Concurrent Undo Operations in Collaborative Environments using Operational Transformation

Jean Ferrié, Nicolas Vidot, Michelle Cart

Concurrent Undo Operations in Collaborative Environments Using Operational Transformation

Jean Ferrié[1], Nicolas Vidot[2], and Michelle Cart[1]

[1] LIRMM, Montpellier II University
161, rue Ada, 34392 Montpellier (France)
{ferrie, cart}@lirmm.fr
[2] GRIMAAG, Université des Antilles et de la Guyane
B.P. 7209, 97275 Schœlcher Cedex, Martinique (France)
http://www.martinique.univ-ag.fr

Abstract. In distributed collaborative systems, replicated objects, shared by users, are subject to concurrency constraints. All methods [4, 13, 18, 15, 16, 19, 22] proposed to serialize concurrent operations and achieve copies convergence of replicated objects are based on the use of *Operational Transformations*. In this context, giving the user the ability to undo an operation has been recognized as a difficult problem [1, 2, 3, 12, 14, 20, 21]. The few general propositions to solve the problem sometimes compromise copies convergence and/or users' intention, insofar as the Operational Transformations used are unsuitable for undo. This paper has a twofold objective. Firstly, it aims to highlight two general conditions (named C3 and C4) that need to be satisfied by any transformation adapted to undo. Secondly, it presents a general undo algorithm based on the definition of a generic undo-fitted transformation, which automatically verifies these conditions. The interest of the proposed method is that the undoing of an operation obeys to the same processing as the one used for regular operations in collaborative systems such as [15,19].

Keywords: Distributed collaborative systems, copies consistency, operational transformations, concurrent undo

1 Introduction

The purpose of a collaborative system is to facilitate team working and, in particular, to enable the manipulation of shared objects by members of a team whilst making them evolve in a coherent way. Usually, a shared object involved in a collaborative activity (shared text edition, shared CAD, electronic conferences, etc.) is subject to concurrent accesses and real-time constraints. The real-time aspect necessitates every user seeing the effects of his own actions on the object immediately, and the effects resulting from the actions of other users as soon as possible. In a distributed system when assuming non-negligible network latency, this high reactivity cannot be achieved unless each object is replicated on every site. Consequently, the problem is to conciliate both real-time constraint and consistency preservation of object copies, as they can be modified concurrently by many users.

R. Meersman, Z. Tari (Eds.): CoopIS/DOA/ODBASE 2004, LNCS 3290, pp. 155–173, 2004.

In this context, various algorithms [4, 13, 18, 15, 16, 19, 22], which exploit the semantic properties of the operations on the objects, have been proposed to serialize concurrent operations and thus ensure the convergence of all copies of an object. All these algorithms, which are based on Operational Transformations, exploit a transposition function to transform an operation before integrating it into the history associated with an object copy so as to respect user intention in case of concurrency. The same problem is found in configuration management [9]. In these contexts, giving the user the ability to undo an operation has been recognized as a difficult problem [1, 2, 3, 12, 14, 20, 21], when taking concurrency between operations into account. In [20] an undo algorithm ANYUNDO was proposed to enable a user to undo any operation (local or remote) that has been executed on the object. The action of undoing an operation is based on the generation of the inverse operation and the transformation of the latter to take concurrent operations into account. Unfortunately, some critical situations can compromise the convergence of the copies and/or the user intention. In a recent paper [21], corrections to ANYUNDO algorithm are made in order to remedy some critical situations. Such situations are avoided in adOPTed [14], at the expense of a restrictive undo policy which only allows local operations to be undone in the reverse execution order. The lack of generality of these algorithms is due to the fact that the Operational Transformations used are not well suited to undo. To obtain a correct result, the transposition function would have to satisfy two conditions (called C3 and C4) highlighted by our study. These conditions are difficult to check in practice. In this context, our approach proposes a general undo algorithm which automatically satisfies conditions C3 and C4 thanks to the definition of a generic Operational Transformation adapted to undo.

The paper is organized as follows. Section 2 describes the model used along with the use of the Operational Transformations to ensure the consistency of the copies of an object in distributed collaborative environments. Section 3 describes the problems presented by the undo of an operation and the conditions that must be met by the inverse operation to ensure that the action of undoing an operation is carried out correctly. Section 4 details the principles of the general undo algorithm. Section 5 illustrates how it works with an example. Section 6 compares it with the other known algorithms.

2 Operational Transformations

A distributed collaborative system is constituted from a set of sites interconnected by a supposed reliable network. Each object shared by the users is replicated so that a copy of the object exists on every site and it can be handled using definite operations. In order to maintain consistency between copies, every operation generated and executed on a site must be executed on all other copies as well. This requires every operation *generated* on a site to be *broadcast* to the other sites; after *reception* on a site, the operation is *executed* on the local copy of the object. Given a site, a *local* operation is an operation generated on this site whereas a *remote* operation is one that has been generated on another site. In order to guarantee users a minimum response time, operations generated on a site (i.e. local operations) are executed immediately on this site.

This section reviews the three constraints encountered when trying to achieve consistency maintenance of object copies and outlines the principles of their solutions: (1) causality preservation, (2) user intention preservation and (3) convergence. A collaborative text editor will be used as an example. Let us assume a text is an ordered collection of sentences, each one being an object represented by a string of characters. The operations defined on this object are:

insert(p, c): inserts character *c* at position *p* in the string,
delete(p): deletes character at position *p* in the string.

In the following, we suppose that users are working concurrently and are modifying the same sentence.

