

# Synchronizer Based on Operational Transformation for P2P Environments

Jean Ferrié, Michelle Cart

► **To cite this version:**

Jean Ferrié, Michelle Cart. Synchronizer Based on Operational Transformation for P2P Environments. 06039, 2006, pp.23. <lirmm-00086098>

**HAL Id: lirmm-00086098**

**<https://hal-lirmm.ccsd.cnrs.fr/lirmm-00086098>**

Submitted on 17 Jul 2006

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Synchronizer Based on Operational Transformation for P2P Environments

Michelle CART, Jean FERRIÉ

LIRMM / University Montpellier 2  
161, rue Ada - 34392 Montpellier (France)  
{cart, ferrie}@lirmm.fr

## Abstract

Reconciling divergent copies is a common problem encountered in distributed or mobile systems, asynchronous collaborative groupware, concurrent engineering, software configuration management, version control systems and personal work involving several mobile computing devices. Synchronizers provide a solution by enabling two divergent copies of the same object to be reconciled. Unfortunately, a master copy is generally required before they can be used for reconciling  $n$  copies, otherwise copy convergence will not be achieved. This paper presents the principles and algorithm of a Synchronizer which provides the means to reconcile  $n$  copies, without discriminating in favour of any particular copy. Copies can be modified (concurrently or not) on different sites and the Synchronizer we propose enables them to be reconciled pairwise, at any time, regardless of the pair, while achieving convergence of all copies. For this purpose, it uses the history of operations executed on each copy and Operational Transformations. It does not require a centralised or ordering (timestamp, state vector, etc.) mechanism. Its main advantage is thus to enable free and lazy propagation of copy updates while ensuring their convergence – it is particularly suitable for P2P environments in which no copy should be favoured.

## Keywords

Replication, copy consistency, reconciliation, history merge, synchronizer, operational transformation

## 1. Introduction

Users involved in mobile computing, concurrent engineering or distributed collaborative work, generally work on copies of shared objects. For instance, in mobile computing a user can replicate an object (calendar, file, address book, etc.) on mobile devices (Laptop, PDA, etc.) before disconnection and then manipulate these copies in disconnected work and on different devices. In asynchronous or multi-synchronous [Do95] collaborative groupware and distributed engineering, each user manages its private copy of the shared object (document, map, etc.) and propagates its updates to the others (or makes them public) when connected. In both cases, as each copy of the same object can be modified separately and independently of the others, copies could diverge and therefore have to be reconciled [SS05].

In this context, a *Synchronizer* is a tool that enables *two copies* of the same object to be reconciled in order to obtain an integrated view of the object. It takes two divergent copies of the object as an input and then returns the copies in the same state, after somehow merging the updates. For this merging, Synchronizers use a mechanism based on the state of copies or on the history and semantic properties of operations executed on these copies.

File Synchronizers (Microsoft's Briefcase, Power Merge, Windows File Synchronizer, Unisson [BP98], etc.) consider the object to be a file hierarchy. They allow create/delete actions on files and directories, and also updates, except when they concern two copies of the same file to be propagated from one copy of the file system to another. Solving the conflict is delegated to the user when updates concern two copies of the same file. Data Synchronizers (Palm Pilot Hotsync, Puma Technology Intellisync, Microsoft ActivSync, Apple I-Sync, etc.) allow reconciling and merging of updates relative to two copies of the same file (calendar,

address book, etc.), while delegating to the user conflicting situations which generally correspond to non-commutative actions. Several merge tools [MD94] presently used in the field of collaborative work and also in software engineering can be considered as Data Synchronizers – they allow two copies of the same file to be reconciled in accordance with the data type (text [Be90, Ti85], UML or XML [TKS03]), after having detected divergences using, for instance, Diff algorithms [CG97, MD94]. In another context, a reconciliation platform called IceCube [KRS01], was proposed in order to reconcile divergent copies and to merge (in a combinatory manner) their histories using semantic properties of operations executed on these copies and also using two kinds of constraints, i.e. static (e.g. when concerning non-commutative operations) and dynamic (e.g. when referring to object state). All of these merging and synchronizing tools, which were initially designed for reconciling two copies, can also be used for reconciling  $n$  copies (with  $n > 2$ ), on condition that a master copy exists and that each copy is reconciled with the master copy, otherwise copy convergence will not be achieved.

*Synchronizing multiple copies* of the same object can be obtained in a synchronous or asynchronous mode. In *synchronous methods*, all the copies play the same role and no copy is favoured. Among these, methods based on Operational Transformations [EG89], and developed for collaborative (CSCW) and real-time environments [RNG96, SYZ97, SCF97, SCF98, SE98, VCF00, SXS04, FVD04] maintain copy consistency of the same object by memorizing operations in histories and exploiting their semantic properties. In these environments, every operation generated by a user is immediately executed on his copy before broadcasting to all the other copies in order to be executed on them. As concurrent operations are not necessarily executed in the same order on each copy, they need to be ordered (by means of timestamps, state vectors or sequencer) and then transformed before being executed using, for instance, Operational Transformation, to achieve copy convergence. These methods used to synchronize  $n$  copies are limited because of their synchronous aspect and because some ordering mechanisms (timestamps, state vectors, etc.) must be available. Hence, they are not suitable for P2P environments.

With asynchronous methods, synchronizing multiple copies involves pairwise copy synchronization. To our knowledge, all published methods require a master copy, also called primary or public copy. Every copy therefore must be separately resynchronized with the master copy to achieve copy convergence. Examples of such methods are given by Configuration Management Environments [Be90, CW98, Es00], most of which are based on the *Copy/Modify/Merge* paradigm. Briefly, the master copy of the object is stored in the public space. Two (or several) users can simultaneously work on their private copy of the object, provided it has been *copied* in their private workspace. User  $U_1$  can then *modify* his copy and propagate modifications to the master copy by committing, provided that no user has committed from the time when user  $U_1$  downloaded the object copy into his workspace and the one he wants to commit. If this is not the case, user  $U_1$  needs to synchronize his copy with the last committed version and to download the latter into his workspace before *merging* his modifications with this last committed version using merge tools [MD94]. Version control systems based on these principles were specified to synchronize text files [Be90, Ti85], XML files [TKS03] or graphical objects [IN04, IN04b]. Copies are reconciled using state of their representation [Be90, Ti85, TKS03, IN04] and more recently history and semantic properties of operations [IN04b].

A Synchronizer actually has the same objective as an asynchronous method. From this standpoint, a generic data Synchronizer was proposed in [MSO03] to synchronize updates on  $n$  copies (pairwise) using the same master copy. This Synchronizer is built on an adaptation of an algorithm which was initially proposed in the context of distributed real-time collaborative (synchronous) environments [VCF00]. It exploits semantic properties and histories of operations executed on the copies and merges two histories using Operational Transformations [EG89].

The fact that a master copy is required to (pairwise) synchronize updates of  $n$  copies of the same object is incompatible with a P2P environment in which no copy should be favoured. Indeed, a P2P environment offers a completely decentralised approach for sharing objects by permitting replication of copies of the same object from site to site. Unfortunately, in existing P2P systems, there is only partial or no copy consistency, i.e. when a copy is updated the modification is not propagated to all the other copies. Although this aspect is not important for musical files, it may be for other applications. In popular P2P systems involving

millions of users, i.e. Gnutella and Kazaa, manipulated objects are read-only files, so updates are not taken into account. In Freenet [CMH02], updates are partially taken into account insofar as they are propagated from origin sites to neighbour sites when connected. In P-Grid [ACD03], update propagation is ensured by an algorithm offering probabilistic guarantees for copy consistency. However, the main drawback is that updates on copies of the same object can only come from the origin site of this object, i.e. from a master copy.

In this paper, we present a *Synchronizer* that allows copy consistency, *without discriminating in favour of any particular copy*. Even though the copies may have been modified (concurrently or not) on different sites, the Synchronizer we present, can reconcile them pairwise, regardless of the pair, and copy consistency is always ensured. The main advantage of this Synchronizer is that it allows *free and lazy propagation* of updates originating from the different copies; so, it is particularly suitable for P2P environments in which no copy is favoured. It uses the history of operations executed on each copy and is based on Operational Transformations.

The paper is set up as follows. Section 2 reviews the Synchronizer principle as well as the model, based on synchronous collaborative algorithms and on Operational Transformations, that inspired our proposal. Section 3 presents the algorithm of a Synchronizer, called MOT1, that ensures copy reconciliation while requiring a master copy. Section 4 highlights the limits of MOT1 when no master copy is available. Then we outline the principles and the algorithm of an original Synchronizer, called MOT2, which enables any two copies to be reconciled and therefore any number of copies, without favouring any particular copy, but while ensuring their convergence. Finally, some properties of histories produced by MOT2 are described and MOT2 proof is provided.

## 2. Objectives and Underlying Model

### 2.1 Synchronizer Principle

In the following, we consider an object (i.e. text, graphics, file, file system, XML tree, calendar, etc.) can be handled using definite operations  $\{op\}$  and is replicated on different sites. With each copy  $C$  of the object, the history  $H$  of operations that have been executed on  $C$  is associated. Initially, the copies are identical and they progress independently of each other on their own site, which causes them to diverge. A Synchronizer (see Figure 1) is able to reconcile two divergent copies of the same object. As an entry, it takes both copies to be reconciled as well as their associated histories, thus producing both identical copies as well as their corresponding histories which have become equivalent. A couple  $(C, H)$ , where  $C$  is the copy and  $H$  its associated history, is indicated by the name  $S$  of the site that manages the copy  $C$  – so we may talk about synchronizing either some copies or some sites. Synchronizing sites  $S_i$  and  $S_j$  is denoted  $Synch(S_i, S_j)$ .

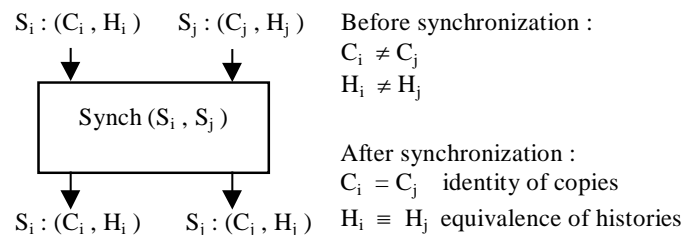


Figure 1. Synchronizer principle

Two kinds of synchronizers are considered. A Synchronizer *with a master site* assumes the existence of a particular site, called the master site, and only permits synchronization between any site and the master site. On the contrary, a Synchronizer *without a master site* permits synchronization between any two sites.

The Synchronizers we present here rely on an algorithm that can merge the histories  $H_i$  and  $H_j$  associated with copies and obtain two *identical* histories. This algorithm can be either centralised (so it runs on a single site) or distributed (so it runs on two sites, each one managing one copy and the associated history).

The following sections review the different principles underlying this algorithm: copy synchronization using synchronous collaborative algorithms, especially using SOCT4 as well as Operational Transformations.

## 2.2 Synchronous Collaborative Algorithms and Operational Transformations

Collaborative work often relies on the use of shared objects that are concurrently accessed by different users. In order to conciliate both availability and reactivity constraints when the environment is distributed, objects are generally replicated so that a copy is associated with every site or user. Therefore, the purpose of synchronous collaborative algorithms is to maintain the consistency of these copies in real-time. The real-time aspect means that an operation generated by a user is immediately executed on his copy and *integrated* as soon as possible on the other copies. Copy consistency involves three properties: (1) causality preservation among users' actions, (2) user intention preservation, and (3) copy convergence.

The difference between the various algorithms [EG89, RNG96, SYZ97, SCF97, SCF98, SE98, VCF00] concerns how they achieve *integration* of an operation. From a general standpoint, the history of operations whose execution leads to the current state is required for each object. When an operation *op* is generated at a site, it is immediately executed on the current state of the copy at this site and appended to the history, then broadcast to the other sites. The reception of an operation *op* by a site and its integration into the history at this site must take operations which are concurrent to *op* and already executed on the copy into account in order to determine the operation *op'* whose execution on the current state of the copy realises the same intention as the operation *op*. The transformation of *op* into *op'* is achieved using transposition functions, forward transposition and backward transposition [SCF97, SE98], called Operational Transformations. However, to guarantee copy convergence, Operational Transformations must meet two conditions called C1 and C2 [EG89, RNG96]. We can distinguish two kinds of collaborative algorithms:

1. algorithms [SCF98, SE98] for which the histories associated with the copies can be different (i.e. concurrent operations may be ordered in different ways according to the sites) while being *equivalent*. These algorithms use both forward transposition and backward transposition and conditions C1 and C2 must be met.
2. algorithms [SYZ97, VCF00] for which the histories associated with the copies are *identical* (i.e. concurrent operations appear in the same order on all the sites). In this class, the SOCT4 algorithm [VCF00] has a twofold advantage. It only uses forward transposition, and secondly condition C2 does not have to be met.

