Grouping in Collaborative Graphical Editors

Claudia-Lavinia Ignat and Moira C. Norrie
Institute for Information Systems, ETH Zurich
ETH Zentrum, CH-8092 Zurich, Switzerland
{ignat,norrie}@inf.ethz.ch

ABSTRACT
Often collaborative graphical systems lag behind well accepted single-user applications in terms of features supported. The frequently used operations of group/ungroup offered by almost every single-user graphical editor have not been considered by the collaborative graphical editing systems that try to preserve the intentions of the users involved in the concurrent editing. In this paper we present a novel algorithm based on operation serialisation for consistency maintenance in collaborative graphical editing dealing not only with simple operations such as create, delete, move, change colour or position, but also with group/ungroup operations. Based on the classification of conflicts into real and resolvable, an undo/redo mechanism is used in order to re-execute the operations in an imposed serialisation order.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Network**]: Distributed Systems – *Distributed Applications*; D.2.2 [**Software Engineering**]: Designing Tools and Techniques; H.1.2 [**Models and Principles**]: User/Machine Systems– *Human Factors*; I.7.1 [**Document and Text Processing**]: Document and Text Editing

General Terms
Algorithms, Design, Human Factors

Keywords
Collaborative graphical editors, grouping/ungrouping, consistency maintenance, serialisation

1. INTRODUCTION
Within the CSCW field, collaborative editing systems have been developed to support a group of people simultaneously editing shared documents from different sites. Object-based graphical editing systems are a particular class of collaborative editing systems, where the shared objects subject to concurrent accesses are the graphic objects such as lines, rectangles, circles and text boxes.

The existing collaborative graphical editing systems are based on one of the following approaches: locking, serialisation or multi-versioning.

The *locking* approach [3] guarantees that users access objects in the shared workspace one at a time. Concurrent editing is allowed only if users are locking and editing different objects. Non-optimistic locking introduces delays for acquiring the lock. Optimistic locking avoids the delays, but it is not clear what to do when locks are denied and the object optimistically manipulated by the user must be restored to its original state. Another problem with the locking technique is to decide on the granularity of the locking, the choice being a balance between locking overhead and the amount of concurrency desired. Aspects [17], Ensemble [11] and GroupDraw [4] are systems relying on the locking technique.

Serialisation ensures that the effect of executing a group of concurrent operations is the same as if the operations were executed in the same total order at all sites. If there is any conflict among concurrent operations, only the effect of the last operation in the total ordering is maintained. LICRA [8] and GroupDesign [9] are examples of prototypes implementing this technique. LICRA relies on dependency relations between the operations and on the operational transformation mechanism [2], while GroupDesign uses an undo/redo mechanism for maintaining consistency.

The *multi-versioning* approach tries to achieve all operation effects, preserving the intentions of all operations. For each concurrent operation targeting a common object, a new version of the object is created. However, the multi-versioning approach raises some issues related to the graphical user interface, such as the way the versions of an object are related to the base object or the way the navigation through the versions of an object is realised. GRACE [15] and TIVOLI [10] are two prototype systems that rely on multi-versioning.

From the perspective of users, a cooperative editor should be an extension of a single user editor with additional functionalities for supporting group collaboration. Often collaborative systems lag behind well accepted single-user applications in terms of features supported. For example, group and ungroup operations are fundamental operations required in the editing process of graphical documents. Single user applications such as Microsoft Word, CorelDraw, Xfig or StarOffice offer the operations of grouping and ungrouping of objects. Most of the existing consistency maintenance algorithms for collaborative graphical editors deal only with elementary operations: create object, delete object, move object, change object colour. To our knowledge no other work investigated the issues related to grouping and ungrouping operations in a real-time environment where the intentions of the users involved in the concurrent editing are preserved.

In this paper we present an algorithm for consistency maintenance in the case of collaborative graphical editors dealing, not only with simple operations, but also with the operations of group and ungroup. Moreover, operations targeting individual objects of a group are allowed. For instance, it is possible to change the colour or the position of an object inside a group. One of the keys for offering the group/ungroup operations is the use of a hierarchical model of the document.

In our approach, we allow the members of the group to freely edit any part of the document. Three approaches can be adopted for dealing with conflicts: to reject the intentions of all users, to maintain the intention of only one of the users or to respect the intentions of all users by generating versions for the objects subject to conflict. We consider that a collaborative editing system should be general to be used in different domains that require collaboration e.g. news agency, medical applications, architectural design and musical notations that might require different policies for solving the conflicts. The collaborative graphical editing system we have implemented is a customizable editor allowing groups of users to choose a policy for dealing with concurrency, such as *null-effect based policy* where none of the operations in conflict are executed and the *priority based policy* where only the operation issued by the user with the highest priority wins the conflict.

We start our paper by describing the principles for maintaining consistency in the case of object-based graphical collaborative editing. We present the model of the document and the set of operations that can be performed, as well as their classification according to the type of conflict. We describe the algorithm, discuss its correctness and give an example illustrating its functionality. We also present some features of the collaborative graphical editor that we have implemented based on the proposed algorithm. In section 3 we compare our approach with some related work. Concluding remarks and the main directions of our future work are presented in the last section.

2. CONSISTENCY MAINTENANCE IN COLLABORATIVE GRAPHICAL EDITING

In this section we describe the mechanisms that we have used for maintaining consistency in object-based collaborative editing. We first describe how we model graphical documents, the operations that can be performed on the shared objects and the principles for consistency maintenance underlying our algorithm. We then classify the concurrent operations according to the types of conflict that occur between them.