2.1 Causality Preservation

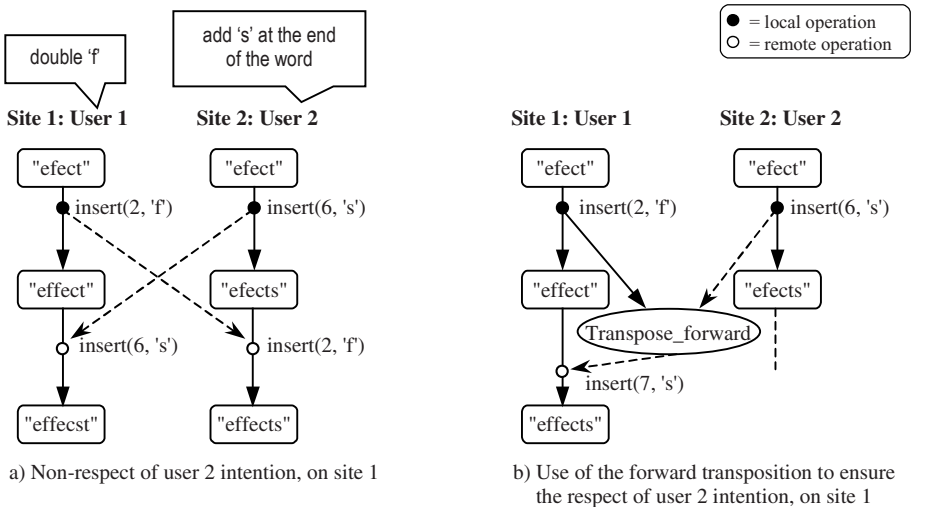
An operation op_1 is said to *causally precede* op_2 (noted op_1 $precedes_C$ op_2) iff op_2 was generated on a site after op_1 has been executed on this site. Consequently, op_2 is supposed to depend on the effects of operation op_1. *Causality preservation* ensures that all operations related by a causality relation are executed in the same order on every copy. It is achieved in the majority of the methods [4, 13, 18, 15, 19], by using a state vector associated with each site and each object and by timestamping each operation. Instead of state vectors, method [22] uses continuous timestamps delivered by a sequencer which, when associated with a differed broadcast, makes it possible to ensure a sequential reception compatible with the causal reception.

2.2 User Intention Preservation

Operations that are not causally related are said to be concurrent. In other words op_1 and op_2 are *concurrent* iff neither (op_1 $precedes_C$ op_2) nor (op_2 $precedes_C$ op_1). In this case, neither one depends on the effects of the other. Thus, they can be executed in any order on the different sites. Nevertheless, if a site executes op_1 before op_2, it must take into account the changes made by op_1 when it executes op_2 so as the *intention* of the user who generated op_2 to be respected. The intention of a user may be for instance to add 's' at the end of a word or to double a letter in a word. This intention is achieved by the execution of an operation which is relative to a specific state of the object. In the example of Figure 1-a, two users work simultaneously on the same object whose state is "efect". The intention of user 1 is to add 'f' to obtain "effect". This is achieved by operation insert(2, 'f'). The intention of user 2 is to add 's' at the end of the word which is achieved by the operation insert(6, 's'). When this operation is delivered and executed on site 1, the new state is "effecst" which is not what user 2 expected. To respect his intention, operation insert(6, 's') needs to be transformed on site 1 in order to execute insert(7, 's') instead of insert(6, 's') (see Figure 1-b).

User intention preservation ensures that the execution of an operation op on each copy has an effect that achieves the intention of the user at the time when op was generated. The problem of user intention preservation is due to the fact that an operation generated on a site achieves user intention depending on the state of the copy on this site. If this operation were to be executed on a remote site after the execution of a concurrent operation, it might no longer achieve the initial intention in

the case of the state of the copy not being the same. The solution to this problem is based on the use of Operational Transformations. This consists of transforming every remote operation to be executed so that it takes into account the modifications made by all the concurrent operations serialized before it. This transformation is possible provided that a function specific to the semantics of the operations is defined which gives for all pairs of operations (op_1, op_2) an operation written as $op_2^{op_1}$, which is defined for the state resulting from the execution of op_1 and which achieves the same intention as op_2. This transformation function introduced in [4] is also used in other systems [13, 18, 16, 19, 22] under various denominations. We call it *forward transposition*.



a) Non-respect of user 2 intention, on site 1

b) Use of the forward transposition to ensure the respect of user 2 intention, on site 1

Fig. 1. Respecting the intention of the user

Let O_i be the initial state of the object, $O_i.op$ the state obtained after the execution of op and Intention(op, O_i), the intention which is achieved by operation op on object state O_i. The forward transposition is then formally defined as follows:

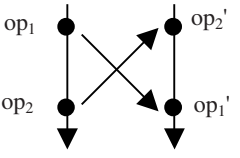
$$\text{Transpose\_forward}\ (op_1, op_2) = op_2^{op_1}$$
$$\text{with: } \forall\ O_i,\ \text{Intention}\ (op_2^{op_1}, O_i.op_1) = \text{Intention}\ (op_2, O_i).$$

Figure 1-b depicts the effect of the forward transposition for the pair of operations (insert(2, 'f'), insert(6, 's')). More generally, let seq_n be a sequence of n operations; the forward transposition of operation op with seq_n, noted op^{seq_n}, is defined recursively by: $op^{seq_n} = \text{Transpose\_forward}(op_n, op^{seq_{n-1}})$ with $seq_n = op_1.op_2....op_n = seq_{n-1}.op_n$ and $op^{seq_0} = op$, where $op_i.op_j$ represents the execution of op_i followed by the execution of op_j.

It is important to note that the forward transposition requires both operations to be defined with the object in the same state. To satisfy this requirement in all situations,

different solutions have been proposed in order to apply forward transposition in the right way.

In [18] operation op_1 is transformed using the reverse function of forward transposition (called Exclusion_Transformation), so that it is defined for the same state as op_2 and enables the use of the forward transposition. In [13] several equivalent histories which respect the causal order are kept on each site so that the intermediate states of the object can be retrieved on each site. In [11, 15, 19] a new transformation is defined. This function [11] which we call *backward transposition* makes it possible to change the execution order of a pair of operations while respecting user intention. More accurately, the backward transposition of a couple of operations (op_1, op_2), executed in this order, gives as a result the couple (op_2', op_1') corresponding to their execution in reverse order which leads to the same state, and is compatible with the forward transposition. Formally:



$op_1.op_2 \equiv op_2'.op_1'$

Transpose_backward (op_1, op_2) = (op_2', op_1')
with: op_2 = Transpose_forward (op_1, op_2') and
op_1' = Transpose_forward (op_2', op_1)

The backward transposition is only defined for a sequence of operations (op_1, op_2) obtained from concurrent operations (op_1, op_2'). Both forward and backward transpositions are examples of what is called *Operational Transformation*. In the following, these Operational Transformations are applied to objects of the type "string of characters". They are applied to XML objects in [9], and spreadsheets objects in [10]. They are also applied recursively over the different levels of a tree representation of documents in [6].

