Ins(3, f, [2])$ whose the execution leads to the word "*aeffect*" which is different from what is obtained at site 1. This divergence situation is due to wrong transformation of u_2 at site 2. Indeed, $IT(u_2, u_3)$ requires that u_2 and u_3 be concurrent and defined on the same state. However, u_3 is generated on "*fect*" while u_2 is generated on "*afect*".

To overcome this partial concurrency problem, we can use the solution adopted by most existing OT integration algorithms, such that SOCT2 [12] and GOTO [14], that impose to reorder local log into equivalent one before transforming u_2. Indeed, to integrate u_2, using function like $PERM$, the local log $L = [u_3; u_1']$ must be reordered into a concatenation of two sequences L_h and L_c, where L_h contains all updates that happened before u_2 and L_c includes updates that are concurrent with u_2. In our case, $L_h = [u_1]$ and $L_c = [u_3']$ with $u_3' = IT(u_3, u_1)$. Next, u_2 is just inclusively transformed against L_c (see Figure 5.(b)).

In general, the above solution is very expensive because it may require several permutations inside local log in order to integrate a remote update. In our approach, we deal with the partial concurrency problem by avoiding the log reorganization. This is done by directly deduce the form to be executed of the remote update from its precedent update inside the local log. In the previous example, we have $u_1 \overset{s}{\rightarrow} u_2$ at site 1. As this dependency must be preserved at site 2, we can say that $u_1' \overset{s}{\rightarrow} u_2'$ and $u_2'.o = u_2.o$. For more details on our technique for integrating remote updates, see section 4.



(a) Wrong application of IT.



(b) Correct application of IT.

Figure 5: Partial concurrency.

4. CONCURRENCY CONTROL ALGORITHM

We have designed a new algorithm for managing all concurrent interactions occurring in RCE. This algorithm relies on (i) the replication of the shared documents in order to provide data access without constraints, and; (ii) the consistency model based on causal dependency. In [2], we showed that our algorithm satisfies the criteria consistency given in Definition 2.2.

4.1 Reordering logs

In this section we will give a class of logs which allow us to build transformation paths leading to data convergence. Firstly, we show how to determine the causal dependency between two updates belonging to the same log.

DEFINITION 4.1. *Let $L = [u_1; u_2; \ldots; u_n]$ be a log. We say that $u_i \xrightarrow{s} u_j$, where $i, j \in \{1, \ldots, n\}$ and $i < j$, iff there exists either (i) $u' = ET^*(u_j, L[i+1, j-1])$ such that $u_i \xrightarrow{s} u'$; or, (ii) an update u_k with $i < k < j$ and $u_i \xrightarrow{s} u_k \xrightarrow{s} u_j$.*

Function ET^* denotes the exclusive transformation of an update against a log. It is defined recursively as follows:

(i) $ET^*(u, []) = u$;

(ii) $ET^*(u, [u_1; \ldots; u_{n-1}; u_n]) =$
$ET^*(ET(u, u_n), [u_1; \ldots; u_{n-1}])$.

As all updates only depend on insertion updates, we are interested in the following logs:

DEFINITION 4.2. *A log L is* canonical *iff L is the concatenation of two sub-logs L_i and L_d such that L_i contains insertion updates and L_d contains deletion updates.*

Note that empty logs and logs containing only insert (resp. delete) updates are also canonical. Using our causal dependency relation, we can build canonical logs by applying one or several update permutations.

4.2 Control procedure

In our approach, a collaborative editor consists of a group of N sites (where N is variable in time) starting a collaboration session from the same initial state l_0. Each site stores all executed updates in canonical log L (*i.e.* insertions before deletions). Our control concurrency procedure is given in Algorithm 3.

```
1:  Main:
2:  INITIALIZATION
3:  while not aborted do
4:     if there is an input o then
5:        GENERATE_UPDATE(o)
6:     else
7:        RECEIVE_UPDATE
8:        INTEGRATE_REMOTE_UPDATES
9:     end if
10: end while

11: INITIALIZATION:
12: Q ← []
13: L ← []
14: l ← l_0
15: r ← 1
16: c ← Identification of local user

17: GENERATE_UPDATE(o):
18: l ← Do(o, l)
19: u ← (c, r, null, o)
20: u' ← COMPUTEBF(u,L)
21: L ← CANONIZE(u,L)
22: broadcast u' to other users

23: RECEIVE_UPDATE:
24: if there is an update u from a network then
25:    Q ← Q + u
26: end if

27: INTEGRATE_REMOTE_UPDATE:
28: if there is u in Q that is causally-ready then
29:    Q ← Q − u
30:    u' ← COMPUTEFF(u,L)
31:    l ← Do(u'.o, l)
32:    L ← CANONIZE(u',L)
33: end if
```