2.1 Document model

A tree model is used for representing the scene of objects. Groups are represented as internal nodes, while simple objects are represented as leaves. A group can contain other groups or simple objects. Each object has a unique identifier assigned. The identifiers are uniquely generated at each site and are sent to the other sites, together with the information related to the create operation.

2.2 Operations

The operation types offered by our graphical editor are the following: *create*, *delete*, *group*, *ungroup*, *translate*, *scale*, *setColour*, *setBckColour*, *setZ* (setting the depth of the target object/group), *setText*.

An operation is identified by its *type*, the *initiator site identifier*, the *state vector* of the generating site, the *target list* containing the identifiers of the target objects of the operation, the *output list* containing the identifiers of the objects created by the operation. Each operation has associated a boolean value called *nop* showing whether the operation is valid or has been cancelled.

In the case of the *create* operation, the *target list* is empty. The *output list* is non-empty only in the case of *create* and *group* operations. Some types of operations have additional attributes, such as the colour in the case of *setColour* and *setBckColour*, the position in the case of *translate*, the size in the case of *scale*, the depth in the case of *setZ* and the text in the case of *setText*. Note that *ungroup* operation cannot be applied for the groups inside a group. It can be applied only for the uppermost group.

2.3 Principles for consistency maintenance

Our algorithm follows the same principles for consistency maintenance as presented in [16].

We start by defining the notions of causal ordering relations and concurrent operations.

Causal ordering relation "$\rightarrow$": Given two operations O_a and O_b generated at sites i and j respectively then O_a is causally ordered before O_b, denoted $O_a \rightarrow O_b$ iff: (1) $i=j$ and the generation of O_a happened before the generation of O_b; or (2) $i \neq j$ and the execution of O_a at site j happened before the generation of O_b; or (3) there exists an operation O_x such that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$.

Concurrent operations: Given any two operations O_a and O_b, O_a and O_b are said to be concurrent iff neither $O_a \rightarrow O_b$, nor $O_b \rightarrow O_a$. This is denoted $O_a || O_b$.

We define that two concurrent operations O_1 and O_2 are *conflicting* if one of the following three cases occurs:

- O_1 and O_2 intend to modify the same property (colour, background colour, position or depth coordinate) of a common target object to different values
- O_1 and O_2 intend to destroy one of the common target objects/groups (*delete* or *ungroup* operation)
- O_1 or O_2 intends to destroy one of the common target objects/groups (it is a *delete* or *ungroup* operation), while the other operation intends to modify that object or to use it in a grouping operation.

Note that if an operation targets a group of objects, we consider that it targets all the objects in the group. Therefore, an operation targeting a group and modifying a property of that group will be in conflict with any concurrent operation that targets an object/group belonging to that group and modifying the same property as the first operation. Similarly, an operation destroying a group is in conflict with any concurrent operation that either destroys an object from the group, intends to modify an object from the group or use it in a grouping operation. Also note that the *create* operation is not in conflict with any other operation, since no other operation can concurrently target the object created by create.

We now go on to present the consistency model underlying the algorithm. The model comprises the following consistency properties:

The *causality preservation* property requires that, for any pair of operations O_a and O_b, if $O_a \rightarrow O_b$, then O_a is executed before O_b at all sites. This means that at each site the effects of the execution of O_a are seen before the effects of the execution of O_b. However, during the serialisation process in the redo process the causality order might not be preserved. The redo process is performed in a very short period of time and not respecting the causality property for the undone operations during the reordering process will not be noticeable.

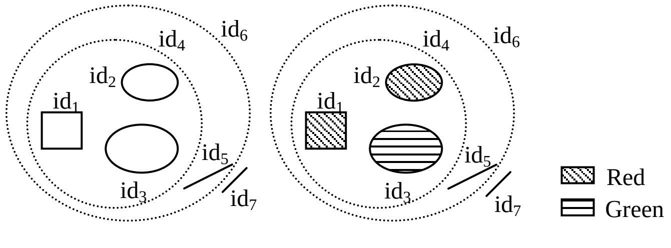
The *convergence* property requires that all copies of the same document are identical after executing the same set of operations.

The *intention preservation* property requires that, for any operation *O*, the effects of executing *O* at all sites are the same as the intention of *O* and the effect of executing *O* does not change the effects of non concurrent operations.

To achieve causality preservation, we use a time-stamping scheme based on a data structure called version vector[12] or state vector[2]. To achieve convergence, a serialisation process is used.

Preserving the intentions of the users is very difficult to achieve, because intentions seem to be defined at a higher level than that of a simple operation such as changing the position of an object. Only understanding the full context of the set of user actions makes it possible to find the user intention from a particular operation. We distinguish between syntactic and semantic consistency as described in [1]. Syntactic consistency means to reconcile the divergent copies to ensure that all sites have the same view, even if the view has no meaning in the context of the application. Syntactic consistency has been realised in our approach by using a serialisation mechanism. Semantic consistency, on the other hand, is defined specifically for an application and requires some extra techniques such as locking. By locking parts of the drawing, the users can define the context for applying their operation. By having exclusive access to that part of the drawing, the users ensure that the intent of their operation in the context of the locked part will be preserved. At the syntactic level, we achieve intention preservation by combining the effect of as many conflicting operations as possible. At the semantic level, we plan to implement a locking mechanism.