2.3 Copies Convergence

Taking into account causality as well as user intention is not always sufficient to achieve executions that guarantee the convergence of the copies on all sites. Indeed, as concurrent operations can be executed in any order on different sites, the forward transposition needs to verify two conditions [4, 13]. The first condition C1, ensures that, starting from the same state, the execution of op_1 followed by the execution of $op_2^{op_1}$ produces the same state as the execution of op_2 followed by the execution of $op_1^{op_2}$. It is formally defined as:

Condition C1. Let op_1 and op_2 be two concurrent operations defined on the same state. The forward transposition verifies C1 iff:

$$O_i.op_1.op_2^{op_1} \equiv O_i.op_2.op_1^{op_2}$$

where ≡ denotes the equivalence of states obtained after applying both sequences from the same state O_i.

Figure 2 gives an example of a forward transposition verifying condition C1. In the case of concurrent insertions of different characters at the same position (p_1=p_2), the alphabetical order (noted pr()) is arbitrarily privileged. In the case of concurrent insertions of the same character, only one character is inserted, and the returned operation is identity (id).

```
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
                 else if pr(c2) > pr(c1) then return insert (p2, c2)
                      else return insert (p2+1, c2) ;
                      endif ;
                 endif ;
  endcase
```

Fig. 2. Example of a forward transposition verifying C1

The second condition C2, ensures that the forward transposition of an operation with a sequence of two or more concurrent operations does not depend on the order used to serialize these operations. It is formally defined as follows:

Condition C2. Whatever operations op_1, op_2 and op_3 are, the forward transposition verifies C2 iff:

$$op_3^{op_1 : op_2} = op_3^{op_2 : op_1}$$

where the notation op_i:op_j denotes $op_i.op_j^{op_i}$.

Most methods, adOPTed [13], SOCT2 [15, 16] and GOTO [19] are based on satisfying conditions C1 and C2. In [18] conditions C1 and C2 are not required but a unique serialization order which complies with the causal order is imposed for the operations on all the sites; unfortunately, it may be necessary to Undo/Redo some operations to conform to this order. In [4], condition C2 is not required to the detriment of the convergence of the copies. In [22], condition C2 is not needed thanks to the implementation of a unique and continuous serialization order, given by a sequencer.

2.4 Principles of Collaborative Algorithms

Generally speaking, the principle of collaborative algorithms capable of ensuring the consistency of the copies involves memorizing the history of the operations executed from the initial state to the current state for each copy of object. Any operation generated locally is executed immediately before being added to the history. The reception of a remote operation OP requires a phase of integration to determine the operation OP' achieving the same intention as OP, to be executed on the current state. The difference between the algorithms lies in how they transform the received operation OP. For instance, in the algorithms such as SOCT2 [15, 16] or GOTO [19], when a site receives a remote operation OP, it determines the sequence seqconc of concurrent operations, then executes the processing shown in Figure 3.

1. Forward transpose OP with the sequence of concurrent operations, noted seqconc, to obtain operation OP' such that:
   Transpose_forward (seqconc, OP) = OP'
2. Execute OP' on the current state
3. Append OP' to the local history

Fig. 3. Processing of a remote operation once received by a site

In the following, we show how to undo any operation (do or undo) so that the undo operation is processed in the same way as any other operation OP, while respecting the three constraints: (1) causality preservation, (2) user intention preservation and (3) copies convergence.

3 Undo Problems

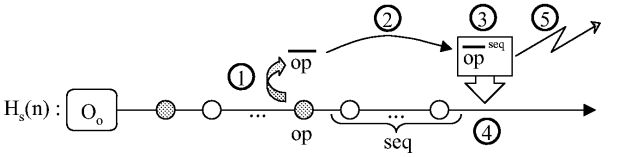
Let op be the operation to achieve intention I, executed on the object O from the initial state O_i. Let us consider the sequence seq of (n-1) operations (with seq = $op_1.op_2 \ldots op_{n-1}$), respectively, to achieve the intentions I_1, I_2,…I_{n-1} executed in this order starting from the state O_i.op.

Undoing operation op consists of generating and executing, at the current state O_i.op.seq, the operation op_{n+1} that cancels the effects of op without modifying the intentions of the other operations. This operation must lead the object to the same state as the sequence seq' of the (n-1) operations (with seq' = $op_1'.op_2' \ldots op_{n-1}'$), where these operations achieve the *same* intentions I_1, I_2,...I_{n-1} and are executed in this order starting from the initial state O_i. In other words: $O_i.op.seq.op_{n+1} = O_i.seq'$.

The operation op_{n+1} can be obtained using two different strategies.

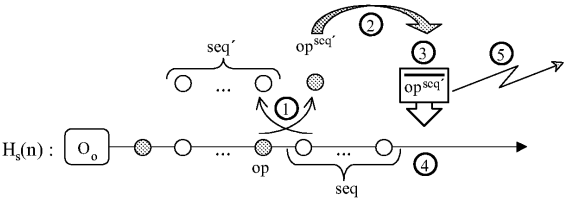
Strategy 1. It consists in generating operation $\overline{op}$, the inverse operation of op, from state O_i.op and considering it as an operation concurrent with the sequence of operations seq. Thus $\overline{op}$ must be forward transposed with seq; the operation obtained $\overline{op}^{seq}$ can then be executed on the current state. This strategy and the algorithm called naïve algorithm of undo which implements it are illustrated by Figure 4. The algorithm is executed on the site where a decision is made to undo; the operation $\overline{op}^{seq}$, which is broadcast to the other sites, is processed on these sites like a regular operation.

Strategy 2. It consists in backward transposing the pair (op, seq) so as to obtain an equivalent history (seq', op') in which the operation op' (i.e. $op^{seq'}$) to be undone is the last one executed. To undo op, it then suffices to generate operation $\overline{op}'$, the inverse operation of op', and to execute it on the current state. This strategy and the algorithm which implements it are illustrated by Figure 5.