## 2.3 Forward Transposition

Forward transposition is used when concurrent and non-commutative operations are executed on copies in different orders. To illustrate this concept, let us consider two copies of an object,  $O_1$  and  $O_2$ , that are in the same initial state. Let us suppose that operation  $op_1$  [resp.  $op_2$ ] executed on copy  $O_1$  [resp.  $O_2$ ] leads to the state  $O_1.op_1$  [resp.  $O_2.op_2$ ]. Later, the execution of  $op_2$  [resp.  $op_1$ ] on copy  $O_1$  [resp.  $O_2$ ] leads to the state  $O_1.op_1.op_2$  [resp.  $O_2.op_2.op_1$ ]. When operations are not commutative<sup>1</sup>, the resulting states are not equivalent ( $O_1.op_1.op_2 \neq O_2.op_2.op_1$ ). In these conditions, the forward transposition allows transformation of an operation before its execution so that it takes all concurrent operations serialized before it into account. In short, the forward transposition function, denoted in the following as  $Transpose\text{-}forward(op_1, op_2)$ , is specific to a couple of concurrent operations ( $op_1, op_2$ ) which are defined from the same object state. As a result, it gives the operation  $op_2'$  (also written as  $op_2^{op_1}$ ) which has the same effect as  $op_2$  but which is defined from the state resulting from the execution of  $op_1$ , and  $op_2'$  is called the forward transposition of  $op_2$  with  $op_1$ . To guarantee copy convergence, Operational Transformations and particularly forward transpositions must meet condition C1 [EG89, RNG96] which is summed up by state equivalence:  $\forall O_i, O_i.op_1.op_2' \equiv O_i.op_2.op_1'$ . It generally must meet a further condition C2 which is not detailed.

*Example 1.* Let us assume that object  $O$  is represented as a string of characters and the operation  $insert(p, c)$  inserts character  $c$  at position  $p$  in the string. Copies  $O_1$  and  $O_2$  are in the same initial state

---

<sup>1</sup> That concerns the "forward" commutativity [We88], as opposed to the "backward" commutativity.

"effect". The operation  $op_1 = \text{insert}(2, 'f')$  executed on copy  $O_1$  adds 'f' at position 2, whereas operation  $op_2 = \text{insert}(6, 's')$ , which is concurrently executed on copy  $O_2$ , adds 's' at the end of the string. Execution of these operations in different orders leads to copies in divergent states:  $O_1.op_1.op_2 \equiv \text{"effecst"}$  and  $O_2.op_2.op_1 \equiv \text{"effects"}$ . To guarantee copy convergence, operation  $op_2$  [resp.  $op_1$ ] has to be forward transposed with  $op_1$  [resp.  $op_2$ ] before being executed on copy  $O_1$  [resp.  $O_2$ ] so as to take the concurrent operation  $op_1$  [resp.  $op_2$ ] serialized before it into account, in order to obtain:  $O_1.op_1.op_2' \equiv O_2.op_2.op_1' \equiv \text{"effects"}$ . The transposition function to be used and meeting condition C1 is as follows:

```
Transpose-forward(insert(p1, c1), insert(p2, c2)) =
  case p1 ? p2 of
    p1 < p2 : return insert(p2+1, c2) ;
    p1 > p2 : return insert(p2, c2) ;
    p1 = p2 : if c1 = c2 then return id ;
                  elsif pr(c2) > pr(c1) then return insert(p2, c2) ;
                  else return insert(p2+1, c2) ; endif ;
  endcase.
```

The forward transposition of  $op_2 = \text{insert}(6, 's')$  with  $op_1$  here would be  $op_2' = \text{Transpose-forward}(\text{insert}(2, 'f'), \text{insert}(6, 's')) = \text{insert}(7, 's')$ , whereas the forward transposition of  $op_1 = \text{insert}(2, 'f')$  with  $op_2$  would give the operation  $op_1' = op_1$  as a result.

## 2.4 Relationships between Operations and Histories

By convention, we will say that an operation is executed on a site, when it is executed on the copy located at this site.

**Definition 1.** The history  $H_i$ , associated with the copy  $C_i$  at site  $S_i$ , memorizes the sequence of operations that transforms the copy  $C_i$  from its initial state into its current state, when executed in this order on copy  $C_i$ .

More precisely, a history  $H_i$  is constituted by elements such as  $\langle \text{Id}_{op}, S_{op}, op \rangle$  where  $op$  is an operation executed on the copy associated with  $H_i$ ,  $S_{op}$  the site where the operation was generated and  $\text{Id}_{op}$  the operation identifier. Operations stored in the history  $H_i$  at site  $S_i$  have been either generated at  $S_i$  and therefore immediately executed on  $S_i$ , or were generated at another site  $S_j$  and executed on  $S_i$  as a result of synchronizing  $S_i$  with  $S_j$  or with another site.

Two operations are related by a causal ordering relation or are concurrent.

**Definition 2.** Given any two operations  $op_k$  and  $op_l$ , generated at sites  $S_{op_k}$  and  $S_{op_l}$  we say that  $op_k$  *causally precedes*  $op_l$  (noted  $op_k \rightarrow_C op_l$ ) iff: (i)  $S_{op_k} = S_{op_l}$  and  $op_k$  was generated *before*  $op_l$ , or (ii)  $S_{op_k} \neq S_{op_l}$  and  $op_l$  was generated *after* the execution of  $op_k$  by site  $S_{op_l}$ , or (iii) there is an operation  $op_m$  such that  $op_k \rightarrow_C op_m$  and  $op_m \rightarrow_C op_l$ .

Therefore, any new operation  $op$  generated at  $S_i$  and executed on the current state of the copy  $C_i$  is such that:  $\forall op_l \in H_i$ , then  $op_l \rightarrow_C op$ .

**Definition 3.** Operations  $op_k$  and  $op_l$  are said to be *independent* or *concurrent* (noted  $op_k // op_l$ ) iff: not ( $op_k \rightarrow_C op_l$ ) and not ( $op_l \rightarrow_C op_k$ ).

The relation  $\rightarrow_C$  expresses the potential causality between the operations. When  $op_k$  *causally precedes*  $op_l$ ,  $op_l$  is assumed to be dependent on the effects of  $op_k$ . In other words, the generation of  $op_l$  takes the effects produced by the execution of  $op_k$  into account. Conversely, when  $op_k$  and  $op_l$  are concurrent, then these operations are completely independent and neither one has been affected by the other.

The order of operations in  $H_i$  expresses *precedence*.

**Definition 4.** Given two operations  $op_k$  and  $op_l$  in the history  $H_i$ ,  $op_k$  is said to *precede*  $op_l$  (noted  $op_k \rightarrow_{H_i} op_l$ ) iff  $op_k$  appears *before*  $op_l$  in  $H_i$ .

Let us note that *precedence* is compatible with the causal ordering relation, i.e. if  $op_k \rightarrow_C op_l$  then  $op_k \rightarrow_{H_i} op_l$ . However, two operations may belong to the same history as a result of a synchronization, without having a causal precedence relation between them.

## 2.5 Principle of Integration in SOCT4

This section details the integration procedure of SOCT4 [VCF00], which is the basis of our synchronizer proposal. In SOCT4, operations are memorized in the histories associated with the copies according to a global unique order. To this end, a timestamp delivered by a sequencer is associated with each operation, and determines its position in every history. The integration procedure is executed by a site whenever an operation generated and broadcast by a remote site is received (reception is sequential in the timestamp order). It determines the operation to be executed on the current state of the local copy and inserts the received operation into the history at the position corresponding to its timestamp (see Figure 2).

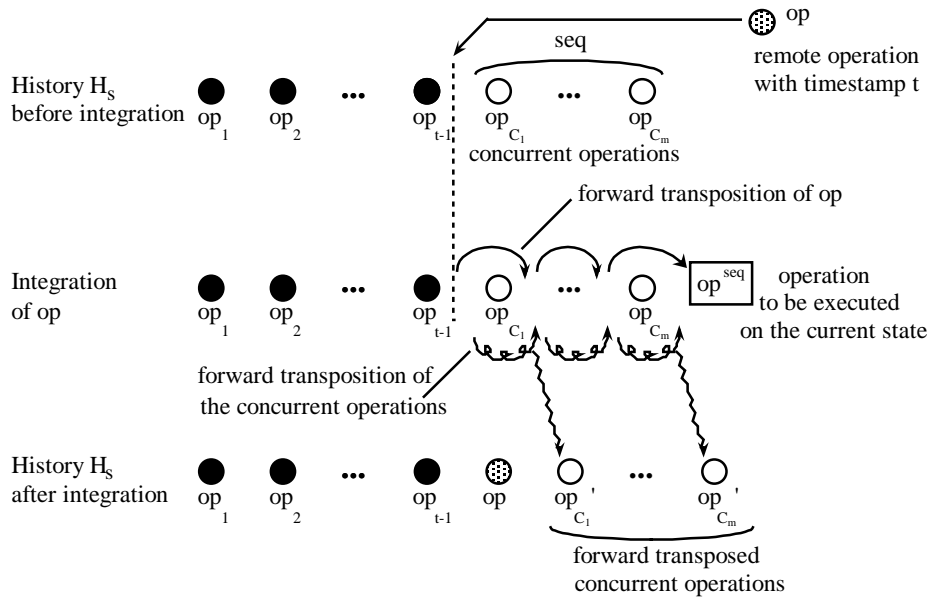


Figure 2. Integration of an operation in SOCT4

The integration of operation  $op$ , with timestamp  $t$ , in position  $t$ , is based on the assumption that  $op$  is defined on the state resulting from the execution of operations  $op_1$  to  $op_{t-1}$  and that all operations located from position  $t$  (sequence  $seq$ ) are concurrent to  $op$ .

The *Integration* procedure defined below is called to integrate the remote operation  $op$ , received as a triplet  $\langle Id_{op}, S_{op}, op \rangle$ , in position  $t$  in history  $H_S$  of site  $S$ .

*Notations:*

$C_S$  : copy of the object on site  $S$

$H_S$  : representation of the history by a table of items in the form  $\langle Id_{op}, S_{op}, op \rangle$  ;

$H_S[k].operation$  corresponds to the field  $op$  of  $H_S[k]$

$sizeofH_S$  : number of items in  $H_S$

**procedure Integration** ( $H_S, t, \langle Id_{op}, S_{op}, op \rangle$ ) ;

-- Step 1. shift the history to insert  $\langle Id_{op}, S_{op}, op \rangle$  in position  $t$

for  $k := sizeofH_S$  downto  $t$  do  $H_S[k+1] := H_S[k]$  ; end for ;

$H_S[t] := \langle Id_{op}, S_{op}, op \rangle$  ;  $sizeof H_S := sizeofH_S + 1$  ;

-- Step 2. determine the operation to be executed on the current

-- state and transform operations that follow  $op$  in  $H_S$

for  $k := t+1$  upto  $sizeofH_S$  do

$op_k := H_S[k].operation$  ;

```

HS[k].operation:= Transpose-forward(op, opk) ;
op:= Transpose-forward(opk, op) ;
end for ;
-- Step 3. execute the obtained operation on the copy of S
execute(op, CS) ;
end Integration ;

```

Step 2 is the essential integration step. Along this step, the operation  $op$  is forward transposed with each operation of  $seq$ ; the resulting operation, noted  $op^{seq}$ , is executed on the current state of the copy. During the calculation of  $op^{seq}$ , each operation of  $seq$  is transposed to take the insertion of  $op$  in  $H_S$  into account. More precisely, considering  $seq=op_{c_1}.op_{c_2}....op_{c_m}$ , each operation  $op_{c_i}$  of  $seq$  is forward transposed with  $op^{seq_{i-1}}$ , where  $seq_{i-1}= op_{c_1}.op_{c_2}....op_{c_{i-1}}$  is the sequence of operations in  $seq$  that precede  $op_{c_i}$ . So, operation  $op_{c_i}$  is replaced in  $H_S$  by  $op_{c_i}op^{seq_{i-1}}$  with  $op^{seq_0}=op$ . In the *Integration* procedure,  $Transpose-forward(op, op_k)$  is the function that delivers  $op_{c_i}op^{seq_{i-1}}$  and  $Transpose-forward(op_k, op)$  is the function that delivers  $op^{seq_i}$ , assuming that  $i=k-t$ .

The memorized history  $H_S$  does not correspond to the real history (i.e. the sequence of operations actually executed on the local copy) but it is equivalent to it. The advantage of SOCT4 is that the histories memorized on the different sites converge towards the same history and are finally identical when all operations have been integrated within all sites.