Group/ungroup operations increase the complexity of the algorithm for maintaining consistency in the case of collaborative graphical editors. Our aim is to maintain as many user intentions as possible in the case that a set of concurrent operations was issued. Consider the scene of objects in Figure 1a). Suppose two users concurrently edit this scene of objects.



a) before editing **b) after editing**

Figure 1. Combined effect of two concurrent operations: *setColour(id_4,red)* and *setColour(id_3,green)*

The first user wants to change the colour of the group of objects with the identifier id_4 to *red*, performing the operation O_1=*setColour*(id_4,*red*) while the second user concurrently wants to change the colour of the object having the identifier id_3 to *green*, performing the operation O_2=*setColour*(id_3,*green*). The two operations are conflicting because they both target the object id_3, one setting its colour to *red* and the other changing its colour to *green*.

One alternative to deal with such situation is to consider that only one of the two operations can be performed. Another alternative would be to consider a partial combined effect of the two intentions of the users, i.e. to change the colour of the object id_3 to *green* and the colours of the objects id_1 and id_2 to *red*, as shown in the Figure 1b). In this case we consider that the effect of an operation performed on an object overwrites the effect of an operation performed on the group to which the object belongs. The partial combined effect is obtained by the serialisation of the two operations: first the operation referring to the group followed by the operation performed on the individual object. We opted for the second alternative, since our aim is to allow the users to act freely and to combine the effect of the conflicting operations rather than maintaining the consistency by forbidding some actions of the users. By capturing the group intention as described above, some individual intentions might not be achieved. However, we believe that the scene of objects before the generation of conflicting operations can always be reestablished by the user by means of undo.

Our first attempt for maintaining the consistency of the copies of the shared graphical document was to use the operational transformation mechanism as in the case of the text editor that we have developed [5]. The operational transformation approach allows local operations to be executed immediately after their generation and remote operations need to be transformed against the other operations. Transformations are performed in such a manner that the intentions of the users are respected and, at the end, the copies of the documents converge. Due to the grouping/ungrouping operations and our aim of maintaining a partial combined effect of conflicting operations, the operational transformation approach was not suitable. The main reason is that transforming an operation targeting a group against a conflicting operation targeting an object belonging to that group would result in a set of operations targeting the individual objects from that group. This would lead to a sort of problems that we are going to highlight in what follows.

Consider the scene of objects illustrated in Figure 1a). Suppose that three users concurrently perform the following operations. At *Site*$_1$, the first user performs O_1=*setColour*(id_4,*red*), at *Site*$_2$, the second user issues O_2=*setColour*(id_1,*green*) and at *Site*$_3$, the third user issues O_3=*setColour*(id_6,*blue*). Let us analyse what happens at *Site*$_2$ in the case that after the execution of O_2, the operations O_1 and O_3 arrive in this order and the operational transformation mechanism is applied. When O_1 arrives at *Site*$_2$, the transformation of O_1 against O_2 would result in the list of operations [O'_{11}=*setColour*(id_2,*red*), O'_{12}=*setColour*(id_3,*red*)] in order to obtain the combined effect of O_1 and O_2. When O_3 arrives at *Site*$_2$, O_3 is first transformed against O_2 resulting in the list of operations [O'_{31}=*setColour*(id_2, *blue*), O'_{32}=*setColour*(id_3,*blue*), O'_{33}=*setColour*(id_5,*blue*)]. Further, the list of transformed operations [O'_{31}, O'_{32}, O'_{33}] needs to be transformed against the list of operations [O'_{11}, O'_{12}]. The pairs of operations O'_{31} and O'_{11} as well as O'_{32} and O'_{12} are conflicting and only the effect of one of them can be satisfied. Using only the information provided by the transformed operations without taking into account the target of the original operation they were generated from, it is not possible to specify rules for obtaining a combined effect of the conflicting operations. In the case of this example, it is not possible to specify the fact that a node in the tree should have the colour specified by the conflicting operation targeting its closest ancestor node (including the node itself). In order that the rules for obtaining the combined effect can be specified in the transformation functions, the transformed operation should include additional information.

Each operation needs to specify for its target object/group the ancestor group (including the node itself) from which the target object/group has inherited the property as the result of the transformation. For instance, O'_{31} should have the form *setColour*$(id_2,blue,id_6)$ meaning that as a result of the transformation of O_3 against O_2, object id_2 should have the colour blue, inherited from the colour of the group id_6. Therefore, the result of the transformation of $[O'_{31},O'_{32},O'_{33}]$ against $[O'_{11},O'_{12}]$ would be $[O''_{31},O''_{32}, O''_{33}]$, where $O''_{31}=setColour(id_2,red,id_4)$, $O''_{32}=setColour(id_3, red,id_4)$, $O''_{33}=setColour(id_5,blue,id_6)$.