1. Generate the *inverse of op*, noted $\overline{op}$, on the state O_i.op such that: $O_i.op.\overline{op} = O_i$
2. Calculate $\overline{op}^{seq}$ = Transpose_forward (seq, $\overline{op}$)
3. Execute $\overline{op}^{seq}$ on the current state O_i.op.seq
4. Append $\overline{op}^{seq}$ to local history
5. Broadcast $\overline{op}^{seq}$ to other sites

Fig. 4. Undoing op according to Strategy 1



1. Calculate (seq', op') = Transpose_backward (op, seq)
   with op' = $op^{seq'}$
2. Generate the *inverse of* $op^{seq'}$, i.e. $\overline{op^{seq'}}$
3. Execute $\overline{op^{seq'}}$ on the current state O_i.op.seq
4. Append $\overline{op^{seq'}}$ to the local history
5. Broadcast $\overline{op^{seq'}}$ to other sites

Fig. 5. Undoing op according to Strategy 2

Strategy 2 was first proposed and used in [12]. All other existing systems in which it is possible to undo and which are based on the Operational Transformations [14, 20, 21] use Strategy 1. They consider the undoing of an operation op as the generation of the inverse operation $\overline{op}$ Insofar as the inverse operation is regarded as a *regular* operation, this process ignores the specificity of undo and fails to observe the conditions needed to ensure the correction of the undo algorithm.

3.1 Neutrality of the Do/Undo Pair for the Transposition (Condition C3)

To ensure the preservation of user intention when undoing operation, constraints on the forward transposition must be satisfied. This is illustrated by the example in Figure 6. To undo op_1 on site 1, the naïve algorithm based on Strategy 1 leads to generation and execution of $op_2 = \sigma p \urcorner$, which is then broadcast to site 2. When operation op_3 = insert(2, 'a'), which carries out the intention to insert 'a' *after* 'b', is received on site 1 it is regarded as being concurrent with op_1 and op_2. As a result, it is forward transposed successively with op_1 and op_2 to give the operation $op_3^{op_1 \cdot op_2}$ = insert(1, 'a') whose execution leads to the state "ab". In this example, the copies

converge towards the same state "ab" and the undoing of op_1 strictly respects the intention of user 1 since 'b' was not removed. However, the intention of user 2 was not respected insofar as 'a' was inserted *before* 'b'. In fact, the transposition of op_3 with the sequence seq = $op_1.op_2$ = $op_1.\overline{op_1}$ should have resulted in op_3. In other words, the sequence $op_1.op_2$ and more generally the do/undo pair, should have acted as a neutral element for the transposition of op_3.

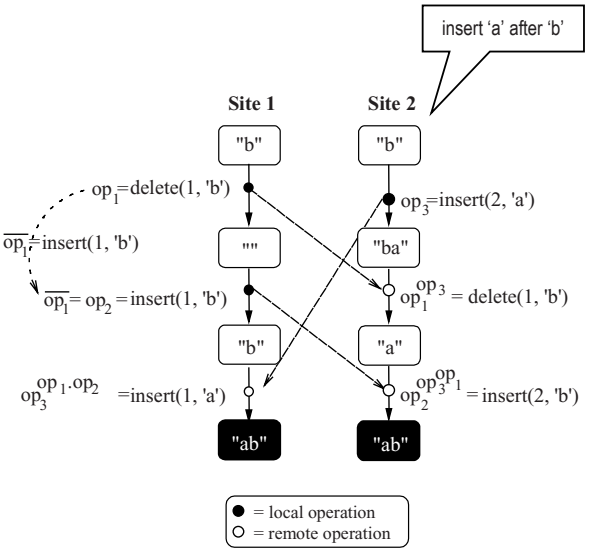


Fig. 6. Situation caused by the failure to respect condition C3

To ensure that the intention is respected, the forward transposition with an undo operation must verify the general condition C3.

Condition C3. Neutrality of do/undo pair for the transposition.

Let seq = $op_i.op_{i+1}.\ldots.op_{j-1}.op_j$ and seq' = $op_{i+1}'.\ldots.op_{j-1}'$ be two sequences such that:

- $\forall k \in [i+1..j-1]$, op_k and op_k' achieve the same intention I_k,
- op_j is the operation which undoes op_i,

then, the forward transposition verifies C3, iff :

$\forall op_k, \quad op_k seq = op_k seq'$

3.2 Forward Transposition of the Inverse of an Operation (Condition C4)

When an operation is undone by using the inverse operation, the forward transposition of this inverse operation must verify the condition C4 which is illustrated by Figure 7.

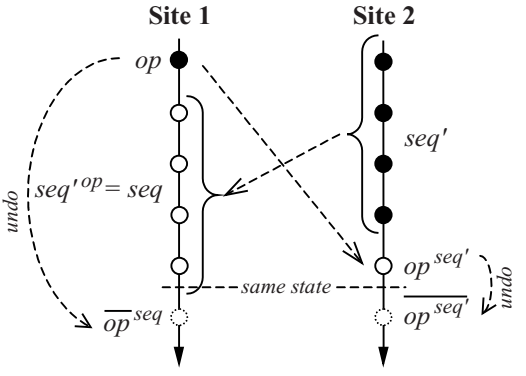


Fig. 7. Illustration of condition C4

The operation op, local to site 1, is considered as concurrent to the sequence of operations seq', which is local to site 2. It is assumed that the sequence seq' was already received on site 1 when the decision is made to undo op. The forward transposition of seq' with op is the sequence seq. On site 2, when op is received, it is forward transposed with seq' to give $op^{seq'}$. As the executions op.seq on site 1 and seq'.$op^{seq'}$ on site 2 lead to the same state, the operation which undoes op on site 1 must be identical to the one that undoes op on site 2. In other words, $\overline{op}^{seq} = \overline{op}^{seq'}$ must hold.

Condition C4. Forward transposition of the inverse of an operation.
Let op be an operation and seq and seq' two sequences such that:
- Transpose_forward (op, seq') = seq,

then the forward transposition verifies C4, iff:

$$op^{seq} = \overline{op}^{\overline{seq}'}.$$

3.3 Critical Cases Analysis

Examples of critical situations were presented in [2, 14, 20]. In these examples, undo is problematic insofar as the use of the naïve algorithm based on Strategy 1 leads to an incorrect result. In fact, as we show in [23], it appears that conditions C3 and/or C4 are not respected.