Note that the integration of an operation in the last position in  $H_S$  does not require any transposition. It only involves memorizing a new item. For clarity, we introduce the *Append* procedure, which in the following enables us to distinguish the integration with transposition from the integration without transposition.

```

procedure Append (HS, <Idop, Sop, op>) ;
  sizeofHS:= sizeofHS + 1 ;
  HS[sizeofHS]:= <Idop, Sop, op> ;
  execute(op, CS) ;
end Append ;

```

The validity conditions of the SOCT4 integration procedure may be expressed independently of timestamps as follows. Given a history  $H_S = H_C.seq$ , the integration of operation  $op$  into  $H_S$ , after  $H_C$  and before  $seq$ , is correct if conditions (a) and (b) have been met:

- (a) :  $op$  is defined from the state produced by the execution of operations in  $H_C$ ;
- (b) :  $\forall op' \in seq$ ,  $op'$  is concurrent to  $op$  (i.e. not ( $op \rightarrow_c op'$ ) and not ( $op' \rightarrow_c op$ )).

The history resulting from the integration of  $op$  is  $H_C.op.seq'$ , where  $seq'$  means that  $seq$  operations have been forward transposed with  $op$  to take the insertion of  $op$  into account.

### 3. Synchronizer with a Master Site

#### 3.1 Principle

A Synchronizer with a master site using Operational Transformation was derived from SOCT4 [MOS03]. The master site, that we designate by  $R$ , maintains the master copy as well as the history  $H_R$  of the operations applied to it from the initial state to obtain the current state. By construction, the master copy is the most up-to-date copy among all copies of the object. Any site  $S$  that wants to resynchronize its copy absolutely must merge its history  $H_S$  with the history  $H_R$  of the master copy. The following describes the merging principle.

Before the merge, the histories  $H_R$  and  $H_S$  are identical until index  $k_S$ , where  $k_S$  corresponds to the last operation of the history obtained from the most recent merge of sites  $R$  and  $S$ . So we have:  $H_S[j] = H_R[j]$ ,  $\forall j : 1 \leq j \leq k_S$ . Let us call  $H_C$  this common sub-history (common prefix) of  $H_R$  and  $H_S$  and let us designate by  $h_R$  [resp.  $h_S$ ] the part of  $H_R$  [resp.  $H_S$ ] that follows  $H_C$ . So we get:

$$H_R = H_C.h_R \quad \text{and} \quad H_S = H_C.h_S$$



More precisely,  $h_S$  corresponds to the sequence of operations generated at  $S$  since the last synchronization of  $S$  with  $R$  and  $h_R$  corresponds to the operations generated at  $R$  or at another site that was synchronized with  $R$ . The  $h_R$  operations are concurrent to the  $h_S$  operations.

Merging of histories  $H_R$  and  $H_S$  involves two phases (see Figure 3). Each one is a direct application of SOCT4.

Phase (1): it proceeds on site  $S$  and it makes the history  $H_S$  progress. It involves importing  $h_R$  and integrating it into  $H_S$  from position  $k_S + 1$ . The obtained history is:  $H_S = H_C.h_R.h_S'$  where  $h_S'$  is the sub-history  $h_S$  modified by forward-transposition to take  $h_R$  operations into account.

Phase (2): it proceeds on site  $R$  and makes the history  $H_R$  progress. It involves importing  $h_S'$  and appending (integrating without transposition) it to  $H_R$  so as to finally obtain  $H_R.h_S'$ .

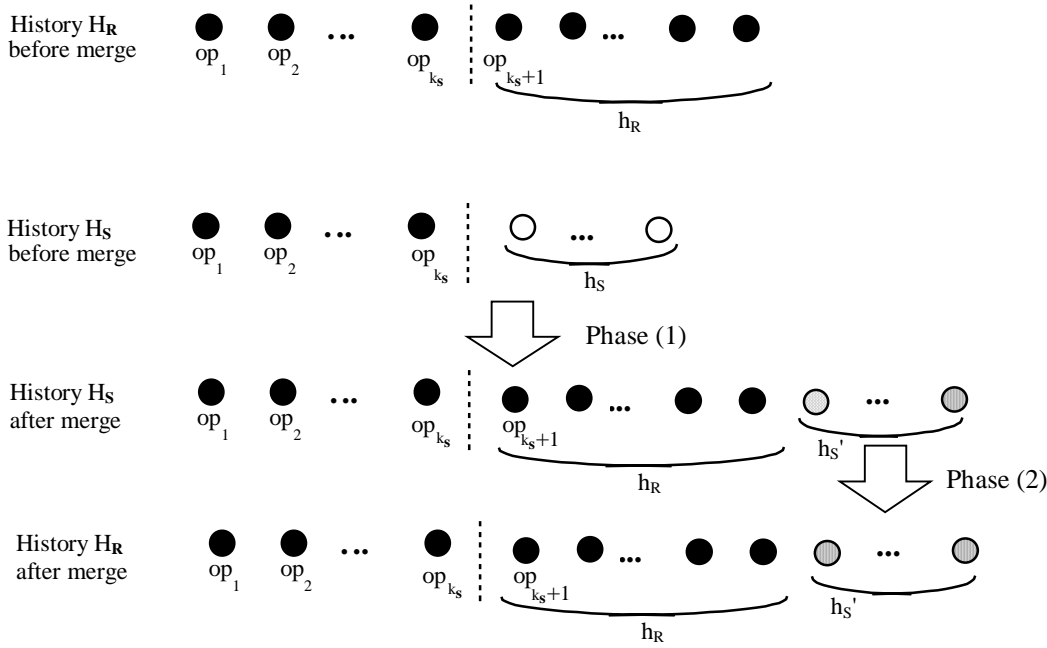


Figure 3. Merging of histories  $H_S$  and  $H_R$

In the history resulting from the synchronization of  $H_R$  and  $H_S$ , note that the sub-history  $h_R$  derived from the master site stayed unchanged whereas the sub-history  $h_S$  generated at site  $S$  was forward transposed with  $h_R$ . In fact, every operation present in  $H_R$  before the merge is already in a definitive form (shown in black in the figure), and after the merge it remains in the same range in history  $H_S$ . The  $h_S'$  operations integrated at the end of  $H_R$  are also in a definitive form after the merge.

### 3.2 The MOT1 Algorithm

The principle of the Synchronizer with a master site described above is fulfilled by the history merging algorithm MOT1 (“Merge based on Operational Transformation”). MOT1 here corresponds to a centralized algorithm, and it accepts both couples  $(C_R, H_R)$  and  $(C_S, H_S)$  as inputs and reconciles the copies  $C_R$  and  $C_S$  by merging the histories  $H_R$  and  $H_S$ . As output, it produces both couples  $(C_R, H_R)$  and  $(C_S, H_S)$ , which are identical. To illustrate the fact that MOT1 is above all an history merging algorithm, we voluntarily overlooked copies that are only accessed in the *Integration* and *Append* procedures.

procedure MOT1 ( $H_R, H_S$ ) ;

-- Look for the prefix common to  $H_R$  and  $H_S$ : determine the index  $k_S$  reached after the most recent  
 -- merging of sites  $R$  and  $S$

$k := k_S + 1$  ;

while  $k \leq \text{sizeof}H_R$  loop

-- Phase (1): integrate  $h_R$  into  $H_S$

$\langle \text{Id}_{op}, S_{op}, op \rangle := H_R[k]$  ;

```

Integration (HS, k, < Idop, Sop, op >) ;           -- HR[k] → HS[k]
k := k + 1 ;
endloop ;
while k ≤ sizeofHS loop                               -- Phase (2): append hS' to HR
    < Idop, Sop, op > := HS[k] ;
    Append (HR, < Idop, Sop, op >) ;                 -- HS[k] → HR[k]
    k := k + 1 ;
endloop ;
end MOT1 ;

```

The history resulting from phase (1) is the final history. Phase (2) only expresses the necessary evolution of H<sub>R</sub> from its initial state. So, the processing to be performed on the history of each site is emphasized, thus making the distribution of MOT1 on both sites easier.

We observe that phases (1) and (2) in MOT1 are not symmetrical. When distributing the MOT1 algorithm, phase (1) corresponds to the processing to be performed on site S, whereas phase (2) corresponds to the processing to be performed on site R. So, with MOT1, the roles of sites R and S can be considered as not symmetrical during the synchronization. Moreover, their roles are fixed *once and for all* in-so-far as the master site stays the same for all synchronizations.

The proof of MOT1 is given in Appendix A1. Beyond the fact that it is presented as a centralized algorithm, MOT1 differs from [MOS03] by the fact that no sequencer is used. A sequencer is needed in SOCT4 to timestamp and thus to globally order operations broadcast by the various sites. In [MOS03], operation timestamping enables the master site to control concurrent synchronizations. In MOT1, timestamping is unnecessary as the merge procedure is executed in a critical section. The operation order naturally corresponds to the order of their integration into the master site history H<sub>R</sub>.

Example 2 illustrates a succession of synchronizations of sites S<sub>2</sub> and S<sub>3</sub> with the master site S<sub>1</sub>, by using MOT1. The following notations are used. Synch(S<sub>i</sub>, S<sub>j</sub>) means synchronization of sites S<sub>i</sub> and S<sub>j</sub>, where S<sub>i</sub> is the master site; Synch(S<sub>i</sub>, S<sub>j</sub>) ⇒ introduces the history resulting from synchronization of S<sub>i</sub> and S<sub>j</sub>. The sequence of operations generated at site S<sub>i</sub> between its (n-1)<sup>th</sup> and n<sup>th</sup> synchronization is designated by h<sub>i</sub>[n]; h<sub>i</sub>[n]' means that operations in h<sub>i</sub>[n] have been forward transposed during the synchronization of S<sub>i</sub> with another site (merging h<sub>i</sub>[n] with another history).

*Example 2.* Initially, all copies of the object are in the same state; then sites progress independently from each other (see Figure 4). The history resulting from the first synchronization between S<sub>2</sub> and S<sub>1</sub> is: Synch(S<sub>1</sub>, S<sub>2</sub>) ⇒ h<sub>1</sub>[1].*h<sub>2</sub>[1]'* (italics are used to highlight sequences of operations that have been modified by transposition during the synchronization). Then S<sub>2</sub> and S<sub>1</sub> continue to progress independently. Further synchronization of S<sub>3</sub> and S<sub>1</sub> leads to the resulting history: Synch(S<sub>1</sub>, S<sub>3</sub>) ⇒ h<sub>1</sub>[1].h<sub>2</sub>[1]'.h<sub>1</sub>[2].*h<sub>3</sub>[1]'*, where h<sub>3</sub>[1] has been forward transposed with the master site history. Just before resynchronizing S<sub>2</sub> and the master site S<sub>1</sub>, the respective histories of S<sub>2</sub> and S<sub>1</sub> are :

$$H_2 = h_1[1].h_2[1]'.h_2[2] \quad \text{and} \quad H_1 = h_1[1].h_2[1]'.h_1[2].h_3[1]'.h_1[3].$$

Synchronizing S<sub>1</sub> and S<sub>2</sub> leads to the result: Synch(S<sub>1</sub>, S<sub>2</sub>) ⇒ h<sub>1</sub>[1].h<sub>2</sub>[1]'.h<sub>1</sub>[2].h<sub>3</sub>[1]'.h<sub>1</sub>[3].*h<sub>2</sub>[2]'*.

All copies converge towards the *same* state and will be identical when, in the absence of newly generated operations, all sites get resynchronized with the master site S<sub>1</sub> (after S<sub>3</sub> has been resynchronized with S<sub>1</sub>, in the absence of new operations generated by S<sub>2</sub>, S<sub>3</sub> and S<sub>1</sub>).