There are cases when the sequence of operations that are the result of transformation should be considered as a group of operations when further transformations have to be performed. In the operational transformation approach, not only the inclusion transformation needs to be performed, but also the exclusion transformation. Inclusion transformation includes the effect of an operation into the context of another operation, while exclusion transformation achieves the inverse operation, i.e. excluding an operation from the context of the other one. Consider the set of operations O_1, O_2, O_3 and O_4 defined for the scene of objects in Figure 1a), where $O_1=group([id_6,id_7],id_8)$, $O_2=ungroup(id_6)$ and $O_3=setColour(id_6,red)$ and O_4 is an arbitrary operation. Consider that the relationships between the operations are as follows $O_1 \| O_2$, $O_1 \| O_3$, $O_2 \| O_3$, $O_3 \rightarrow O_4$, $O_1 \| O_4$, $O_2 \| O_4$. Further, suppose that, at the site where O_2 was issued, after the execution of the local operation O_2, the remote operations are executed in the order O_1,O_3,O_4. When O_1 arrives at the site it needs to be transformed against O_2. The transformed operation would be $O'_1=group([id_4,id_5,id_7],id_8)$. The transformed operation O_3 against $[O_2,O_1]$ is the list $[O'_{31},O'_{32}]$, where $O'_{31}=setColour(id_4,red,id_6)$ and $O'_{32}=setColour(id_5,red,id_6)$. According to any operational transformation algorithm, such as SOCT2[14], when operation O_4 arrives at the site, the history buffer $[O_2,O'_1,O'_{31},O'_{32}]$ needs to be reordered such that the operations preceding O_4, i.e. O'_{31} and O'_{32}, precede the operations concurrent with O_4, i.e. O_2 and O'_1. In order to obtain the reordering of the operations, operations O'_{31} and O'_{32} need to exclude the effect of O_2 and O'_1. If the effect of $[O_2,O'_1]$ is excluded in turn for the operations O'_{31} and O'_{32}, it would be very difficult to infer that the result of the exclusion for operations O'_{31} and O'_{32} form the original operation O_3. Moreover, an issue that needs to be discussed refers to the state vector associated with the operations O'_{31} and O'_{32}. In the case that both operations O'_{31} and O'_{32} would have associated the state vector of O_3, problems can arise when applying the operational transformation algorithms. As we have seen, considering separately each operation from the sequence of operations that are the result of the transformation would not be a good solution. In the case that the sequence of operations that are the result of transformation is treated as a group of operations, we need to define the operational transformation functions on lists of operations and not on simple operations, which will greatly increase the complexity if not making impossible the application of the transformation approach.

Due to many issues of applying the operational transformation mechanism for our approach, we instead adopted a serialisation mechanism which we are going to describe in what follows.

2.4 Conflicting operations

In the case of concurrent operations, two types of conflict can occur between the operations: real conflict and resolvable conflict.

Real conflicting operations are those conflicting operations for which a combined effect of their intentions cannot be established. We have defined that a pair of operations is real conflicting in the case that a serialisation order of execution of these operations cannot be obtained to preserve the intentions of the operations. By executing the two operations in any order, one of the following cases occur:

- the effect of the second executed operation will completely make invisible the effect of the first executed operation
- the execution of the first operation will not make possible the execution of the second operation

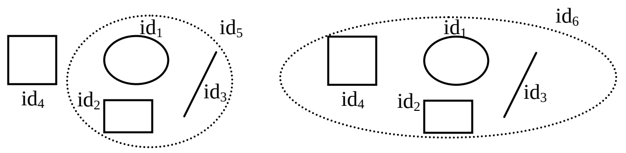
An example of real conflicting operations are the two concurrent operations *setColour*(id_1,red) and *setColour*$(id_1,blue)$, both targeting the same object and changing the colour of that object to different values.

In Table 1, a list of the real conflicting operations is given. The first column contains operations belonging to the history buffer, the second column remote causally ready operations and the third column the condition that a real conflict occurs between the operation from the history buffer and the remote operation. The same real conflicting situations occur if the first column of the table represents the remote operations and the second column the operations from the history buffer.

Table 1. Real conflicting operations

Operation from the HB	**Remote operation**	**Condition**
delete(id_1)	*ungroup*(id_2)	$id_1=id_2$
group$(inList_1,gid_1)$	*group*$(inList_2,gid_2)$	$inList_1 \cap inList_2 \neq \emptyset$
setColour(id_1,c)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2), \forall c$
setColour(id_1, c_1)	*setColour*(id_2,c_2)	$id_1=id_2, \forall c_1 \neq c_2$
setBckColour(id_1,c)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2), \forall c$
setBckColour(id_1,c_1)	*setBckColour*(id_2,c_2)	$id_1=id_2, \forall c_1 \neq c_2$
translate(id_1,d_x,d_y)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2), \forall d_x,d_y$
translate(id_1,d_{x1},d_{y1})	*translate*(id_2,d_{x2},d_{y2})	$id_1=id_2, \forall (d_{x1},d_{y1}) \neq (d_{x2},d_{y2})$
scale$(id_1,x_{ref},y_{ref}, \Delta_x, \Delta_y)$	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2), \forall x_{ref},y_{ref},\Delta_x, \Delta_y$
scale$(id_1,x_{ref1},y_{ref1}, \Delta_{x1}, \Delta_{y1})$	*scale*$(id_2,x_{ref2},y_{ref2}, \Delta_{x2}, \Delta_{y2})$	$id_1=id_2, \forall (x_{ref1},y_{ref1},\Delta_{x1},\Delta_{y1}) \neq (x_{ref2},y_{ref2},\Delta_{x2},\Delta_{y2})$
setText(id_1, t)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2), \forall t$
setText(id_1,t_1)	*setText*(id_2,t_2)	$id_1=id_2, \forall t_1 \neq t_2$
setZ(id_1, z)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2), \forall z$
setZ(id_1, z_1)	*setZ*(id_2,z_2)	$id_1=id_2, \forall z_1 \neq z_2$

Resolvable conflicting operations are those conflicting operations for which a partial combined effect of their intentions can be obtained by serialising those operations. Consequently, ordering relations can be defined between any two concurrent operations. Any two resolvable conflicting operations can be defined as being in the right order, or in the reverse order. Consider that two users concurrently edit the scene of objects illustrated in Figure 2a). Suppose that at $Site_1$ the first user wants to group the object having the identifier id_4 with the group having the identifier id_5. Further, suppose that, concurrently, at $Site_2$ the second user performs an *ungroup* operation upon the group id_5.