Therefore, these conditions are necessary to preserve the intentions of all operations. They are also sufficient to preserve intention since C4 ensures that the intention of undo operations is respected while C3 ensures that the intention of any operation is respected in presence of undo operations.

4 Undo Algorithm

4.1 Principle

As previously seen with the undo algorithm based on Strategy 1, the fact of using the inverse operation and forward transposing it with the operations that follow, makes it necessary to verify condition C4. In practice, it may be very difficult to verify C4 because an unspecified number of operations is involved. The method that we propose ensures that conditions C3 and C4 are automatically verified. In order to achieve this, an undo operation has to be distinguished from a *regular* operation. We thus introduce the operation *undo(op)* which expresses the intention to undo operation op. More accurately, a regular operation is specified by its name op, whereas an undo operation is specified by the name *undo( )* along with the name of the operation to be undone. Using the notations established in the definition of undo problems, we have:

$$O_i.op.seq \equiv O_i.seq'.op'$$

undo(op) *undo(op')*

with Transpose_backward (op, seq) = (seq', op'),

and thus $op' = op^{seq'}$

So, generating the operation *undo(op)*, defined on the O_i.op state, and forward transposing it with seq, must be equivalent to generating operation *undo(op')* defined for the current state O_i.seq'.op' where op' is the last operation executed and achieves the same intention as op. In our method, this is obtained thanks to the definition of forward transposition functions specific to undo, which are such that, ∀op and ∀seq :

$$\text{Transpose\_forward}(seq, \text{undo}(op)) = \text{undo}(op')$$

The execution of *undo(op')* will then consist of executing the inverse operation $\overline{op'}$ on the current state O_i.op.seq. The use of the specific operation *undo(op)* and the definition of specific transposition functions ensure that C4 is automatically verified by construction because:

as $op' = op^{seq'}$, the inverse operations actually verify

$$\overline{op'} = \overline{op^{seq'}} = \overline{op}^{seq}.$$

In practice, the operation *undo(op')* is obtained by successively forward transposing *undo(op)* with each operation in the sequence seq. The final undo algorithm executed on the site where the decision is made to cancel the operation op is shown in Figure 8-a. Before being appended to the history and broadcast, *undo(op')* is timestamped with the current state vector of the site as a regular operation [15, 16, 19].

The other sites which receive the operation *undo(op')* execute the algorithm shown in Figure 8-b. On these sites, the sequence seqconc of the operations which are concurrent to *undo(op')* is determined thanks to state vectors associated to each operation (see section 2.1).

1. Generate *undo(op)*
2. Forward transpose *undo(op)* with the operations in sequence seq to obtain operation *undo(op')* such that:
   Transpose_forward (seq, *undo(op)*) = *undo(op')*
   with op' such that: O_i.op.seq ≡ O_i.seq'.op'
3. Execute *undo(op')*, i.e. the inverse of op', i.e. $\overline{op'}$
4. Timestamp *undo(op')* by using state vector
5. Append *undo(op')* to the local history
6. Broadcast *undo(op')* to other sites

Fig. 8-a. Final Undo Algorithm on the site where the decision is made to cancel op

Forward transpose *undo(op')* with the sequence of concurrent operations, noted seqconc, to obtain operation *undo(op'')* such that:
Transpose_forward (seqconc, *undo(op')*) = *undo(op'')*
Execute *undo(op'')*, i.e. the inverse of op'', i.e $\overline{op''}$
Append *undo(op'')* to the local history

Fig. 8-b. Processing of an *undo(op')* operation once received by a site

The algorithm ensures that the do/undo pair for the transposition remains neutral. Indeed, taking the equivalence of sequences op.seq and seq'.op' into account, forward transposing an operation op_i with the sequence op.seq.undo(op') amounts to successively forward transposing op_i with the operations in sequence seq', then with op' and finally with undo(op'). Achieving this last forward transposition amounts to achieving the inverse of the forward transposition of op_i with op', as shown thereafter. The end result is that the forward transposition of op_i with the sequence op.seq.undo(op'), which contains the do/undo pair, is reduced to the forward transposition of op_i with seq', which ensures that condition C3 is verified. Let us note that the Undo Algorithm works even when the operation op itself is an *undo* operation. The proof of the Undo Algorithm can be found in [23].

An advantage of our approach lies in the fact that an *undo(op')* operation, received by a site, can be processed in the same way as a regular operation. According to Figure 3, when OP represents a regular operation op, then OP' is obtained by forward transposing op with the sequence seqconc. Let us imagine that OP represents an undo operation, *undo(op'),* then OP' would be obtained by forward transposing *undo(op')* with seqconc. That exactly matches the processing shown in Figure 8-b. Finally, the processing of a remote operation, whether it is an undo operation or a regular one, received by a site, obeys to the *same* algorithm.

In a sense, we can say that our method is based on Strategy 2, insofar as operation op' is calculated as if the operation to be undone were the last one to be executed. On another hand, it is also related to Strategy 1 insofar as operation op' is obtained by using forward transposition applied to *undo(op)* (instead of $\overline{op}$ as in Strategy 1). The advantage of our approach is that backward transposition is not needed anymore. The *only* adaptation to be done consists in determining the forward transpositions specific to *undo*.

4.2 Transpositions Specific to Undo

The method supposes that the forward transpositions written by the programmer are completed to take undo into account. In other words, the Transpose_forward (op_1, op_2) function must be specified for the cases where either op_1 or op_2 (or both) are *undo* operations. This section shows how the forward transposition can be written in generic form, taking undo into account; this generic form specific to undo does not require any work on behalf of the programmer because it only uses the operations to be undone, their forward transposition and the corresponding inverse operations which have already been defined.

Forward Transposition with an Undo Operation. This section specifies Transpose_forward (op_1, op_2) when op_1 = undo(op_3). In this case op_2 and undo(op_3) are both defined for the same state (see Figure 9-a). Thus, one can consider that the operation to be undone op_3, was executed right before undo(op_3) (see Figure 9-b).