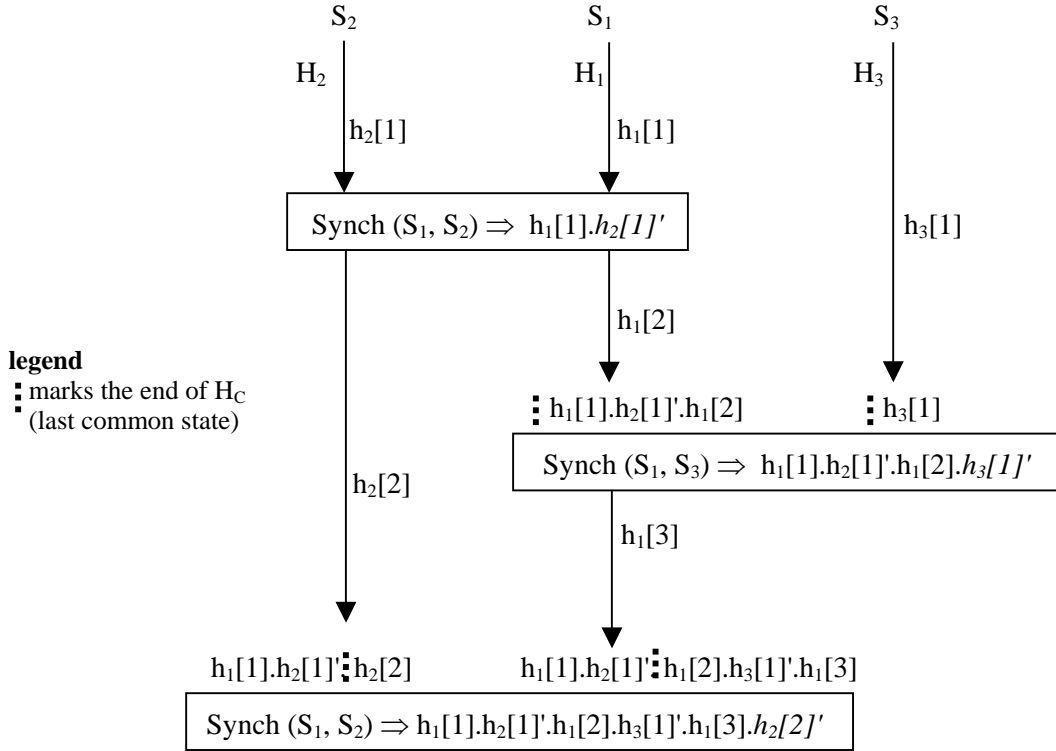


Figure 4. An example of synchronization by MOT1 with the master site  $S_1$

In summary, the following features can be retained for MOT1.

Input histories:  $H_R = H_C.h_R$   
 and  $H_S = H_C.h_S$

where  $h_S$  is a sequence of operations generated at site  $S$  and

$h_R$  is a sequence of operations generated at other sites than  $S$ .

Resulting history:  $\text{Synch}(R, S) \Rightarrow H_C.h_R.h_S'$

where  $h_S'$  means that operations of sub-history  $h_S$  have been forward transposed with respect to  $h_R$  operations.

## 4. Synchronizer without a Master Site

### 4.1 Limitations of MOT1

The principle of a Synchronizer without a master site assumes that sites get synchronized and merge their histories, in pairs, *regardless of the pair associations*, while finally ensuring *copy convergence*. To be able to use the MOT1 algorithm to achieve this, it is necessary to define the role of each  $S_i$  and  $S_j$  site, before each synchronization  $\text{Synch}(S_i, S_j)$ , since this role is not symmetrical. The role assigned to a site is temporary and only set for the duration of the synchronization. We will provide examples to illustrate the effect on the form of the resulting history and demonstrate the impossibility of obtaining copy convergence.

As a convention in the following, the notation  $\text{Synch}(S_i, S_j)$  means that when synchronizing sites  $S_i$  and  $S_j$ , the first  $S_i$  site serves as the master site ( $R$ ).

#### 4.1.1 Arbitrary role assignment to sites

In this section, we assume that each time two sites get synchronized, the site that serves as the master site is chosen arbitrarily.

*Example 3.* Let us consider sites  $S_1$ ,  $S_2$  and  $S_3$  whose copies are initially identical and which progress independently (see Figure 5). When assigning the role of master site to the  $S_1$  site, the history resulting

from synchronizing  $S_1$  and  $S_2$  is:  $\text{Synch}(S_1, S_2) \Rightarrow h_1[1].h_2[1]'$ . Note that if the role assigned to sites were different, the history resulting from their synchronization would be different:  $\text{Synch}(S_2, S_1) \Rightarrow h_2[1].h_1[1]'$ . Further synchronization of sites  $S_3$  and  $S_1$  results in the following history:  $\text{Synch}(S_3, S_1) \Rightarrow h_3[1].h_1[1]'.h_2[1]'.h_1[2]'$ . As in Example 2, the first two synchronizations are made with  $S_1$ . However  $S_1$  plays the role of master site in the first synchronization but not in the second. The histories produced are then different. As no master site is required, it is now possible to directly synchronize  $S_2$  and  $S_3$ . Before synchronization we have:

$$H_2 = \mathbf{h_1[1].h_2[1]}.h_2[2] \quad \text{and} \quad H_3 = h_3[1].\mathbf{h_1[1]'.h_2[1]'}.h_1[2]'$$

Note that  $H_2$  and  $H_3$  have no common prefix. Their last common state is the initial state. However, some operations are present in both histories although they appear under *different forms* because they have been transposed; the sequences  $\mathbf{h_1[1].h_2[1]'}$  in  $H_2$  and  $\mathbf{h_1[1]'.h_2[1]'}$  in  $H_3$  (shown in bold) exemplify this; they involve operations with the same identity (field  $\langle \text{Id}_{op} \rangle$  is identical), but under different forms (field  $\langle op \rangle$  is different). In these conditions, it is no longer possible to synchronize  $S_3$  and  $S_2$  (by either  $\text{Synch}(S_3, S_2)$  or  $\text{Synch}(S_2, S_3)$ ) when using MOT1 since the input histories do not fulfil the required property (common prefix  $H_C$ ). Applying the MOT1 merge algorithm would result in a history in which some operations (operations of  $\mathbf{h_1[1].h_2[1]'}$  and  $\mathbf{h_1[1]'.h_2[1]'}$ ) would each appear twice – which of course is incoherent.

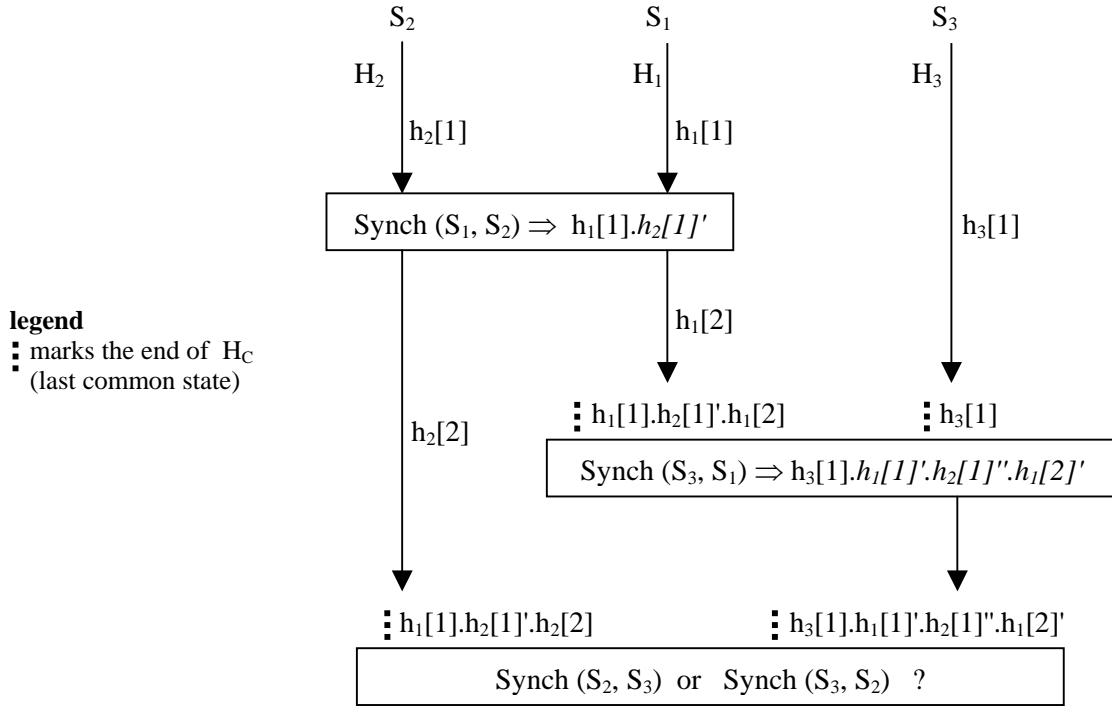


Figure 5. Synchronizing attempt using MOT1 without a master site

In the following, sequences consisting of operations that are identical in their identity ( $\langle \text{Id}_{op} \rangle$ ) but different in their form ( $\langle op \rangle$ ) and that appear in the same order are called *avatars*. In Example 3, sequences  $\mathbf{h_1[1].h_2[1]'}$  in  $H_2$  and  $\mathbf{h_1[1]'.h_2[1]'}$  in  $H_3$  are avatars. Note that avatars are characterized by the fact that they are not defined from the same state in both histories.

#### 4.1.2 Role assignment depending on a predetermined site order

To assign their role to sites that achieve synchronization, we use a predetermined total order between sites. When a synchronization  $\text{Synch}(S_i, S_j)$  between two sites  $S_i$  and  $S_j$  occurs, the  $S_i$  site serving as the master site is such that  $S_i < S_j$ . It could be thought that by totally ordering the synchronizing sites, common operations would be defined from the same state in histories to merge. The following example shows that is not the case.

*Example 4.* Let us consider the four sites  $S_1, S_2, S_3$  and  $S_4$  represented in Figure 6. The order used is:  $S_1 < S_2 < S_3 < S_4$ . With this convention, the following histories result from successive synchronizations:

$$\begin{aligned} \text{Synch}(S_1, S_3) &\Rightarrow h_1[1]; \\ \text{Synch}(S_2, S_4) &\Rightarrow h_2[1]; \\ \text{Synch}(S_2, S_3) &\Rightarrow \mathbf{h_2[1].h_2[2].h_1[1]'.h_3[2]'}; \\ \text{Synch}(S_1, S_4) &\Rightarrow \mathbf{h_1[1].h_2[1]'.h_4[2]'} \end{aligned}$$

Again it can be noted that histories  $H_1$  and  $H_3$  have no common prefix and contain avatars (shown in bold), i.e. some operations are present in both histories (field  $\langle \text{Id}_{op} \rangle$  is identical), while they appear under *different forms* (field  $\langle \text{op} \rangle$  is different) because they are not defined from the same state. Moreover, it can be noted that the avatars of  $h_1[1]$  and  $h_2[1]$  appear in *different orders* in each history. Consequently, it is not possible, for the same reasons as previously outlined, to synchronize  $S_3$  and  $S_1$  (by either  $\text{Synch}(S_3, S_1)$  or  $\text{Synch}(S_1, S_3)$ ) when using MOT1 – applying this merge algorithm would make each avatar of  $h_1[1]$  and  $h_2[1]$  appear twice in the resulting history.

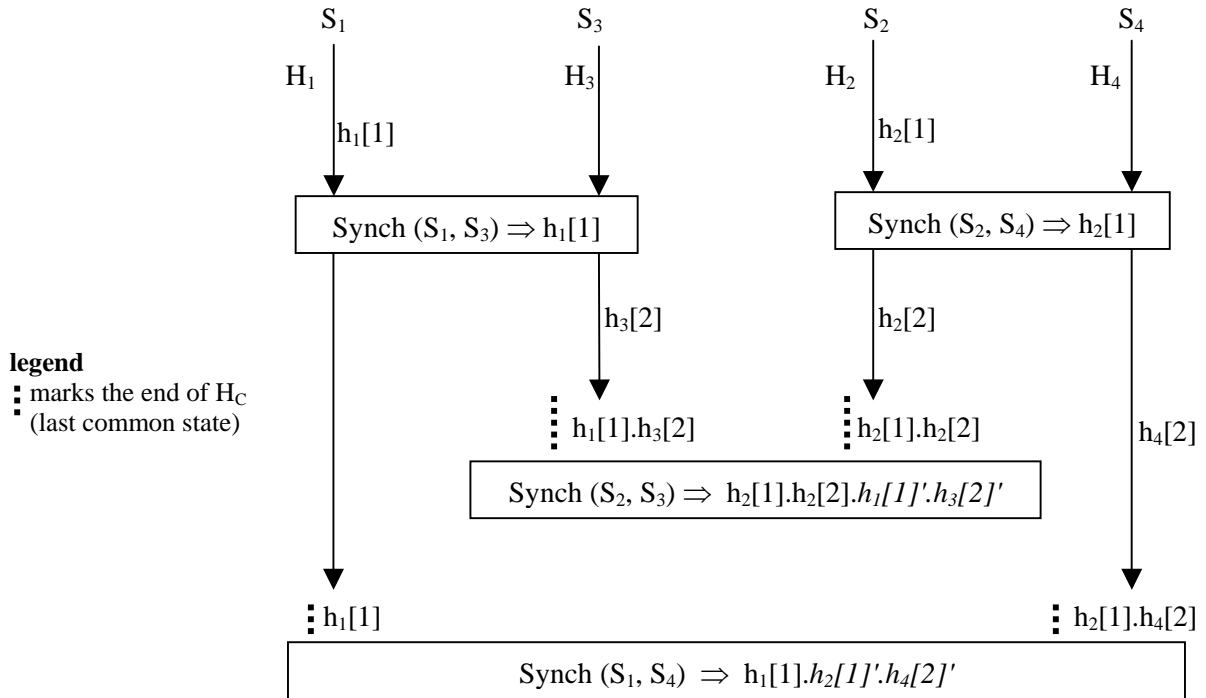


Figure 6. Using MOT1 with a predetermined order between synchronizing sites

To sum up, the use of MOT1 to synchronize a pair of any sites, without favouring any one, is useless since various problems due to characteristics of the produced histories have not finally been solved, namely: (i) the impossibility of guaranteeing the existence of a common prefix in histories that have common sub-histories, (ii) the presence of avatars, and (iii) the possibility for avatars common to several histories to appear in different orders.