a) before editing **b) after editing**

Figure 2. Combined effect of two concurrent operations O_1= *group([id_4,id_5],id_6)* and O_2=*ungroup(id_5)*

In this case, the intentions of both users can be preserved, i.e. at the end a group consisting of the object with the identifier id_4 and the objects belonging to the group with the identifier id_5 will be created. But, in order to obtain this result, the two operations need to be executed in a certain order at both sites, i.e. *group([id_4,id_5],id_6)* before *ungroup(id_5)*. In the case that this order of execution is not respected at $Site_2$, after the *ungroup* operation is executed, the group identified by id_5 being destroyed, the *group* operation will target an object/group that does not belong to the current structure of the document.

In Table 2, a list with resolvable conflicting operations is given. The first column contains the operations belonging to the history buffer, the second column the remote causally ready operations and the third column the condition that a resolvable conflict occurs between the operation from the history buffer and the remote operation. The fourth column shows the order of execution of the remote operation related to the local operation. Right order indicates that the remote operation should be executed after the indicated operation from the history buffer and reverse order indicates that the remote operation should be executed before the operation from the history buffer. As in the case of the table of real conflicting operations, symmetric cases of resolvable conflicting operations occur if the first column of the table represents the remote operations and the second column the operations from the history buffer, but the order of serialisation will be the reverse order specified in the fourth column.

Table 2. Resolvable conflicting operations

Operation from the HB	**Remote Operation**	**Condition**	**Order**
delete(id_1)	*group*($inList_2$,id_2)	$id_1 \in inList_2$	Reverse
delete(id_1)	*delete*(id_2)	$id_2 \in subTree(id_1)$	Reverse
ungroup(id_1)	*setColour*(id_2,c)	$id_1=id_2$, $\forall c$	Reverse
ungroup(id_1)	*setBckColour*(id_2,c)	$id_1=id_2$, $\forall c$	Reverse
ungroup(id_1)	*translate*(id_2,d_x,d_y)	$id_1=id_2$,$\forall d_x$,d_y	Reverse
ungroup(id_1)	*scale*(id_2,x_{ref},y_{ref}, Δ_x,Δ_y)	$id_1=id_2$, $\forall x_{ref}$,y_{ref},Δ_x,Δ_y	Reverse
ungroup(id_1)	*setZ*(id_2, z)	$id_1=id_2$,$\forall z$	Reverse
setColour(id_1,c_1)	*setColour*(id_2,c_2)	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall c_1 \neq c_2$	Reverse
setBckColour(id_1, c_1)	*setBckColour*(id_2, c_2)	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall c_1 \neq c_2$	Reverse
translate(id_1, d_{x1},d_{y1})	*translate*(id_2, d_{x2},d_{y2})	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall (d_{x1},d_{y1}) \neq (d_{x2},d_{y2})$	Reverse
scale(id_1,x_{ref1}, y_{ref1},Δx_1,Δy_1)	*scale*(id_2,x_{ref2}, y_{ref2},Δx_2,Δy_2)	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall (x_{ref1},y_{ref1},\Delta_{x1},\Delta_{y1}) \neq (x_{ref2},y_{ref2},\Delta_{x2},\Delta_{y2})$	Reverse
setZ(id_1,z_1)	*setZ*(id_2,z_2)	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall z_1 \neq z_2$	Reverse

An exhaustive analysis has been performed to obtain the tables with the real conflicting operations and with the resolvable conflicting operations, each pair of types of operations being considered.

The notion of conflict and the types of conflict can be defined also by using the notion of commutation and serialization among operations. A conflict corresponds to concurrent operations that do not commute. A resolvable conflict corresponds to concurrent operations that do not commute, but can be serialised. A real conflict corresponds to the case that the concurrent operations do not commute and cannot be serialised.

The operation serialisation mechanism involves the reordering of the operations from the history buffer. Between the non concurrent operations from the history buffer there exist follow and masking relationships that do not allow two operations to be executed in reverse order during the serialisation process.

An operation O_2 from the history buffer is said to *follow* a preceding operation O_1 from the history buffer (O_2 follows O_1 or O_1 followed by O_2) if either O_2 *depends* on O_1, i.e. O_1 creates an object or group that belongs to the target list of O_2 or O_2 destroys (by deleting or ungrouping) the target of O_1. A follow operation cannot be executed before the operation it follows.

An operation O_2 from the history buffer is *masking* an operation O_1 preceding O_2 in the history buffer if O_1 and O_2 are of the same type (except create, delete, ungroup), both targeting the same object/group and O_2 overwriting the effect of O_1. A masking operation cannot be executed before the masked operation.

For example, consider *HB*=[*group([id_1,id_2],id_3)*,*move(id_3)*, *setColour(id_4,red)*,*setColour(id_4,green)*]. Operation *move(id_3)* follows the operation *group([id_1,id_2],id_3)* that created its target object. Operation *setColour(id_4,green)* masks the effect of *setColour(id_4, red)*.

2.5 Consistency maintenance algorithm

After defining the relations among operations, we can go on to present the algorithm.

2.5.1 Description of the algorithm

Each site involved in the editing process stores locally a copy of the shared hierarchical document. Each site maintains a history buffer containing the executed operations at that site. The site generating the operation executes it immediately and records it in the history buffer. Finally, the operation is broadcast to all other sites, being timestamped using a state vector.