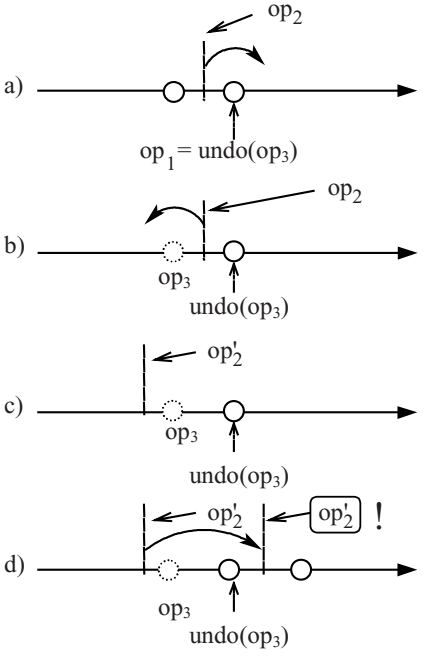


Fig. 9. Forward transposition with an undo operation

As undo(op_3) amounts to undoing the effect of op_3, forward transposing op_2 with undo(op_3) amounts to undoing the effect due to the forward transposition of the operation op_2' (to be determined) with op_3. For this we need the function which delivers op_2' such that Transpose_forward (op_3, op_2') = op_2, where op_2 and op_3 are known. This function is the inverse of the forward transposition. It is written as Transpose_forward^{-1} and formally defined by:

Transpose_forward^{-1}(op_i, Transpose_forward(op_i, op_j)) = op_j.

By applying this function to operations op_3 and op_2 (see Figure 9-b|c) operation op_2' can be obtained by: Transpose_forward$^{-1}(op_3, op_2) = op_2'$. From its definition, operation op_2' achieves the same intention as op_2 but is defined for the same state as op_3. According to the condition C3, as the forward transposition of an operation with the pair op/undo(op) must not modify this operation, the forward transposition of op_2' with the pair op_3/undo(op_3) is quite simply op_2' (see Figure 9-d). To summarize:

$$\text{Transpose\_forward}\,(\text{undo}(op_3), op_2) = \text{Transpose\_forward}^{-1}(op_3, op_2)$$

Forward Transposition of an Undo Operation. The specification of Transpose_forward (op_1, op_2) in the case where op_2 = undo(op_3) proceeds from the same method. It supposes that undo(op_3) and op_1 are defined for the same state, i.e. the state produced by operation op_3 (see Figure 10-a).

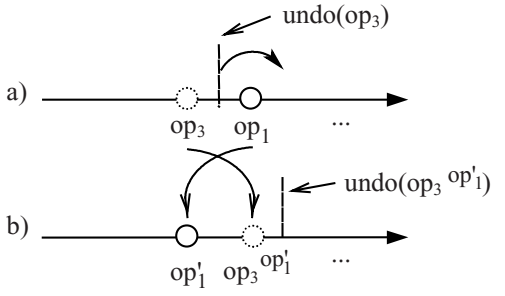


Fig. 10. Forward transposition of an undo operation

To obtain the operation that undoes the effects of op_3 after op_1 was executed, it suffices to backward transpose the pair (op_3, op_1):

$$\text{Transpose\_backward}\,(op_3, op_1) = (op_1', op_3 op_1'),$$
$$\text{with } op_1' = \text{Tranpose\_forward}^{-1}(op_3, op_1).$$

The transposed operation $op_3 op_1'$ achieves the same intention as op_3 if it had been executed just after op_1 (see Figure 10-b). Therefore, the operation we need in order to undo the effect of op_3 is undo($op_3 op_1'$). To summarize:

$$\text{Transpose\_forward}\,(op_1, \text{undo}(op_3)) = \text{undo}\,(\text{Transpose\_forward}\,(\text{Transpose\_forward}^{-1}(op_3, op_1), op_3))$$

The complete generic function of the forward transposition which takes undo into account is given in Figure 11. It uses the inverse of the forward transposition. The following specifies how to obtain it when one of the operations in the couple is also an undo operation.

```
Transpose_forward (op1,op2) =
 case (op1 = undo(op3))
      if (op2 = undo(op4)) and op3 = op4
           return id(op2)
      else
           return Transpose_forward-1(op3,op2)
      endif ;
 case (op2 = undo(op4))
      return undo (Transpose_forward (Transpose_forward -1(op4,op1), op4)) ;
 case {op1 and op2 are regular operations}
 endcase
```

Fig. 11. Forward transposition adapted to undo

Inverse of the Forward Transposition with an Undo Operation. When op_1 is an undo operation, the inverse of the forward transposition, i.e. Transpose_forward$^{-1}(op_1, op_2)$ can be obtained by the same logic [23]. It is written:

$$\text{Transpose\_forward}^{-1}(\text{undo (op)}, op_2) = op_2 op$$

Inverse of the Forward Transposition of an Undo Operation. The inverse of the forward transposition, Transpose_forward$^{-1}(op_1, op_2)$, when op_2 is an undo operation is difficult to obtain. Given that we know op_1 and op_2, this amounts to finding op_2' such that Transpose_forward $(op_1, op_2') = op_2$. As op_2 is an undo operation, written as undo(op_3), then op_2' is also an undo operation, written as undo(op_3'). Thus, finding op_2' amounts to finding op_3'. When considering the relation previously established to calculate the forward transposition of an undo operation, operation op_2 is given by the following:

op_2 = undo (Transpose_forward (Transpose_forward$^{-1}(op_3', op_1), op_3')$).

In addition, when considering op_2 = undo(op_3), given that we know op_2, so we know op_3. As op_3 is given by op_3 = Transpose_forward (Transpose_forward$^{-1}(op3', op_1), op_3')$, finding op_3' would be necessary before knowing Transpose_forward$^{-1}(op_3', op_1)$. This evaluation is impossible using operation op_1 alone. In fact, in order to obtain the result, we need to refer to the history of the site [23] and to reorder the operations to obtain an equivalent history containing the operation op_2', i.e. undo(op_3').