## 4.2 General Principle of MOT2

In the absence of a master site, the MOT2 merge algorithm presented in the following ensures copy convergence while permitting any pair of copies to be synchronized. To obtain this property, MOT2 merges the histories by using *an order between the sites that generate* the operations instead of using an order between the sites that achieve synchronization. A unique global order can thus be built without requiring a centralizing or ordering mechanism (timestamp, state vector, sequencer, etc.). As a result, histories produced by MOT2 are such that the common sub-histories appear in the *same order*. Moreover, in MOT2, the role of

sites is totally *symmetrical* when synchronizing. In particular, the history produced by  $\text{Synch}(S_i, S_j)$  is identical to that produced by  $\text{Synch}(S_j, S_i)$  (i.e.  $\text{Synch}(S_i, S_j) \Rightarrow$  identical to  $\text{Synch}(S_j, S_i) \Rightarrow$ ).

The basic principle of MOT2 is as follows. Given the input histories:  $H_i = H_C.op_i$  and  $H_j = H_C.op_j$  where  $op_i$  and  $op_j$  are operations generated by sites  $S_{op_i}$  and  $S_{op_j}$ , respectively, the resulting history when synchronizing  $S_i$  and  $S_j$  is:

$\begin{aligned} &\text{if } S_{op_i} < S_{op_j} \text{ then } \text{Synch}(S_i, S_j) \Rightarrow H_C.op_i.op_j' \\ &\text{if } S_{op_i} > S_{op_j} \text{ then } \text{Synch}(S_i, S_j) \Rightarrow H_C.op_j.op_i' \end{aligned}$
--

where  $op_i'$  [resp.  $op_j'$ ] means that operation  $op_i$  [resp.  $op_j$ ] has been forward transposed with operation  $op_j$  [resp.  $op_i$ ].

More generally, given the input histories:  $H_i = H_C.op_i.seq_i$  and  $H_j = H_C.op_j.seq_j$  where  $op_i$  and  $op_j$  are operations respectively generated by sites  $S_{op_i}$  and  $S_{op_j}$ ,  $seq_i$  and  $seq_j$  are sequences of operations generated by any sites, the choice of the operation ( $op_i$  or  $op_j$ ) to integrate after  $H_C$  depends on the generator sites and involves the following effects:

$\begin{aligned} &\text{if } S_{op_i} < S_{op_j} \text{ then integration of } op_i \text{ in } H_j, \text{ which gives:} \\ &\quad H_i = H_C.op_i.seq_i \text{ (} H_i \text{ unchanged)} \text{ and } H_j = H_C.op_i.op_j'.seq_j' \\ &\text{if } S_{op_i} > S_{op_j} \text{ then integration of } op_j \text{ in } H_i, \text{ which gives:} \\ &\quad H_i = H_C.op_j.op_i'.seq_i' \text{ and } H_j = H_C.op_j.seq_j \text{ (} H_j \text{ unchanged)} \end{aligned}$
--

where  $op_i'.seq_i'$  [resp.  $op_j'.seq_j'$ ] means that operations  $op_i.seq_i$  [resp.  $op_j.seq_j$ ] have been forward transposed with the operation  $op_j$  [resp.  $op_i$ ].

The following example illustrates the application of this principle in the execution of Example 4.

*Example 5.* As in Example 4, we have:  $\text{Synch}(S_1, S_3) \Rightarrow h_1[1]$  and  $\text{Synch}(S_2, S_4) \Rightarrow h_2[1]$ . We will now detail the next two synchronizations  $\text{Synch}(S_2, S_3)$  and  $\text{Synch}(S_1, S_4)$  when using MOT2. We recall that the order of the sites is:  $S_1 < S_2 < S_3 < S_4$ .

1. Detail of  $\text{Synch}(S_2, S_3)$ . Before synchronization, the histories of sites  $S_2$  and  $S_3$  are:  $H_2 = h_2[1].h_2[2]$  and  $H_3 = h_1[1].h_3[2]$ . Note that  $H_C$  is empty. Synchronizing  $S_2$  and  $S_3$  according to MOT2 successively achieves the following.

- a. Integrate operations of  $h_1[1]$  into  $H_2$  as  $S_1$ , generator site of  $h_1[1]$ , and  $S_2$ , generator site of  $h_2[1]$ , are such that  $S_1 < S_2$ . The result is:

$$H_2 = \underline{h_1[1]}.h_2[1]'.h_2[2]' \text{ and } H_3 = \underline{h_1[1]}.h_3[2].$$

(We underline the resulting common sub-history and write the operations modified by forward transposition in italics).

- b. Integrate operations of  $h_2[1]'$  into  $H_3$  as  $S_2$ , generator site of  $h_2[1]'$ , and  $S_3$ , generator site of  $h_3[2]$ , are such that  $S_2 < S_3$ . The result is:

$$H_2 = \underline{h_1[1].h_2[1]'}.h_2[2]' \text{ and } H_3 = \underline{h_1[1].h_2[1]'}.h_3[2].$$

- c. Integrate operations of  $h_2[2]'$  into  $H_3$  as  $S_2$ , generator site of  $h_2[2]'$ , and  $S_3$ , generator site of  $h_3[2]$ , are such that  $S_2 < S_3$ . The result is:

$$H_2 = \underline{h_1[1].h_2[1]'}.h_2[2]' \text{ and } H_3 = \underline{h_1[1].h_2[1]'}.h_2[2]'.h_3[2]''.$$

- d. Append (integrate without transposition) operations of  $h_3[2]''$  into  $H_2$  as the end of  $H_2$  has been reached. The final result is:

$$H_2 = H_3 = h_1[1].h_2[1]'.h_2[2]'.h_3[2]''.$$

2. Detail of  $\text{Synch}(S_1, S_4)$ . Before synchronization, the histories of sites  $S_1$  and  $S_4$  are:  $H_1 = h_1[1]$  and  $H_4 = h_2[1].h_4[2]$ . Synchronizing according to MOT2 achieves the following.

a. Integrate operations of  $h_1[1]$  into  $H_4$  as  $S_1 < S_2$ . The result is:

$$H_1 = \underline{h_1[1]} \text{ and } H_4 = \underline{h_1[1]}.h_2[1]'.h_4[2]'.$$

b. Append (integrate without transposition) operations of  $h_2[1]'$  and  $h_4[2]'$  into  $H_1$  as the end of  $H_1$  has been reached. The final result is:

$$H_1 = H_4 = h_1[1].h_2[1]'.h_4[2]'.$$

After these synchronizations it should be pointed out that operations common to histories  $H_1, H_2, H_3$  and  $H_4$  (i.e.  $h_1[1].h_2[1]'$ ) appear in the *same* order and constitute their common prefix.

In Example 5, the application of MOT2 produced histories that have common contiguous sequences of operations corresponding to their common prefix. We will see later that this may produce histories that have common non-contiguous sequences of operations, and we will note that these common sequences appear in all histories in the *same* order.

### 4.3 The MOT2 Merge Algorithm

The principle described in the previous section is achieved by the MOT2 history merging algorithm. MOT2 accepts any two couples  $(C_i, H_i)$  and  $(C_j, H_j)$  as inputs and reconciles copies  $C_i$  and  $C_j$  by merging histories  $H_i$  and  $H_j$ . As an output, both couples  $(C_i, H_i)$  and  $(C_j, H_j)$  are identical. As for MOT1, we voluntarily overlooked copies only accessed in the *Integration* and *Append* procedures.

procedure **MOT2** ( $H_i, H_j$ ) ;

-- Look for the prefix  $H_C$  common to  $H_i$  and  $H_j$ : determine the index  $k_S$  of the last operation of  $H_C$

$k := k_S + 1$  ;

while ( $k \leq \text{sizeof}H_i$ ) and ( $k \leq \text{sizeof}H_j$ ) loop

<  $\text{Id}_{op_i}, S_{op_i}, op_i$  > :=  $H_i[k]$  ;

<  $\text{Id}_{op_j}, S_{op_j}, op_j$  > :=  $H_j[k]$  ;

case  $S_{op_i} ? S_{op_j}$  of

$S_{op_i} < S_{op_j}$  : **Integration** ( $H_j, k, < \text{Id}_{op_i}, S_{op_i}, op_i >$ ) ; -- Integrate  $op_i$  into  $H_j$

$S_{op_j} < S_{op_i}$  : **Integration** ( $H_i, k, < \text{Id}_{op_j}, S_{op_j}, op_j >$ ) ; -- Integrate  $op_j$  into  $H_i$

$S_{op_i} = S_{op_j}$  ; -- Operation is present in  $H_i$  and  $H_j$

endcase ;

$k := k + 1$  ;

endloop ;

-- The end of  $H_i$  or  $H_j$  has been reached

while  $k \leq \text{sizeof}H_j$  loop

-- End of history  $H_j$ : append the remainder of  $H_j$  to  $H_i$

<  $\text{Id}_{op_j}, S_{op_j}, op_j$  > :=  $H_j[k]$  ;

**Append** ( $H_i, < \text{Id}_{op_j}, S_{op_j}, op_j >$ ) ;

$k := k + 1$  ;

endloop ;

while  $k \leq \text{sizeof}H_i$  loop

-- End of history  $H_i$ : append the remainder of  $H_i$  to  $H_j$

<  $\text{Id}_{op_i}, S_{op_i}, op_i$  > :=  $H_i[k]$  ;

**Append** ( $H_j, < \text{Id}_{op_i}, S_{op_i}, op_i >$ ) ;

$k := k + 1$  ;

endloop ;

end MOT2 ;

MOT2 begins by determining the prefix common to  $H_i$  and  $H_j$ . Then, the generator sites of operations that follow the common prefix in  $H_i$  and  $H_j$  are compared in order to determine the operation to be integrated. After integration, the common prefix is augmented by one operation and the process is repeated until the end of one of the histories. The remaining operations of the history which is not terminated are then appended to the other history. When the compared operations  $op_i$  and  $op_j$  are such that  $S_{op_i} = S_{op_j}$ , they have the same generator site and are therefore identical (i.e.  $\text{Id}_{op_i} = \text{Id}_{op_j}$  and  $op_i = op_j$ ), which means that the operation is common to both histories  $H_i$  and  $H_j$ . Therefore, one directly skips to integrate the next operation. MOT2 can

thus be applied to two histories that are defined from the same initial state, without explicitly providing their last common state, which will be automatically determined by the algorithm.

The following example illustrates processing of the algorithm and presents a situation where histories produced by MOT2 have common non-contiguous operation sequences.

*Example 6.* Let us again consider the execution of Example 3 (see Figure 5) completed by adding the  $S_4$  site. Figure 5 slightly modified ( $h_2[1]$  is empty and  $h_3[2]$  has been added) is visible inside the dotted frame in Figure 7. The site order is:  $S_4 < S_3 < S_2 < S_1$ .