Algorithm 3: Control Concurrency Algorithm

Generation of local update. When an operation o is locally generated, it is immediately executed on its generation state, namely $l = Do(L, l_0)$. Once the update $u = (c, r, null, o)$[6] is formed,

[6] $null$ means that u does not depend on any update.

function COMPUTEBF(u,L) is called (see Algorithm 4) in order to compute the minimal generation context of u. In other words, instead of considering u as being dependent of all L's updates, our procedure reduces this context by excluding as much as possible some updates of L by means of exclusive transformation ET. To well understand this step, consider the set $Dep(u) = \{u" \in L | u" \xrightarrow{s} u\}$ which is built according to Definition 4.1. If $Dep(u) = \emptyset$ then the new update u' is not dependent of L. Otherwise, $Dep(u) \neq \emptyset$, u' must be executed on all sites after the updates of $Dep(u)$. In this case, $u'.a$ contains only the identity of the direct preceding update plus the dependency form fd_j (see Definition 2.1).

```
1: COMPUTEBF(u,L) : u'
2: u' ← u
3: for (i = |L| − 1; i ≥ 0; i − −) do
4:    if u' is not dependent of L[i] then
5:       u' ← ET(u', L[i])
6:    else
7:       u'.a = (L[i].p, L[i].k, fd_j) {fd_j with j = 1, 2, or 3
         according to the dependency form}
8:       return u'
9:    end if
10: end for
11: return u'
```

Algorithm 4: Detection of causal dependency

Integrating u after L may result in not canonical log. To transform $[L; u]$ in canonical form, we use function CANONIZE(u,L) that is given in Algorithm 5. It relies on function $PERM$ to applying successive permutations. Finally, the update u' (the result of COMPUTEBF) is propagated to all sites in order to be executed on other copies of the shared document.

```
1: CANONIZE(u,L) : L'
2: L' ← [L; u]
3: i ← |L'| − 1
4: while L' is not canonical do
5:    < L'[i − 1], L'[i] >← PERM(L'[i], L'[i − 1])
6:    i ← i − 1
7: end while
8: return L'
```

Algorithm 5: Canonizing logs.

Integration of remote update. Each site has the use of queue Q to store the remote updates coming from other sites. An update u generated on site i is added to Q when it arrives at site j (with $i \neq j$). To preserve the causality dependency, u is extracted from the queue when it is *causally-ready*: if $u" \xrightarrow{s} u$ then $u"$ has been already integrated on site j. Next, function COMPUTEFF(u,L) (see Algorithm 6) is called in order to compute the transformed form u' to be executed on current state l. Let n be the length of L. Two cases are to be considered: (i) if $u.a = null$ then u is concurrent to all updates of L; (ii) if $u.a \neq null$ then there exists update $L[k]$ (with $k \in \{0, \ldots, n-1\}$) whose u depends on.

In case (ii), unlike the others integration algorithms based on OT approach, our algorithm does not require to reorganize L in two sub-logs containing respectively precedent updates and concurrent updates with respect to u. Instead, only the parameter $u.o$ is updated with respect to $L[k]$ and the dependency form fd_j contained in $u.a$ (see Definition 2.1). The obtained update is next inclusively transformed against $L[k+1, n-1]$. Finally, the transformed form of u, namely u', is executed on the current state and function CANONIZE is called in order to turn again $[L; u']$ in canonical form.

```
1: COMPUTEFF(u,L) : u'
2: u' ← u
3: j ← −1
4: if u'.a ≠ null then
5:    Let L[j] be the update whose u' depends on (j ∈
      {0, ..., |L| − 1})
6:    Modify u'.o with respect to L[j].o and the dependency form
7: end if
8: for (i = j + 1; i ≤ |L| − 1; i + +) do
9:    u' ← IT(u', L[i])
10: end for
11: return u'
```

Algorithm 6: Transforming an update against a log.

4.3 Illustrative example.

To highlight the features of our concurrency control algorithm, we present a slightly more complicated example in Figure 6, where the arrows show the integration order. Three sites start a collaboration session with the same initial state l_0 ="abc". They generate respectively three concurrent updates: $u_1.o = Del(2)$, $u_2.o = Ins(3, x, \epsilon)$ and $u_3.o = Ins(2, y, \epsilon)$. After integrating u_1, u_2 and u_3, site 1 generates $u_4.o = Del(1)$. Site 2 generates $u_5.o = Del(1)$ after u_2 and u_1. As for site 3, it generates $u_6.o = Ins(3, z, \epsilon)$ just after u_3, u_2 and u_1. At stable state (*i.e.* all generated updates have been executed at all sites), three sites converge to the same text "yzxc".