5 Illustration of the Undo Algorithm

In this section we show on an example how our Undo Algorithm works. The example, referred as *Insert-Insert-Tie Dilemma* [20], corresponds to a situation illustrated in Figure 12-a. User 1 on site 1 deletes the character 'b', while user 2 on site 2 inserts 'a'

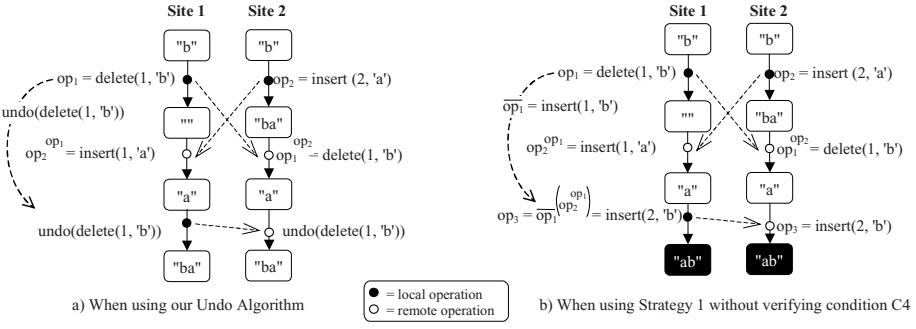


Fig. 12. Insert-Insert-Tie example

after 'b'. When op_2 (resp. op_1) is received on site 1 (resp. site 2), it is forward transposed with op_1 (resp. op_2) before being executed. Respecting condition C1 ensures that the copies converge towards the same state "a". Let us suppose that user 1 then decides to cancel operation op_1. The application of our Undo Algorithm (see Figure 8-a), with operation *op* corresponding to *delete(1, 'b')*, gives the following statements:

Step 1. Generate *undo(delete(1, 'b'))*.

Step 2. Seeing that seq = insert(1, 'a'), thus compute:

Transpose_forward (insert(1, 'a'), *undo(delete(1, 'b'))*).

Referring to Figure 11, insert(1, 'a') corresponds to op_1 and *undo(delete(1,'b'))* to op_2, which leads to the case where $op_2 = \text{undo}(op_4)$, with op_4 = delete(1, 'b'). According to these notations, the result to be computed is:

undo (Transpose_forward(Transpose_forward $^{-1}$*(*op_4*,*op_1*), op4))*

which needs to compute:

i) the inverse of Transpose_forward(delete(1, 'b'), insert(1, 'a')); the result is insert(2, 'a');

ii) Transpose_forward (insert(2, 'a'), delete(1, 'b')); the result is delete(1, 'b').

The final result of step 2 is *undo(delete(1, 'b'))*.

Step 3. Execute the inverse of delete(1, 'b'), that is insert(1, 'b'), which leads to the final state "ba".

After being timestamped and appended to the history, the resulting operation of step 2, *undo(delete(1, 'b'))*, is broadcast to site 2. When it is received on this site, the algorithm shown in Figure 8-b is applied. As there is no concurrent operation, the inverse of delete(1, 'b'), that is insert(1, 'b'), is directly executed and leads to the same state "ba".

The application of the naïve algorithm based on Strategy 1 (see Figure 4) and illustrated in Figure 12-b would lead to a wrong result. The operation $\overline{op_1}$ = insert(1, 'b') would be generated then forward transposed with $op_2^{op_1}$ to obtain $op_3 = \overline{op_1}^{op_2^{op_1}}$ = insert(2, 'b') whose execution would lead to state "ab". Operation op_3, broadcast to site 2 would be executed as it is on this site. Although the copies would converge towards the same state, the undoing of op_1 would lead to an incorrect state since the intention of user 2, namely 'a' placed *after* 'b', would not be respected in the

final state. The reason for this anomaly is due to the failure to respect condition C4. The operation executed on site 2 to undo operation op_1 op_2 = delete(1, 'b'), should be identical to the inverse operation op_1 $\overline{op_2}$ = insert(1, 'b'), which is not the case here.

It results that, when using our Undo Algorithm, the example referred as *Insert-Insert-Tie Dilemma* [20], is not a critical situation anymore.

6 Comparison with Existing Approaches

The adOPTed algorithm [13] is based on the use of Operational Transformations and on a multidimensional history associated with each copy. This history is represented by a graph where each dimension relates to the operations generated by a given user. An extension to the adOPTed algorithm, based on the naïve algorithm, is proposed in [14] to enable a user to undo operations. However, the extension restricts undo to local operations only on condition that they are undone according to the inverse chronological order. As a result of these limitations an operation $\overline{op}$ that undoes op can only be separated from the operation op in a given dimension by a sequence containing do/undo pairs only. This characteristic facilitates the adaptation of the forward transposition function (called *translateRequest*) so that it can take the do/undo pairs into account and ensure that condition C3 is verified. Moreover, thanks to the multidimensional history, for any operation op concurrent to a sequence seq, the operation op^{seq} is directly available. Therefore, the verification of condition C4 is unnecessary because the undoing of op is achieved by generating and executing $op^{\overline{seq}}$. However, the undoing of local operations according to their inverse chronological order remains a restrictive solution.

The DistEdit Selective Undo algorithm [12] implements Strategy 2 and only ensures that a condition equivalent to C3 is met, since condition C4 is automatically satisfied by this Strategy.

The REDUCE system [19] is based on Operational Transformations and on a linear history associated with each copy of the object. The principle of the undo algorithm, called ANYUNDO [20, 21], is a naïve algorithm adaptation obtained by grouping an operation and the corresponding undo in the history. This adaptation makes it possible to ensure the neutrality of the do/undo pairs during the transposition of an operation and, therefore, to ensure that condition C3 is met. More precisely, undoing op is achieved by: generating $\overline{op}$, transposing it forward with the sequence seq of the operations executed after op; and executing and broadcasting the operation obtained $\overline{op}^{seq}$. Grouping op and the corresponding undo operation to obtain the do/undo pair, written as op^*, is achieved by backward transposing the pair (seq, $\overline{op}^{seq}$). In [20], the lack of a timestamp for an undo operation makes it impossible to distinguish between concurrent operations and causally dependent operations; that may result in violating user intention and lead to the divergence of the copies. In [21], this mistake is corrected and conditions IP1, IP2 and IP3 equivalent to conditions C3 and C4 are retrieved. However, the correction is obtained by extending the ANYUNDO algorithm with undo specific additive treatments. The interest of our approach lies in the generality of the processing of a remote operation whether it is an undo operation or a regular one. This generality is obtained thanks to the introduction of a specific undo operation which obeys to the same processing as a regular operation.