The histories produced by the MOT2 algorithm after successive synchronizations are now given below.

$$\begin{aligned} \text{Synch}(S_2, S_1) &\Rightarrow h_1[1] \\ \text{Synch}(S_3, S_1) &\Rightarrow h_3[1].h_1[1]'.h_1[2]' \\ \text{Synch}(S_2, S_3) &\Rightarrow h_3[1].h_1[1]'.h_2[2]'.h_1[2]'''.h_3[2]' \\ \text{Synch}(S_4, S_1) &\Rightarrow h_4[1].h_3[1]'.h_1[1]'''.h_1[2]'''.h_1[3]' \end{aligned}$$

Note that the impossibility of achieving  $\text{Synch}(S_2, S_3)$  using MOT1 (see Example 3) is overcome by using MOT2. Before once more synchronizing sites  $S_2$  and  $S_1$ , their histories are as follows:

$$H_1 = h_4[1].\mathbf{h_3[1]}.h_1[1]'''.h_1[2]'''.h_1[3]' \quad \text{and} \quad H_2 = \mathbf{h_3[1]}.h_1[1]'.h_2[2]'.h_1[2]'''.h_3[2]'.h_2[3]$$

Note that  $H_2$  and  $H_1$  contain several avatars (shown in bold), and these appear in the same order. Synchronizing  $S_2$  and  $S_1$  using MOT2 successively achieves the following statements.

a. Integrate  $h_4[1]$  into  $H_2$ , which gives:

$$\begin{aligned} H_1 &= \underline{h_4[1]}.h_3[1]'.h_1[1]'''.h_1[2]'''.h_1[3]' \quad (\text{unchanged}) \quad \text{and} \\ H_2 &= \underline{h_4[1]}.h_3[1]'.h_1[1]'''.h_2[2]'''.h_1[2]'''.h_3[2]'.h_2[3]'; \end{aligned}$$

b. Integrate  $h_2[2]''$  into  $H_1$ , which gives:

$$\begin{aligned} H_1 &= \underline{h_4[1]}.h_3[1]'.h_1[1]'''.h_2[2]'''.h_1[2]'''.h_1[3]'' and \\ H_2 &= \underline{h_4[1]}.h_3[1]'.h_1[1]'''.h_2[2]'''.h_1[2]'''.h_3[2]'.h_2[3]'} \quad (\text{unchanged}); \end{aligned}$$

c. Integrate  $h_3[2]'$  into  $H_1$ , which gives:

$$\begin{aligned} H_1 &= \underline{h_4[1]}.h_3[1]'.h_1[1]'''.h_2[2]'''.h_1[2]'''.h_3[2]'.h_1[3]'' and \\ H_2 &= \underline{h_4[1]}.h_3[1]'.h_1[1]'''.h_2[2]'''.h_1[2]'''.h_3[2]'.h_2[3]'} \quad (\text{unchanged}); \end{aligned}$$

d. Integrate  $h_2[3]'$  into  $H_1$ , which gives:

$$\begin{aligned} H_1 &= \underline{h_4[1]}.h_3[1]'.h_1[1]'''.h_2[2]'''.h_1[2]'''.h_3[2]'.h_2[3]'.h_1[3]'' and \\ H_2 &= \underline{h_4[1]}.h_3[1]'.h_1[1]'''.h_2[2]'''.h_1[2]'''.h_3[2]'.h_2[3]'} \quad (\text{unchanged}); \end{aligned}$$

e. Append  $h_1[3]''$  into  $H_2$ , which gives:

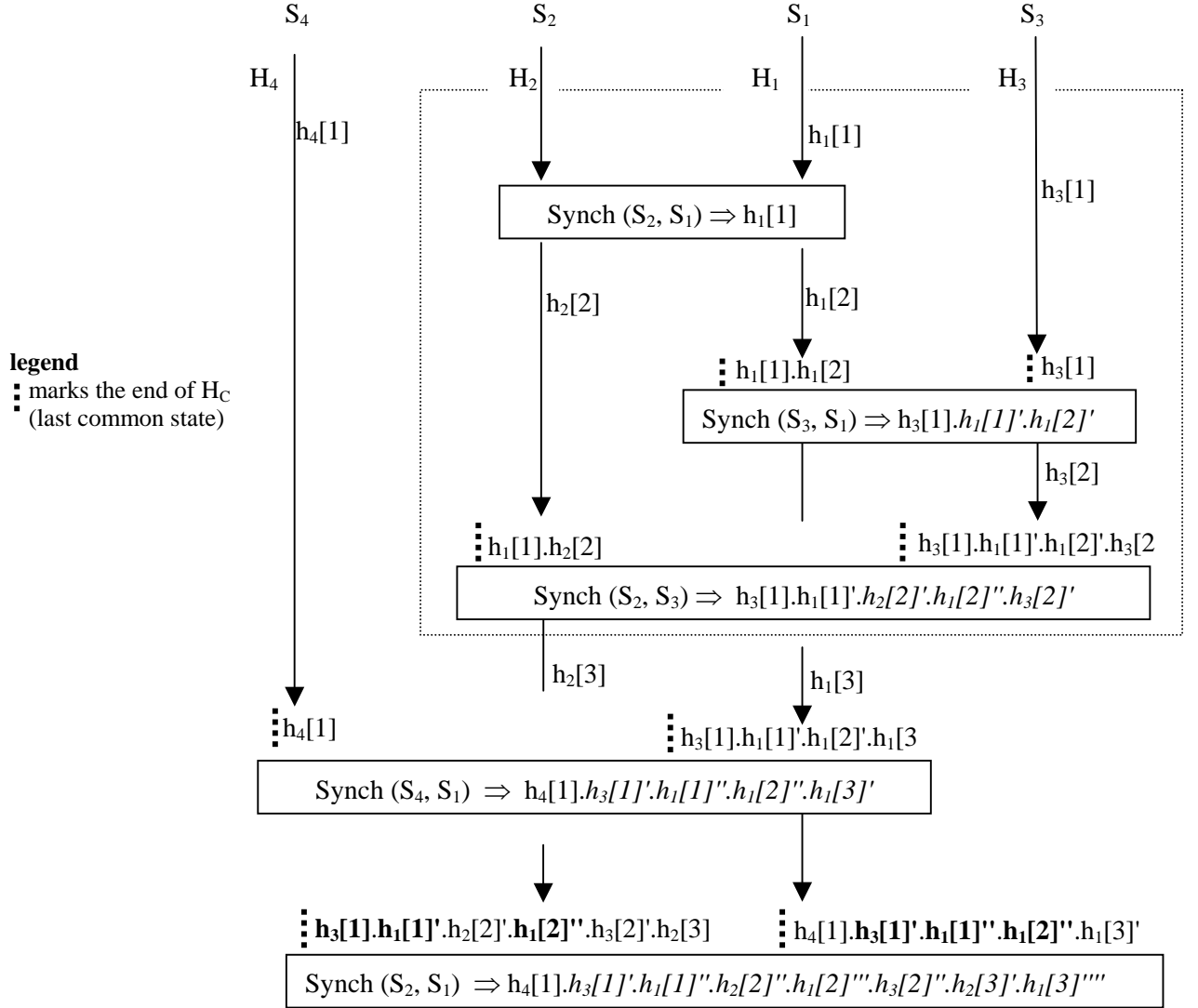
$$H_1 = H_2 = \underline{h_4[1]}.h_3[1]'.h_1[1]'''.h_2[2]'''.h_1[2]'''.h_3[2]'.h_2[3]'.h_1[3]'';$$

Finally we get:  $\text{Synch}(S_2, S_1) \Rightarrow h_4[1].h_3[1]'.h_1[1]'''.h_2[2]'''.h_1[2]'''.h_3[2]'.h_2[3]'.h_1[3]''$ .

In Example 6, we observe that the common sequences of operations, either contiguous or not, appear in the *same order* in all histories. This is actually the main feature of MOT2. While allowing free propagation of histories, since a site may at any time synchronize its copy with any site, MOT2 guarantees that sub-histories common to various histories appear in the *same order* (see Theorem 5). By means of successive synchronizations, a *global order* is built without requiring any centralized mechanism. MOT2 thus guarantees that copies will converge towards the same state. For this reason, MOT2 is particularly well suited to P2P environments where copies may be concurrently modified without discriminating in favour of any copy or site. As MOT2 uses Operational Transformations (i.e. forward transposition) and SOCT4 to merge histories, it



achieves automatic copy reconciliation while respecting causality precedence between operations when this exists.



#### 4.4 MOT2 Properties and Proof

This section presents the properties and theorems that enable us to prove the correction of the MOT2 merge algorithm. Demonstrations of the theorems are given in the Appendix.

The subsequent results we are interested in concern the operation order. Consequently, although the operations may appear under different forms because of the transpositions, we will combine the operation and its various transposed forms in the notation. In other words, the notation  $op_k$  will represent either the generated operation  $op_k$  or its forward transposed forms  $op_k'$ ,  $op_k''$ , etc. The fact that an operation got transformed through transpositions therefore no longer appears in the formulations.

**Property P1.** Given  $H_i$ ,  $\forall op_k$  and  $\forall op_l \in H_i$  such that:  $op_k \rightarrow_{H_i} op_l$ , after merging  $H_i$  with another history using MOT2, the precedence in the resulting history  $H$  is still:  $op_k \rightarrow_H op_l$ .

As (see Definition 4) the precedence is compatible with the causal ordering relation, merging preserves both the precedence ( $\rightarrow_H$ ) and the causal precedence relation ( $\rightarrow_C$ ).

Let us now consider the histories  $H_i$  and  $H_j$  of sites  $S_i$  and  $S_j$  with a common prefix  $H_C$ . Let us call  $op_i$  and  $op_j$  the operations belonging to histories  $H_i$  and  $H_j$ , respectively, and defined from the *same* state (the state left

by  $H_C$ ). Then we have:  $H_i = H_C.op_i.seq_i$  and  $H_j = H_C.op_j.seq_j$ , where  $seq_i$  and  $seq_j$  are sequences of any operations. Merging  $H_i$  and  $H_j$  using the MOT2 algorithm will produce two new histories  $H_i$  and  $H_j$  that are identical.  $H$  designates the history resulting from the merge of  $H_i$  and  $H_j$ . The precedence between  $op_i$  and  $op_j$  in  $H$  is determined by the order between their generator sites. Property P2 expresses this result.

**Property P2.** Given  $op_i$  and  $op_j$  such that:  $H_i = H_C.op_i.seq_i$  and  $H_j = H_C.op_j.seq_j$ , after merging  $H_i$  and  $H_j$  using MOT2, the precedence in the resulting history  $H$  is such that:

if  $S_{op_i} < S_{op_j}$  then  $op_i \rightarrow_H op_j$

if  $S_{op_i} > S_{op_j}$  then  $op_j \rightarrow_H op_i$ .

Property P3 generalizes property P2 to the case of a sequence of operations.

**Property P3.** Given  $op_i \in H_i$  and a sequence  $seq \in H_j$ , defined from the *same* state and such that  $H_i = H_C.op_i.seq_i$  and  $H_j = H_C.seq.seq_j$ , after merging  $H_i$  and  $H_j$  using MOT2 the precedence in the resulting history  $H$  is such that:

if  $\forall op \in seq: S_{op} < S_{op_i}$  then  $seq \rightarrow_H op_i$ .

The proof is given in Appendix A2.

Given two operations  $op_k$  and  $op_l$  that occur consecutively in a history  $H$  produced by MOT2, the following theorem specifies that if they are concurrent then they are ordered according to their generator site.

**Theorem 1.** Given two operations  $op_k$  and  $op_l$  that occur consecutively in a history  $H$  produced by MOT2 and such that  $op_k \rightarrow_H op_l$ :

if  $op_k // op_l$  then  $S_{op_k} < S_{op_l}$ .

The demonstration is given in Appendix A3. Theorem 2, obtained by contraposition, specifies that if the operations are both consecutive and ordered in the history according to the inverse order of their generator site, then they are related by a causal precedence relation.

**Theorem 2.** Given two operations  $op_k$  and  $op_l$  that occur consecutively in a history  $H$  produced by MOT2 and such that  $op_k \rightarrow_H op_l$ :

if  $S_{op_k} \geq S_{op_l}$  then  $op_k \rightarrow_C op_l$ .

These theorems are illustrated by Example 6, when each sequence  $h_i[n]$  corresponds to a single operation. The considered final history is:

Synch ( $S_2, S_1$ )  $\Rightarrow h_4[1].h_3[1].h_1[1].h_2[2].h_1[2].h_3[2].h_2[3].h_1[3]$ .

We observe that consecutive concurrent operations ( $h_4[1]$  and  $h_3[1]$ ,  $h_3[1]$  and  $h_1[1]$ ,  $h_2[2]$  and  $h_1[2]$ ,  $h_2[3]$  and  $h_1[3]$ ) are actually ordered according to their generator site order (recall that in this example the site order is:  $S_4 < S_3 < S_2 < S_1$ ). Concerning consecutive operations that are ordered according to the inverse order of their generator site ( $h_1[1]$  and  $h_2[2]$  on one hand,  $h_1[2]$  and  $h_3[2]$  on the other), they are actually related by a causal precedence relation:  $h_1[1] \rightarrow_C h_2[2]$  and  $h_1[2] \rightarrow_C h_3[2]$ .

The following theorem is a generalization of Theorem 1. Its demonstration is given in Appendix A4.

**Theorem 3.** Given the sequence  $seq.op$  in a history  $H$  produced by MOT2, where  $seq$  is constituted from operations concurrent to operation  $op$ , then:

$\forall op_k \in seq \Rightarrow S_{op_k} < S_{op}$ .

Theorem 4 is deduced by contraposition.