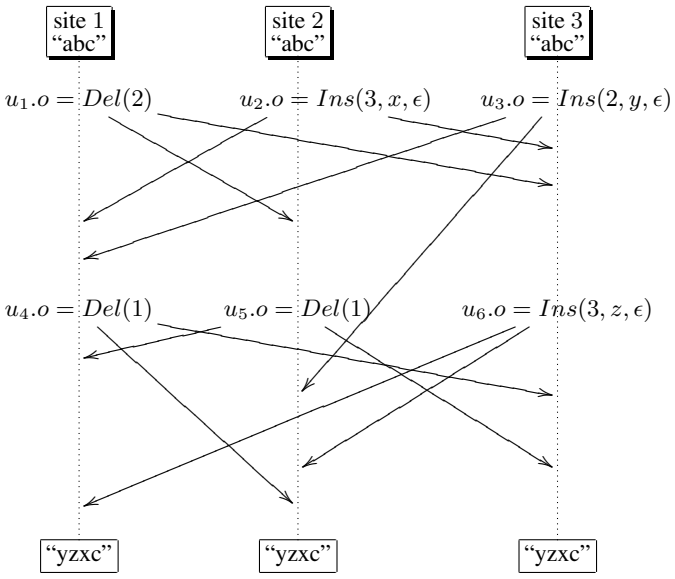


Figure 6: Collaboration scenario between three sites.

We describe the update integration in three steps:

Step 1. Initially, the log of each site is empty ($L_i^0 = []$ for $i \in 1, 2, 3$). In this case, u_1, u_2 and u_2 are considered as concurrent as they do not depend on any update.

At site 1, after u_1 is locally executed, the text becomes l_1^1 ="ac" and the new log is $L_1^1 = [u_1]$. As u_2 and u_1 are concurrent, the transformed form $u_2' = IT(u_2, u_1)$ (with $u_2'.o = Ins(2, x, [3])$ is executed and it produces text l_1^2 ="axc". The log $[L_1^1; u_2']$ is

not canonical; it is next transformed in $L_1^2 = [u_2; u_1]$. When u_3 arrives, it is transformed against L_1^2 in u'_3 (with $u'_3.o = Ins(2, y, [2])$) that leads to text l_1^3 ="ayxc". The canonization of $[L_1^2; u'_3]$ results in $L_1^3 = [u_2; u_3; u'_1]$ with $u'_1.o = Del(3)$.

At site 2, the local execution of u_2 produces text l_2^1 ="abxc" and the local log becomes $L_2^1 = [u_2]$. The integration of u_1 produces $u'_1 = IT(u_1, u_2) = u_1$. Text l_2^2 ="axc" and log $L_2^2 = [u_2; u_1]$ are the results of the u_2's execution.

At site 3, after u_3 is locally executed, the text becomes l_3^1 ="aybc" such that log $L_3^1 = [u_3]$. When u_2 arrives, it is first transformed in $u'_2 = IT(u_2, u_3)$ (with $u'_2.o = Ins(4, x, [3])$) and next it is executed to get text l_3^2 ="aybxc" and log $L_3^2 = [u_3; u'_2]$. Finally, u_1 is transformed against L_3^2 and results in u'_1 (with $u'_1.o = Del(3)$) that leads to text l_3^3 ="ayxc" and log $L_3^3 = [u_3; u'_2; u'_1]$.

Step 2. Three concurrent updates u_4, u_5 and u_6 are generated respectively on sites 1, 2 and 3. They are propagated as follows.

At site 1, the execution of u_4 after $L_1^3 = [u_2; u_3; u'_1]$ (with $u'_1.o = Del(3)$) produces text l_1^4 ="yxc". To broadcast u_4 to other sites with minimal generation context, it is exclusively transformed against L_1^3 (by using function COMPUTEBF) to produce the same update u_4 that is propagated to all sites 2 and 3. Thus, u_4 is concurrent to L_1^3. The obtained log is $L_1^4 = [u_2; u_3; u'_1; u_4]$.

At site 2, u_5 is executed after $L_2^2 = [u_2; u_1]$ and it produces text l_2^3 ="xc". The same update is propagated to other sites as $u_5 = ET^*(u_5, L_2^2)$ (u_5 is concurrent to L_2^2). The obtained log is $L_2^3 = [u_2; u_1; u_5]$.