7 Conclusion

This article reviews the problems arising from the cancellation of an operation in distributed collaborative environments that use Operational Transformations. Traditionally, undo operations were often limited to the handling of the inverse operation. However, we show that when concurrency occurs, this approach is insufficient to ensure that the copies converge and it fails to respect user intention. Moreover, the forward transposition function must verify two conditions, which we have highlighted. However, these conditions are difficult to check in practice. In this context, we proposed a general undo algorithm for which these conditions are automatically met. Its originality lies in its capacity to consider undo as a specific operation that requires the adaptation of the Operational Transformations, an adaptation for which we give a generic specification. This algorithm makes it possible to undo any operation, local or remote, in all situations of concurrency, including those that are widely considered as problems. The paper concludes with a comparison with existing algorithms.

References

[1] Abowd G. D. , Dix A. J.; "Giving Undo Attention"*;* Interacting with Computers, vol. 4, n° 3, pp. 317-342, 1992.

[2] Berlage T.; "A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects"*;* ACM Transactions on Computer-Human Interaction, vol. 1, n° 3, pp. 269-294, 1994.

[3] Choudhary R., Dewan P.; "A general Multi-User Undo/Redo Model"; Proc. of European Conference on Computer Supported Work (ECSCW'95), Stockholm, October 1995, pp. 231-246.

[4] Ellis C.A., Gibbs S.J.; "Concurency Control in Groupware Systems"; Proc. ACM Int. Conf. on Management of Data (SIGMOD'89), Seattle, May 1989, pp.399-407.

[5] Ellis C.A., Gibbs S.J., Rein G.L.; "Groupware: Some issues and experiences"*;* Commun. ACM, January 1991, vol.34, n° 1, pp.39-59.

[6] Ignat C., Norrie M.C.; "Customizable Collaborative Editor Relying on the treeOPT Algorithm"*;* Proc. 8th European Conf. on Computer Supported Cooperative Work (ECSCW'03), Helsinki, September 2003.

[7] Li D., Zhou L., Muntz R. R.; "A new paradigm of user intention preservation in realtime collaborative editing systems"; Proc. 7th Int. Conf. on Parallel and Distributed Systems (PADS'00), Iwate, Japan, July 2000.

[8] Mancini R., Dix A., Levialdi S.; "Reflections on Undo"; Technical report, University of Hudders eld, 1996.

[9] Molli P., Skaf-Molli H., Oster G.; "Divergence Awarenes for Virtual Team through the Web"; Proc 6th Int. Conf. on Integrated Design and Process Technology (IDPT'02), June 2002

[10] Palmer C.R., Cormack G.V.; "Operation Transforms for a Distributed Shared Spreadsheet"; Proc. ACM Int. Conf. on Computer Supported Cooperative Work (CSCW'98), Seattle, November 1998, pp. 69-78, 1998.

[11] Prakash A., Knister M.J.; "Undoing Actions in Collaborative Work"; Proc. ACM Int. Conf. on Computer Supported Cooperative Work (CSCW'92), November 1992, pp. 273-280.

[12] Prakash A., Knister M.J.; "A Framework for Undoing Actions in Collaborative Systems"; ACM Transactions on Computer-Human Interaction, vol. 1, n° 4, pp. 295-330, 1994.

[13] Ressel M., Nitssche-Ruhland D., Gunzenhäuser R.; "An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors"; Proc. ACM Int. Conf. on Computer Supported Cooperative Work (CSCW'96), Boston, November 1996, pp. 288-297.

[14] Ressel M., Gunzenhäuser R.; "Reducing the Problems of Group Undo"; Proc. ACM Int. Conf. on Supporting Group Work (GROUP'99), Phoenix, November 1999, pp.131-139.

[15] Suleiman M., Cart M., Ferrié J.; "Serialization of Concurrent Operations in a Distributed Collaborative Environment"; Proc. ACM Int. Conf. on Supporting Group Work (GROUP'97), Phoenix, November 1997, pp. 435-445.

[16] Suleiman M., Cart M., Ferrié J.; "Concurrent Operations in a Distributed and Mobile Collaborative Environment"; Proc. 14th IEEE Int. Conf. on Data Engineering (IEEE / ICDE'98), Orlando, February 1998, pp. 36-45.

[17] Suleiman M.; "Sérialisation des opérations concurrentes dans les systèmes collaboratifs répartis"; Doctoral thesis, Université de Montpellier 2, July 1998.

[18] Sun C., Jia X., Yang Y., Zhang Y.; "A generic operation transformation schema for consistency maintenance in real-time cooperative editing systems"; Proc. ACM Int. Conf. on Supporting Group Work (GROUP'97), Phoenix, November 1997, pp.425-434.

[19] Sun C., Ellis C.S.; "Operational Transformation in Real-Time Group Editors : Issues, Algorithms and Achievements"; Proc. ACM Int. Conf. on Computer Supported Cooperative Work (CSCW'98), Seattle, November 1998, pp. 59-68.

[20] Sun C.; "Undo Any Operation at Any Time in Group Editors "; Proc. ACM Conf. on Computer Supported Cooperative Work (CSCW'00), Philadelphia, Pennsylvania, December 2-6, 2000, pp. 191-200.

[21] Sun C.; "Undo as Concurrent Inverse in Group Editors"; ACM Transactions on Computer-Human Interaction, vol. 9, n° 4, pp. 309-361, December 2002.

[22] Vidot N., Cart M., Ferrié J., Suleiman M.; "Copies convergence in a distributed real-time collaborative environment"; Proc. ACM Int. Conf. on Computer Supported Cooperative Work (CSCW'00), Philadelphia, Pennsylvania, December 2-6, 2000, pp. 171-180.

[23] Vidot N.; "Convergence des Copies dans les Environnements Collaboratifs Répartis"; Doctoral thesis, Université de Montpellier 2, September 2002.