**Theorem 4.** Given the sequence  $seq.op$  in a history  $H$  produced by MOT2 where  $seq = op_0 \dots op_n$ . If sequence  $seq$  contains an operation  $op_k$  such that  $S_{op_k} \geq S_{op}$  then:

$\exists op_l \in \{op_k \dots op_n\} \Rightarrow op_l \rightarrow_C op$ .

These theorems can be illustrated by the same example as given previously, still considering that each sequence  $h_i[n]$  corresponds to a single operation. The final history considered is:

$$\text{Synch}(S_2, S_1) \Rightarrow h_4[1].h_3[1].h_1[1].\underline{h_2[2]}.h_1[2].h_3[2].h_2[3].h_1[3].$$

Let us consider the underlined sequence  $seq$  with  $op = h_3[2]; h_2[2]$  (i.e.  $op_k$ ) and  $h_3[2]$  (i.e.  $op$ ) are such that  $S_2 > S_3$  (i.e.  $S_{op_k} > S_{op}$ ), although  $h_2[2]$  is concurrent to  $h_3[2]$  (i.e.  $op_k // op$ ); we actually observe that  $h_1[2]$  (i.e.  $op_l$ ) exists such that:  $h_1[2] \rightarrow_C h_3[2]$  (i.e.  $op_l \rightarrow_C op$ ).

Theorem 5 states that, when histories  $H_i$  and  $H_j$  produced by MOT2 contain common operations, then these appear in the *same* order.

**Theorem 5.** Given  $op_k$  and  $op_l \in H_i$ ,  $op_k$  and  $op_l \in H_j$ , where  $H_i$  and  $H_j$  are histories produced by MOT2:

$$\text{if } op_k \rightarrow_{H_i} op_l \text{ then } op_k \rightarrow_{H_j} op_l.$$

The proof is given in Appendix A5. This theorem underlies the proof of the MOT2 algorithm. Indeed, as operations common to histories produced by MOT2 appear in the *same* order in these histories, we are sure that, whatever successive synchronizations occur, the histories which have integrated the same operations are *identical*. In other words, by means of synchronizations, histories associated with each copy integrate new operations and therefore converge towards the *same* history. So the MOT2 algorithm dynamically guarantees operation ordering according to a unique *global order* without requiring a centralising or ordering (timestamp, state vector, sequencer, etc.) mechanism.

Finally, the proof that the integration of an operation meets the validity conditions of SOCT4 (conditions (a) and (b) in section 2.5) is given in Appendix A6.

## 5. Conclusion

This paper has studied problems involving reconciliation of multiple divergent copies of the same object by means of a Synchronizer when using Operational Transformation. In this setting, we have proposed an original Synchronizer, i.e. MOT2, that enables the users to reconcile copies while ensuring their convergence and respecting the potential causal precedence between operations, without favouring any copy. MOT2 is particularly suitable for a P2P environment as it enables pairwise copy reconciliation, with any pair association, and without requiring a master site. Each site can thus synchronize its copy, *when it wants*, with *any* other site that owns a copy of the object. While permitting free propagation of update operations, MOT2 guarantees that they will appear in the *same order* in all the histories. It thus ensures, through successive synchronizations, the construction of a *global order* without requiring any ordering mechanism (timestamps, state vectors, sequencer). Finally, the Operational Transformation used (forward transposition only) necessitates verification of condition C1 only, which is easily met. The MOT2 algorithm can be centralized (it then runs on a single site) or distributed (it runs on two sites, each one owning a copy and the corresponding history). Here we have proposed a centralized version of the MOT2 algorithm in order to present its functioning principle and provide proof of its correction. It can be used as such in a distributed P2P environment by supplying each site with a version of MOT2. The algorithm is the *same* on all sites, with no site playing a specific role. Two sites only need to transmit their history to each other and then run MOT2 to achieve synchronization.

## 6. Appendix

### A1. Proof of the MOT1 Algorithm

MOT1 is verified by checking that the integration of operations in each phase meets the SOCT4 validity conditions (conditions (a) and (b) of section 2.5). In phase (1), the operation  $op_k$  (with  $op_k = H_R[k].operation$ ) to be integrated at the same position  $k$  into  $H_S$  is such that: (a) it is defined from the state left by the  $H_S[k-1].operation$  since  $\forall j: 1 \leq j \leq k-1, H_S[j] = H_R[j]$  and (b)  $\forall j: k < j \leq \text{sizeof}H_S, op = H_S[j].operation$  is concurrent to  $op_k$ , because  $op$  has been generated at site  $S$  after the last synchronization of  $S$  with  $R$ . In phase (2), the operation  $op_k$  (with  $op_k = H_S[k].operation$ ) which is added to  $H_R$  at the same position  $k$  is such that: (a) it is defined from the state left by  $H_R[k-1].operation$  since  $\forall j: 1 \leq j \leq k-1, H_R[j] = H_S[j]$  and (b)  $\nexists op$  with  $op = H_R[j].operation, \forall j: k < j \leq \text{sizeof}H_R$  since  $k = \text{sizeof}H_R$ .

### A2. Proof of Property P3

Let  $H_i = H_C.op_i.seq_i$  and  $H_j = H_C.seq.seq_j$  with  $seq = op_1.op_2.... op_h$ . As  $S_{op_1} < S_{op_i}$ , according to the P2 property, we have:  $op_1 \rightarrow_H op_i$ . As  $op_1$  is integrated into the history  $H_i$ , after the first iteration of the algorithm we get:  $H_i = H_C.op_1.op_i'.seq_i'$ , where  $op_i'.seq_i'$  is the forward transposition of  $op_i.seq_i$  with  $op_1$ . If we only consider operation ordering, then  $H_i = H_C.op_1.op_i.seq_i$ . Now, let us consider  $op_1 \in H_i$  and  $op_2 \in H_j$  which are defined on the same state (the one left by  $H_C.op_1$ ). As  $S_{op_2} < S_{op_i}$  according to P2 we get:  $op_2 \rightarrow_H op_i$ . After successive iterations, it turns out that  $op_h \rightarrow_H op_i$ . Finally,  $op_1 \rightarrow_H op_2 \rightarrow_H \dots \rightarrow_H op_h \rightarrow_H op_i$  then  $seq \rightarrow_H op_i$ .

### A3. Proof of Theorem 1

The operations  $op_k$  and  $op_i$  are concurrent, so we can deduce that they were joined together in the same history when synchronizing two sites. Let us call these sites  $S_i$  and  $S_j$ . Moreover, the operations  $op_k$  et  $op_i$  are consecutive. We can thus deduce that when histories  $H_i$  and  $H_j$  were merged, there was an integration step with:  $H_i = H_C.op_k...$  and  $H_j = H_C.op_i...$ . Since the result of this step was  $H_C.op_k.op_i...$ , then, according to the P2 property:  $S_{op_k} < S_{op_i}$ . CQFD.

### A4. Proof of Theorem 3

Operations of the sequence  $seq$  are concurrent to  $op$ , i.e. they were joined together in the same history as  $op$  as result of synchronizations. Let us consider the first synchronization when operation  $op$  and all or part of the sequence  $seq$  were joined together in the same history. Let us call this initial sequence  $seq_0$ . Let us denote  $S_i$  and  $S_j$  as the sites which got synchronized and let us assume that  $op \in H_i$  and  $seq_0 \in H_j$ . Since  $seq_0$  and  $op$  are consecutive in the resulting history, then there was, during some merging step:  $H_i = H_C.op...$  and  $H_j = H_C.seq_0...$ . Since the result was  $H_C.seq_0.op$ , then, according to the P3 property:  $\forall op_k \in seq_0, S_{op_k} < S_{op}$ .

The sequence  $seq_0$  increased with some further operations and finally became equal to  $seq$  as a result of the next synchronizations. If an operation  $op_1$  was added to  $seq_0$ , which constituted a new sequence denoted as  $seq_1$ , then, during some merge, operations  $op_1$  and  $op_m$  (with  $op_m \in \{seq_0.op\}$ ) were defined on the same state and  $op_1$  was integrated *before*  $op_m$  (i.e.  $op_1 \rightarrow_H op_m$ ). More precisely, let us suppose that  $S_i$  and  $S_j$  are the sites which got synchronized and that  $seq_0 = seq_x.op_m.seq_y$ , then:

$$H_i = H_C.seq_x.op_m.seq_y.op \dots$$

$$H_j = H_C.seq_x.op_1 \dots$$

Since the result of the merge was  $op_1 \rightarrow_H op_m$  then  $S_{op_1} < S_{op_m}$ . As  $S_{op_m} < S_{op}$ , after having integrated  $op_1$  into  $seq_0$ , we still have the property:  $\forall op_k \in seq_1, S_{op_k} < S_{op}$ . By repeating this reasoning for every operation added to  $seq_0$ , the property can finally be stated:  $\forall op_k \in seq, S_{op_k} < S_{op}$ . CQFD.

### A5. Proof of Theorem 5

Two cases must be considered according whether operations  $op_k$  and  $op_l$  that belong to histories  $H_i$  and  $H_j$  are concurrent or not.

1) op<sub>k</sub> and op<sub>l</sub> are not concurrent

The operations were ordered according to the causality relation. One of the operations, let us say op<sub>l</sub>, was generated at site S<sub>op<sub>l</sub></sub> after execution of the other one at this site. In the history of S<sub>op<sub>l</sub></sub>, according to definition 2, we can state: op<sub>k</sub> →<sub>C</sub> op<sub>l</sub>. As from the P1 property, merging preserves the precedence (and therefore the causal ordering precedence), *in all the histories* there will be: op<sub>k</sub> →<sub>C</sub> op<sub>l</sub>.

2) op<sub>k</sub> and op<sub>l</sub> are concurrent

The operations were generated independently of each other in distinct histories. As a result of synchronizing sites, operation op<sub>k</sub> which was generated at site S<sub>op<sub>k</sub></sub>, got propagated to other histories, denoted H<sup>k</sup>. In the same way, operation op<sub>l</sub> which was generated at site S<sub>op<sub>l</sub></sub> got propagated to other histories, denoted H<sup>l</sup>. Operations op<sub>k</sub> and op<sub>l</sub> are joined together in the same history when some site from H<sup>l</sup> and some site from H<sup>k</sup> get synchronized. We will demonstrate that *in any case* op<sub>k</sub> and op<sub>l</sub> will be ordered by MOT2 in the *same* way. There are again two situations according whether op<sub>k</sub> and op<sub>l</sub> were generated either from the same state or from different states of the copies.

2.1) op<sub>k</sub> and op<sub>l</sub> were generated from the same copy state

As op<sub>k</sub> and op<sub>l</sub> were generated from the *same* state, then: ∃op<sub>c</sub>: op<sub>c</sub> →<sub>C</sub> op<sub>k</sub> then op<sub>c</sub> →<sub>C</sub> op<sub>l</sub>

and conversely: ∃op<sub>c</sub>: op<sub>c</sub> →<sub>C</sub> op<sub>l</sub> then op<sub>c</sub> →<sub>C</sub> op<sub>k</sub>.

When synchronizing some site S<sub>i</sub> from H<sup>k</sup> and some site S<sub>j</sub> from H<sup>l</sup>, MOT2 execution will lead to one of the two following intermediate situations:

either (case 2.1.1): H<sub>i</sub> = H<sub>C</sub>.op<sub>k</sub>.... and H<sub>j</sub> = H<sub>C</sub>.seq.op<sub>l</sub>....

or (case 2.1.2): H<sub>i</sub> = H<sub>C</sub>.seq.op<sub>k</sub>.... and H<sub>j</sub> = H<sub>C</sub>.op<sub>l</sub>....

Since ∃op<sub>c</sub>: op<sub>c</sub> →<sub>C</sub> op<sub>k</sub> then op<sub>c</sub> →<sub>C</sub> op<sub>l</sub> and conversely: ∃op<sub>c</sub>: op<sub>c</sub> →<sub>C</sub> op<sub>l</sub> then op<sub>c</sub> →<sub>C</sub> op<sub>k</sub>, in both cases we get: ∃op<sub>c</sub>: op<sub>c</sub> →<sub>C</sub> op<sub>k</sub> or op<sub>c</sub> →<sub>C</sub> op<sub>l</sub>, op<sub>c</sub> ∈ H<sub>C</sub>. Then: ∃op ∈ seq: op // op<sub>k</sub> and op // op<sub>l</sub>. Hereafter, we have to show that, in both cases, synchronization will produce the same result. Let us suppose that S<sub>op<sub>k</sub></sub> < S<sub>op<sub>l</sub></sub> (the same reasoning holds for S<sub>op<sub>k</sub></sub> > S<sub>op<sub>l</sub></sub>).

In case 2.1.1, according to the P2 property, in the resulting history H, we necessarily get: op<sub>k</sub> →<sub>H</sub> op<sub>l</sub>.