At site 3, u_6 is generated after $L_3^3 = [u_3; u'_2; u'_1]$ (with $u'_2.o = Ins(4, x, [3]$ and $u'_1.o = Del(3)$) and its execution leads to text l_3^4 ="ayzxc". The result of $ET^*(u_6, [u'_2; u'_1])$ is the same as u_6 (with $u_6.o = Ins(3, z, \epsilon)$). It is clear that $u_3 \xrightarrow{s} u_6$ as $u_3.o = Ins(2, y, \epsilon)$ (by using the second dependency form df_2). Therefore, u_6 must be executed after u_3 at all sites. The canonization of $[L_3^3; u_6]$ results in $L_3^4 = [u_3; u'_2; u_6; u''_1]$ where $u''_1.o = Del(4)$.

Step 3. In this step we will present how to execute u_4, u_5 and r_6 on other sites.

At site 1, u_5 is concurrent to $L_1^4 = [u_2; u_3; u'_1; u_4]$. In this case, it is transformed against L_1^4 and results in idle update u'_5 (with $u_5.o = Nop$). The obtained text is $l_1^5 = l_1^4$ ="yxc" and the obtained log is $L_1^5 = [u_2; u_3; u'_1; u_4; u'_5]$. When u_6 arrives, it is causally ready ($u_3 \xrightarrow{s} u_6$) as u_3 was already integrated. Thus, u_6 is transformed only against $[u'_1; u_4; u'_5]$ to produce u'_6 (with $u'_6.o = Ins(2, z, [3.3])$. Its execution leads to text l_1^6 ="yzxc". The canonization of $[L_1^5; u'_6]$ results in $L_1^6 = [u_2; u_3; u_6; u''_1; u_4; u'_5]$ with $u''_1.o = Del(4)$.

At site 2, when u_3 arrives, it is transformed against $L_2^3 = [u_2; u_1; u_5]$ and to produce u'_3 such that $u'_3.o = Ins(1, y, [2.2])$. The execution of u'_3 leads to text l_2^4 ="yxc" and canonical log $L_2^4 = [u_2; u_3; u'_1; u_5]$ with $u'_1.o = Del(3)$. As u_6 is causally ready, it is first transformed only against $[u'_1; u_5]$ to become u'_6 (with $u'_6.o = Ins(2, z, [3.3])$) whose the execution leads to text l_2^5 ="yzxc" and canonical log $L_2^5 = [u_2; u_3; u_6; u''_1; u_5]$ with $u''_1.o = Del(4)$. As seen in step 2, u_4 do not depend on any update. It is then transformed against L_2^5 to produce an idle update u'_4. The final text is $l_2^6 = l_2^5$ ="yzxc" and the final log is $L_2^6 = [u_2; u_3; u_6; u''_1; u_5; u'_4]$.

At site 3, the current text is l_3^4 ="ayzxc" and the current log is $L_3^4 = [u_3; u'_2; u_6; u''_1]$. When u_4 arrives, it is transformed against L_3^4. This leads to text l_3^5 ="yzxc" and log $L_3^5 = [u_3; u'_2; u_6; u''_1; u'_4]$ with $u'_4.o = Del(1)$. In the same way, when u_5 arrives it is transformed against L_3^5. The final text is $l_3^6 = l_3^5$ ="yzxc" and the final log is $L_3^6 = [u_3; u'_2; u_6; u''_1; u'_4; u'_5]$ with $u'_5.o = Nop$.

5. RELATED WORK

Since the $TP2$ puzzle has been discovered [12, 14], several works have tried to address this problem. These works may be categorized in two approaches.

The first one tries to avoid the $TP2$ puzzle scenario. This is done by constraining the communication among users in order to restrict the space of possible execution order. For instance, the SOCT4 algorithm [17] uses a sequencer, associated with a deferred broadcast and a sequential reception, to enforce a continuous global order on updates. This global order can also be obtained by using an undo/do/redo scheme like in GOT [15]. These ensure the data convergence but they cannot scale because they rely on client-server architecture in order to get a global and unique order of execution.

The second approach deals with resolution of $TP2$ puzzle. In this case, concurrent updates can be executed in any order, but the transformation algorithm requires to satisfy property $TP2$. This approach has been developed in adOPTed [11], SOCT2 [12], GOTO [14], and SDT [6]. Unfortunately, we have proved elsewhere [2,4] that all previously proposed transformation algorithms fail to satisfy $TP2$ when we deal with linear objects.