Upon receiving a remote operation, the receiving site will test it for causally readiness. If the operation is not causally ready, it will be queued, otherwise a serialisation process will be applied. Aiming to preserve the intentions of as many users as possible during the concurrent editing involving group/ungroup operations, the complexity of the serialisation mechanism is higher than in the case of other collaborative systems that use serialisation but do not deal with groups of objects.

Given the history buffer *HB=[O_1,…,O_m,…,O_n]*, the main idea underlying the procedure of execution of a new causally ready operation O_{new} in the case of the priority based policy is presented in what follows.

Firstly, O_{new} is checked for whether it depends on any cancelled operation from the history buffer. If it is the case, O_{new} is cancelled too and added at the end of the history buffer. Otherwise, the history is traversed again from left to right till a concurrent operation O_m in conflict with O_{new} is found. If no such operation is found, the remote operation is executed and appended to the history buffer. Otherwise, all operations in the list $HB_{[m,n]}$ (denoting the *HB* list starting at index *m* and ending at index *n*) will be undone. A list *Real_Conflict* containing the operations from $HB_{[m,n]}$ in real conflict with O_{new} is created. A list called *Right_Order* is created to contain all the operations belonging to $HB_{[m,n]}$ that have to be executed before O_{new}. The *Right_Order* list will contain those operations that are in a right order resolvable conflict relation with O_{new} and the operations that are masked by O_{new}. The list of operations *Reverse_Order* is created to contain all the operations belonging to $HB_{[m,n]}$ that have to be executed after O_{new}. The *Reverse_Order* list contains the operations from $HB_{[m,n]}$ that are in a reverse order relation with O_{new}, the operations that follow or mask the operations from *Reverse_Order* and the operations for which a right order relationship has been established with the operations from *Reverse_Order*. Before inserting an operation into *Right_Order* or *Reverse_Order* list, a check has to be performed whether the operation belongs to *Real_Conflict* list. In the case the operation belongs to *Real_Conflict* list, it will not be inserted into *Right_Order* or *Reverse_Order* list, respectively.

In the case that *Real_Conflict* is not empty, either the remote operation or all the local conflicting operations will be executed. The decision as to which of the operations is to be executed, the remote operation or the operations belonging to the *Real_Conflict*, depends on the policy chosen by the application. Note that no pair of operations in the *Real_Conflict* are in a real conflict relation, otherwise they would not have been integrated both into the history buffer. In the case of conflict between two operations, an application might choose to execute none of the conflicting operations or might choose to execute one of the conflicting operations based on some criteria, such as priorities for the users. According to the priority based policy in the case that *Real_Conflict* is not empty, the remote operation is executed if its priority is greater than any priority of the conflicting operations and it is cancelled otherwise.

In what follows we will use the term NOP operation to denote those operations that are cancelled, i.e. the attribute *nop* of the operation is set to true. The NOP operations are considered by the serialisation algorithm only for checking whether a remote operation depends on a NOP operation.

The idea underlying the algorithm for operation serialisation is to redo the undone operations from the history buffer in such a way that the operations from *Right_Order* are re-executed before the remote operation and the operations from *Reverse_Order* are re-executed after the remote operation.

If *Real_Conflict* is empty, then the following steps are performed:

(i) the operations in the history buffer that have been undone, except the operations in the *Reverse_Order*, are redone respecting their initial order in the history buffer

(ii) O_{new} is executed

(iii) the operations from the *Reverse_Order* are re-executed.

If *Real_Conflict* is not empty, either the remote operation is chosen as the winning operation or the operations in the *Real_Conflict* list are chosen as winning operations.

(a) If the winning operation is not the remote operation, then the following steps are performed:
   (i) all the undone operations from the history buffer are re-executed
   (ii) O_{new} is appended to the history buffer as a NOP operation.

(b) If the winning operation is the remote operation, then the following steps are performed:
   (i) all operations from the list *Real_Conflict* as well as the operations that depend on them are set to NOP
   (ii) the operations from the history buffer, except the operations from *Reverse_Order* are redone respecting their initial order in the history buffer
   (iii) O_{new} is executed
   (iv) the operations from *Reverse_Order* are re-executed in the order they appeared in the history buffer.

2.5.2 Discussion on the correctness of the algorithm

The history buffer is a mixture of real conflicting operations and right order or reverse order resolvable conflicting operations. In the case of re-executing the undone operations from the history buffer, a problem that might appear is the case that an operation from *Right_Order* belongs to *Reverse_Order*. In this case no reordering of operations is possible. However, given the pairs that are in a resolvable conflict, such a situation occurs only in some particular cases that are handled by the algorithm.

We did not develop a theoretical model for the abstraction of the problem in order to be able to prove the correctness of the algorithm. However, in what follows we are going to analyse the set of operations that might cause the occurrence of the case that an operation belongs to both *Reverse_Order* and *Right_Order*.

For checking whether a remote operation can be at the same time in a right order and reverse order relation with the operations in the history buffer, the resolvable conflicting pairs of operations from Table 2 have to be analysed. One of the remote operations that can be in a resolvable conflict, both right order and reverse order with operations from history buffer is the *delete* operation. But, actually, this case cannot occur. In order that the remote *delete* operation targeting *G* is in a right order with the group operation whose target list contains *G*, *G* needs to be the uppermost group. But, in order that the remote *delete* operation is in a reverse order resolvable conflict with a *delete* operation targeting G_1, $G \in \text{subtree}(G_1)$, which is a contradiction with the fact that *G* is the uppermost group.