In case 2.1.2, according to the theorem 3, we get: ∃op ∈ seq, S<sub>op</sub> < S<sub>op<sub>k</sub></sub>. Based on S<sub>op<sub>k</sub></sub> < S<sub>op<sub>l</sub></sub> and taking property P3 into account, in the resulting history H, we necessarily get: seq →<sub>H</sub> op<sub>k</sub> →<sub>H</sub> op<sub>l</sub>.

Finally, we obtain the following result.

Two concurrent operations op<sub>k</sub> and op<sub>l</sub>, which were generated from the *same* copy state, are ordered in the resulting history according to their generator site:

if S<sub>op<sub>k</sub></sub> < S<sub>op<sub>l</sub></sub> then op<sub>k</sub> →<sub>H</sub> op<sub>l</sub>

if S<sub>op<sub>l</sub></sub> < S<sub>op<sub>k</sub></sub> then op<sub>l</sub> →<sub>H</sub> op<sub>k</sub>

Therefore, they are ordered in the same way in all the histories.

2.2) op<sub>k</sub> and op<sub>l</sub> were generated from different copy states

In this case, op<sub>k</sub> and op<sub>l</sub> are said to be *partially concurrent*. This means there is an operation op<sub>m</sub> such that:

either: op<sub>m</sub> →<sub>C</sub> op<sub>l</sub> and op<sub>m</sub> and op<sub>k</sub> were generated from the same state,

or: op<sub>m</sub> →<sub>C</sub> op<sub>k</sub> and op<sub>m</sub> and op<sub>l</sub> were generated from the same state.

Let us consider the first case (similar reasoning is required for the second one). Since op<sub>m</sub> and op<sub>k</sub> are concurrent and were generated from the same state, according to the framed result obtained in 2.1, they will be ordered in the resulting history according to their generator site. Two cases thus have to be investigated depending on the respective values of S<sub>op<sub>k</sub></sub> and S<sub>op<sub>m</sub></sub>.

2.2.1 Case where S<sub>op<sub>k</sub></sub> < S<sub>op<sub>m</sub></sub>

In this case, we get:  $op_k \rightarrow_H op_m$ . By hypothesis, we also have  $op_m \rightarrow_C op_l$ . As any history containing  $op_l$  also contains all operations which causally precede  $op_l$ , we deduce that any history of  $H^l$  contains  $op_m$ . Consequently, when synchronizing any site of  $H^l$  with any site of  $H^k$ , in the resulting history we get:  $op_k \rightarrow_H op_m \rightarrow_C op_l$  and therefore  $op_k \rightarrow_H op_l$ .

Finally, by applying a similar reasoning to the second case, we obtain the following result.

Two partially concurrent operations  $op_k$  and  $op_l$  (i.e. which were generated from different states) are ordered in the resulting history in the following way:

if  $\exists op_m: op_m \rightarrow_C op_l$  and  $op_m$  and  $op_k$  were generated from the same state  
and if  $S_{op_k} < S_{op_m}$  then  $op_k \rightarrow_H op_l$ ;

if  $\exists op_m: op_m \rightarrow_C op_k$  and  $op_m$  and  $op_l$  were generated from the same state  
and if  $S_{op_l} < S_{op_m}$  then  $op_l \rightarrow_H op_k$ .

As any history containing an operation also contains all operations which causally precede it, we deduce that any history containing  $op_l$  [resp.  $op_k$ ] also contains  $op_m$ . Therefore,  $op_k$  and  $op_l$  are ordered in the same way in all histories.

### 2.2.2 Case where $S_{op_k} \geq S_{op_m}$

In this case, we get in the resulting history:  $op_m \rightarrow_H op_k$ . More precisely, execution of the MOT2 algorithm will produce the intermediate result:

firstly,  $H_i = H_C.op_k\dots$  and  $H_j = H_C.op_m\dots op_l\dots$

then,  $H_i = H_C.op_m.op_k\dots$  and  $H_j = H_C.op_m\dots op_l\dots$  where  $op_k$  has been forward transposed with  $op_m$

The  $op_k$  operation obtained after being forward transposed is defined from the state  $H_C.op_m$ , as if it had been generated from this state. Hereafter, the problem is to determine the respective positions of this new operation  $op_k$  and  $op_l$ . We are then led again to the beginning of case 2. The reasoning is iterated until obtaining a relation between  $op_k$  and  $op_l$  relevant to case 2.1 or case 2.2.1. CQFD.

The characterisation of two operations to be concurrent and generated from the same state, or partially concurrent, changes with the algorithm execution. Two operations which are initially partially concurrent, may later become concurrent by the effect of successive transpositions.

## A6. Proof of the Correct Integration of an Operation in MOT2

We have to prove that the integration of an operation meets the validity conditions of SOCT4 (conditions (a) and (b) of section 2.5).

During execution of MOT2, when two sites  $S_i$  and  $S_j$  are synchronized, the integration procedure is called to integrate an operation  $op$  of  $H_i$  into  $H_j$  under the following circumstances:  $H_i = H_C.op.seq_i$ ,  $H_j = H_C.op_k.seq_j$  and  $S_{op} < S_{op_k}$ ;  $op$  is then integrated into  $H_j$  after  $H_C$  and before  $op_k$ .

Condition (a) is verified since  $op$  is defined from the state produced by  $H_C$ .

In order to verify condition (b), we must show that:

$$\forall op_l \in op_k.seq_j, op_l // op \text{ (i.e. } \underline{\text{not}} (op \rightarrow_C op_l) \text{ and } \underline{\text{not}} (op_l \rightarrow_C op)\text{)}.$$

1. Let us suppose that:  $\exists op_l \in op_k.seq_j$  such that  $op \rightarrow_C op_l$ . It turns out that  $op$  also belongs to  $H_j$  and then we get more precisely:  $H_j = H_C.op_k.seq_{j_1}.op.seq_{j_2}.op_l.seq_{j_3}$ . We deduce that:  $\forall op_m \in op_k.seq_{j_1}$ ,  $op_m$  is concurrent to  $op$ , and therefore according to theorem 3,  $\forall op_m \in op_k.seq_{j_1}, S_{op_m} < S_{op}$ . Here there is a contradiction with the hypothesis  $S_{op} < S_{op_k}$ .

2. Let us consider an operation  $op_m$  such that:  $op_m \rightarrow_C op$ . Since any operation which causally precedes  $op$  belongs to  $H_C$ ,  $op_m$  belongs to  $H_C$  and therefore:  $\forall op_l \in op_k.seq_j, \underline{\text{not}} (op_l \rightarrow_C op)$ . CQFD.

## 7. References

- [ACD03] Aberer K., Cudré-Mauroux P., Datta A., Despotovic Z., Hauswirth M., Puceva M., Schmidt R.; *P-Grid : a self-organizing structured P2P system*; ACM SIGMOD Record, vol.32, n°3, pp.29-33, 2003.
- [Be90] Berliner B.; *Parallelizing software development*; Proc. USENIX, Washington D.C., 1990.
- [BP98] Balasubramaniam S., Pierce B.; *What is a file Synchroniser*; Proc. 4<sup>th</sup> Annual ACM / IEEE International Conference on Mobile Computing and Networking (MobiCom'98), pp.98-108, 1998.
- [CG97] Chawathe S., Garcia-Molina H.; *Meaningful change detection in structured data*; Proc. ACM International Conference on Management of Data (SIGMOD'97), pp.26-37, 1997.
- [CMH02] Clarke I., Miller S., Hong T., Sandberg O., Wiley B.; *Protecting free expression online with Freenet*; IEEE Internet Computing, vol.6, n°1, pp.40-49, 2002.
- [CW98] Conradi R., Westfechtel B.; *Version models for software configuration management*; ACM Computing Surveys (CSUR), vol. 30, n°2, pp.232-282, 1998.
- [Do95] Dourish P.; *The parting of the ways: divergence, data, management and collaborative work*; Proc. 4<sup>th</sup> European Conference on Computer Supported Cooperative Work (ECSCW'95), pp.215-230, 1995.
- [EG89] Ellis C.A., Gibbs S.J.; *Concurrency control in groupware systems*; Proc. ACM International Conference on Management of Data (SIGMOD'89), Seattle, pp.399-407, May 1989.
- [Es00] Estublier J.; *Software configuration management: a roadmap*; Proc. International Conference on Software Engineering (ICSE'00), Limerick, Ireland, pp.279-289, June 2000.
- [FVC04] Ferrié J., Vidot N., Cart M.; *Concurrent undo operations in collaborative environments using operational transformation*; Proc. 12<sup>th</sup> International Conference on Cooperative Information Systems (CoopIS'04), Cyprus, pp.155-173, October 2004.
- [IN04] Ignat C., Norrie M.C.; *Operation-based versus state-based merging in asynchronous graphical collaborative editing*; Proc. 6<sup>th</sup> International Workshop on Collaborative Editing Systems, Chicago, November 2004.
- [IN04b] Ignat C., Norrie M.C.; *Grouping in collaborative graphical editors*; Proc. ACM International Conference on Computer Supported Cooperative Work (CSCW'04), Chicago, November 2004.
- [IMO03] Imine A., Molli P., Oster G., Rusinowitch M.; *Proving correctness of transformation functions in real-time groupware*; Proc. 8<sup>th</sup> European Conference on Computer Supported Cooperative Work (ECSCW'03), Helsinki, Finland, September 2003.
- [KRS01] Kermarrec A.M., Rowstron A., Shapiro M., Druschel P.; *The IceCube approach to the reconciliation of divergent replicas*; Proc. 20<sup>th</sup> ACM Symposium on Principles of Distributed Computing (PODC), Newport R.I., August 2001.
- [MD94] Munson J.P., Dewan P.; *A flexible object merging framework*; Proc. ACM International Conference on Computer Supported Cooperative Work (CSCW'94), pp.231-242, 1994.
- [MOS03] Molli P., Oster G., Skaf-Molli H., Imine A.; *Using the transformational approach to build a safe and generic data synchronizer*; Proc. ACM International Conference on Supporting Group Work (GROUP'03), November 2003.
- [MSO02] Molli P., Skaf-Molli H., Oster G.; *Divergence awareness for virtual team through the Web*; Proc. 6<sup>th</sup> Biennial World Conference on Integrated Design & Process Technology (IDPT'02), Pasadena, June 2002.
- [RNG96] Ressel M., Nitssche-Ruhland D., Gunzenhäuser R.; *An integrating, transformation-oriented approach to concurrency control and undo in group editors*; Proc. ACM International Conference on Computer Supported Cooperative Work (CSCW'96), Boston, pp. 288-297, November 1996.
- [SCF97] Suleiman M., Cart M., Ferrié J.; *Serialization of concurrent operations in a distributed collaborative environment*; Proc. ACM International Conference on Supporting Group Work (GROUP'97), Phoenix, pp.435-445, November 1997.
- [SCF98] Suleiman M., Cart M., Ferrié J.; *Concurrent operations in a distributed and mobile collaborative environment*; Proc. 14<sup>th</sup> IEEE International Conference on Data Engineering (IEEE / ICDE'98), Orlando, pp.36-45, February 1998.
- [SE98] Sun C., Ellis C.S.; *Operational transformation in real-time group editors : issues, algorithms and achievements*; Proc. ACM International Conference on Computer Supported Cooperative Work (CSCW'98), Seattle, pp.59-68, November 1998.

- [SS05] Saito Y., Shapiro M.; *Optimistic replication*; ACM Computing Surveys, vol.37, n°1, pp.42-81, 2005.
- [SXS04] Sun D., Xia S., Sun C., Chen D.; *Operational transformation for collaborative word processing*; Proc. ACM International Conference on Computer Supported Cooperative Work (CSCW'04), Chicago, November 2004.
- [SYZ97] Sun C., Jia X., Yang Y., Zhang Y.; *A generic operation transformation schema for consistency maintenance in real-time cooperative editing systems*; Proc. ACM International Conference on Supporting Group Work (GROUP'97), Phoenix, pp.425-434, November 1997.
- [Ti85] Tichy W.F.; *RCS - A system for version control*; Software-Practice and Experience, vol.15, n°7, pp.637-654, 1985.
- [TKS03] Torii O., Kimura T., Segó J.; *The consistency control system of XML documents*; Proc. Symposium on Applications and the Internet, January 2003.
- [VCF00] Vidot N., Cart M., Ferrié J., Suleiman M.; *Copies convergence in a distributed real-time collaborative environment*; Proc. ACM International Conference on Computer Supported Cooperative Work (CSCW'00), Philadelphia, Pennsylvania, pp.171-180, December 2000.
- [We88] Weihl W. E.; *Commutativity-based concurrency control for abstract data types*; IEEE Transactions on Computers, vol. 37, n° 12, pp.1488-1505, December 1988.