A first thought has been conducted in LBT environment [7] to build a total order on elements (*e.g.* characters) of the shared linear object. To establish such an order, each element must be uniquely identified. To achieve convergence, the integration of remote update consists in deducing the total order between elements. To do so, the following schemes are used: (i) a hash table containing all order relations between elements; (ii) vector timestamps for tracking causality dependency between updates; (iii) bidirectional (inclusive and exclusive) transformation algorithms; (iv) a particular class of transformation paths like the ones we have used in our integration algorithm, namely logs containing insertions before deletions. This approach is very complicated due to the number of schemes used to ensure convergence and it is also less practical even for small collaborative editors. In addition, LBT does not scale with the number of sites as the convergence is dependent on vector timestamps.

In ABT environment [8], the authors make up for the drawback of LBT. Indeed, ABT uses few schemes to ensure convergence. Besides using vector timestamps and canonical logs (insertions before deletions), ABT proceeds as follows: (i) only the effect of deletion updates are excluded from local update before to broadcast it; (ii) the integration of remote update requires the reorganization of local log to deal with the partial concurrency problem. Unlike ABT, our integration algorithm scales easily and it can provide a high degree of concurrency. Indeed, we propagate local update with minimal generation context. We also solve merely the problem of partial concurrency by integrating directly remote update without reorganizing the local log.

6. CONCLUSION

In this paper, we have proposed a new framework for managing collaborative editing work in real-time context. It is based on optimistic replication. Using OT approach, it provides a simultaneous access to shared documents and an automatic reconciliation of divergent copies. A minimal causality between updates is given by means of a dependency relation based on semantics of the shared document. This causality relation is not tributary to the number of users and it allows to support dynamic groups. Thus, our framework may be deployed in P2P networks. Note that our work is the first that deals with the scalability and the consistency maintenance problems in OT research community. A prototype based on our

OT framework has been implemented in Java[7]. It supports the collaborative editing of wiki pages and it is deployed on P2P JXTA platform[8].

In future work, we intend to make performance measurements to experimentally validate the impact of our OT framework on real-timeliness and scalability. Moreover, we plan to investigate the impact of our work when undoing updates [13].

7. REFERENCES

[1] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.

[2] A. Imine. *Conception Formelle d'Algorithmes de Réplication Optimiste. Vers l'Edition Collaborative dans les Réseaux Pair-à-Pair*. Phd thesis, University of Henri Poincaré, Nancy, France,, December 2006.

[3] A. Imine, G. Molli, P. Oster, and M. Rusinowitch. Towards Synchronizing Linear Collaborative Objects with Operational Transformation. In *IFIP FORTE'2005*, volume LNCS vol. 3731, pages 411–427, Taipei, Taiwan, October 2005. Springer.

[4] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving Correctness of Transformation Functions in Real-Time Groupware. In *ECSCW'03*, Helsinki, Finland, 14.-18. September 2003.

[5] A. Imine and M. Rusinowitch. Applying a Theorem Prover to the Verification of Optimistic Replication Algorithms. In *LNCS vol. 4600*, June 2007.

[6] D. Li and R. Li. Ensuring Content Intention Consistency in Real-Time Group Editors. In *IEEE ICDCS'04*, Tokyo, Japan, March 2004.

[7] R. Li and D. Li. A landmark-based transformation approach to concurrency control in group editors. In *ACM GROUP'05*, pages 284–293, New York, USA, 2005.

[8] R. Li and D. Li. A new operational transformation framework for real-time group editors. *IEEE Trans. Parallel Distrib. Syst.*, 18(3):307–319, 2007.

[9] B. Lushman and G. V. Cormack. Proof of correctness of ressel's adopted algorithm. *Information Processing Letters*, 86(3):303–310, 2003.

[10] A. Prakash and M. J. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4):295–330, 1994.

[11] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhauser. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *ACM CSCW'96*, pages 288–297, Boston, USA, November 1996.

[12] M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *IEEE ICDE'98*, pages 36–45, 1998.

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

[14] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *ACM CSCW'98*, pages 59–68, 1998.

[15] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving Convergence, Causality-preservation and Intention-preservation in real-time Cooperative Editing Systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

[16] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Trans. Comput.-Hum. Interact.*, 13(4):531–582, 2006.

[17] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *ACM CSCW'00*, Philadelphia, USA, December 2000.

[7] `http://webloria.loria.fr/~imine/freewiki.zip`

[8] `http://www.sun.com/software/jxta/`