The only remote operations that can be in a resolvable conflict, both right order and reverse order with operations from the history buffer are *setColour*, *setBckColour*, *translate*, *scale* and *setZ*. The behaviour of these operations is similar, so without loss of generality, we analyse only the case of *setColour* as a remote operation. The concurrent operations belonging to the history buffer that can be in a resolvable conflict with the remote operation *setColour* are *ungroup* and *setColour*. We divide our analysis in two main parts, depending on the considered operation in reverse order with the remote operation, i.e. the *ungroup* or *setColour* operations.

For the first part of our analysis, the first case that we need to examine is the one for which the remote operation is the *setColour* operation targeting the group *G*, the *ungroup* operation targeting the group *G* belongs to the *Reverse_Order* list corresponding to the remote operation and the *setColour* operation targeting a supergroup of *G*, denoted *SupG*, belongs to the *Right_Order* list. We have to analyse whether *ungroup* or any other operation belonging to *Reverse_Order* can belong to *Right_Order*.

The case that the *ungroup* operation targeting *G* appears in the history buffer before the *setColour* operation targeting *SupG* cannot occur. An *ungroup* operation cannot be performed on a subgroup, *G* being the subgroup of *SupG*. We might think that *SupG* is created after the *ungroup* operation has been performed. But, once an *ungroup* operation has been executed, the regrouping of the elements that have been ungrouped will not generate a new group with the same identifier, so *SupG* cannot be inferred to be the supergroup of *G*.

The case that the *setColour* operation targeting *SupG* appears in the history buffer before the *ungroup* operation targeting *G* does not cause problems because the *ungroup* and any operation that masks or follows the *ungroup* or any other operation from *Reverse_Order*, as well as any operation being in right order resolvable conflict with *ungroup* or any other operation from *Reverse_Order* are positioned to the right of the ungroup operation. Therefore, it is not possible that one of the operations from *Reverse_Order* belongs to *Right_Order*.

The second situation for the remote operation *setColour* targeting *G* to have both a corresponding operation belonging to *Right_Order* list and *Reverse_Order* list is that *ungroup* operation targeting *G* belongs to *Reverse_Order* and the *setColour* operation targeting a subgroup of *G*, denoted *SubG* is masked by the remote operation and it belongs to the *Right_Order* list. The case that the operation from *Right_Order* list precedes in the history buffer the operation from the *Reverse_Order* list does not cause any problems. Let us analyse in what follows the case that the *ungroup* operation targeting *G* precedes in the history buffer the *setColour* operation targeting *SubG*. There is no operation that is in a right order resolvable conflict with the *ungroup* operation and no operation masks the *ungroup* operation. The only operation that is in a follow relation with the *ungroup* operation is the *group* operation applied on some of the subgroups of *G* and creating *NewG*. The *group* operation cannot belong to the *Right_Order* list. Therefore, we go further by analysing the *group* operation. The operation in a right order relation with *group* is *delete* applied to one of the target objects of group operation, denoted *SubgroupG*. But the *delete* operation does not belong to *Right_Order* list. Moreover, there is no operation that masks it, no operation that is in a follow relation with it. The only operation that is in a right order relation with the *delete* operation is the *delete* operation targeting a parent of *SubgroupG*, which leads to the recursive analysis of *delete*. Therefore, the *delete* operation does not generate new operations that belong to *Reverse_Order* list. Returning to the analysis of the *group* operation, there is no operation that masks the *group* operation and the operations that are in a follow relation with *group* are *setColour* targeting *NewG* and *group* operation grouping the *NewG* with other objects/groups. The *group* operation targeting *NewG* leads to the recursive analysis of the *group* operation. The operations that are in a follow relation with *setColour* operation targeting *NewG* are *ungroup* and *delete* operations targeting *NewG*. The *ungroup* operation targeting *NewG* will lead to the recursive analysis of the *ungroup* operation, the starting point of our analysis. The *delete* operation cannot belong to *Right_Order* and there are no operations that mask or are in a follow or right order relation with the *delete* operation (except *delete* operation that does not generate new cases to be analysed). The operation that masks *setColour* operation targeting *NewG* is the *setColour* operation targeting a parent of *NewG* which does not belong to *Right_Order* list. The operation that is in a right order relation with *setColour* operation targeting *NewG* is the *setColour* operation targeting a subgroup of *NewG*, which might be *SubG*. Therefore, this yields to the case that an operation from *Reverse_Order* belongs to *Right_Order* list, this operation being a masked operation by the remote operation. The solution that we have adopted is to eliminate from the *Reverse_Order* list the masked operations.

In the second part of our analysis we need to consider that the operation that is in a reverse order relation with the remote operation *setColour* targeting *G* is the *setColour* operation targeting a subgroup of *G*, denoted *SubG*. As in the first part of our analysis, we are going to analyse, in turn, the cases that the *Right_Order* list contains the *setColour* operation targeting a supergroup of *G*, denoted *SupG* and the *setColour* operation targeting *SubG*.

For the first case, we are going to analyse the situation that the *setColour* operation targeting *SubG* appears before the *setColour* operation targeting *SupG* in the history buffer. As previously mentioned, the case that the operations in *Right_Order* list appear before the operations in *Reverse_Order* list in the history buffer does not cause problems. The operations that are in a follow relation with *setColour* targeting *SubG* are *ungroup* and *delete* operations both targeting *SubG*. The *delete* operation cannot belong to *Right_Order* and there are no operations that mask or are in a follow or right order relation with the *delete* operation (except delete operation that does not generate new cases to be analysed). As we have already seen, the *ungroup* operation targeting *SubG* cannot appear before the *setColour* operation targeting *SupG*, since it is not possible to perform an ungroup of a subgroup. The operation that is in a right order relation with *setColour* operation targeting *SubG* is *setColour* operation targeting a subgroup of *SubG* which leads to the initial recursive analysis of the *setColour* operation. The operation that masks the *setColour* operation targeting *SubG* is the *setColour* operation targeting a parent of *SubG*, which might belong to the *Right_Order* list. This yields to the case that an operation from *Reverse_Order* belongs to *Right_Order*. The solution that we have adopted for this puzzle is to eliminate from the *Reverse_Order* list the masked operations.

The second case that is left for discussion is the one for which the *Reverse_Order* list contains the *setColour* operation targeting the subgroup of *G*, *SubG* and the *Right_Order* list contains the *setColour* operation targeting another subgroup of *G*, denoted $SubG_1$, masked by the remote operation. Following the same analysis as in the previous case, the only operation belonging to the *Reverse_Order* that might belong to the *Right_Order* list is the *setColour* operation targeting $SubG_1$ as an operation in a right order relation or as a masking operation of the *setColour* operation targeting *SubG*. The solution that we have adopted is to eliminate from the *Reverse_Order* list the masked operations.

The analysis of the cases that the *Reverse_Order* list contains a set of operations can be reduced to the analysis of the above described cases containing only one operation in the *Reverse_Order* list.

We can reduce our approach to a graph problem. The operations that need to be serialised are represented as the nodes of a directed graph. Between two operations O_1 and O_2, there is an arc directed from O_1 to O_2 iff O_1 has to be executed before O_2. When a new operation needs to be executed, all the relations among the operations in the history buffer and the remote operation are represented in the graph: the right order, the reverse order, the follow and the masking relations. The operations that are in a real conflict and are cancelled are marked as NOP operations. A serialisation order is a topological sort order of the represented directed graph. The condition of existence of a serialisation order is that the directed graph contains no cycles. In our approach we define the pairs of conflicting operations: real conflicting, right order, reverse order, follow and masking relations among the operations in order to satisfy the semantics of our application. We have shown that the order that we have defined among the operations yields no cycles in the graph with the exception of some particular cases, for which the algorithm is adapted in order to eliminate the cycle. However, different ordering relations among operations can be defined to adjust to the semantics of the application with the restriction of no cycles.

2.5.3 Example

For a better understanding of the algorithm, we illustrate its functionality by means of an example.

Consider the scene of objects shown in Figure 3. It consists of a group having the identifier *id_3* composed of two objects with the identifiers *id_1* and respectively *id_2*. The scene of objects contains also other two objects with the identifiers *id_4* and *id_5*.

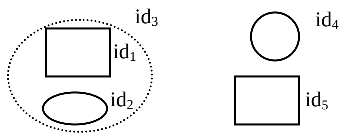


Figure 3. The initial scene of objects

Suppose three users concurrently edit this graphical document, as illustrated in Figure 4. The user at *$Site_1$* groups the group with the identifier *id_3* with the object having the identifier *id_5* and changes the colour of the newly created group into *red*. Concurrently, the user at *$Site_2$* wants to ungroup the group *id_3* and to change the colour of the object *id_1* into *blue*. At the same time, the user at *$Site_3$* performs a grouping of group *id_3* with the object *id_4*. In the case of real conflicting operations we consider that the operation generated from a site with the highest priority wins the conflict. Let us consider that *$Site_1$* has priority 3, *$Site_2$* has priority 2 and *$Site_3$* has priority 1. Let us analyse the application of the algorithm at each of the three sites.

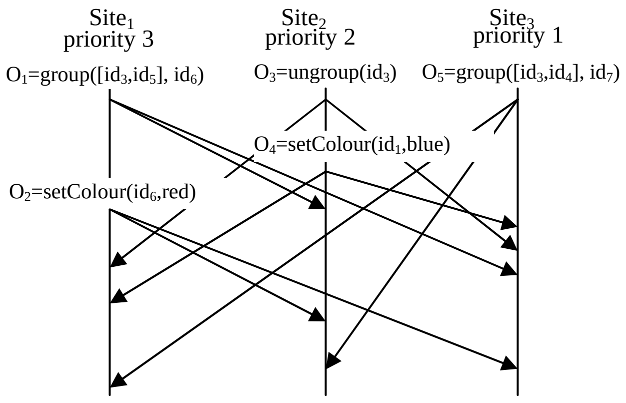


Figure 4. Scenario – functionality of the algorithm

At *$Site_1$*, after O_1 and O_2 are executed, *HB*=[O_1,O_2]. When the causally ready operation O_3 is received, the algorithm detects that O_3 is concurrent and in conflict with O_1, so O_2 and O_1 are undone. The conflict between O_1 and O_3 is a right order resolvable conflict. So, O_1 and O_2 are redone and afterwards O_3 is executed. So, *HB*=[O_1,O_2,O_3]. When the causally ready operation O_4 is received, the algorithm detects that O_4 is concurrent and in conflict with O_2, so O_3 and O_2 are undone. The conflict between O_2 and O_4 is a right order resolvable conflict, the *Reverse_Order* list is empty, so, O_2 and O_3 are redone and O_4 executed. After the execution of O_4 *HB*=[O_1,O_2,O_3,O_4]. When O_5 is received, the real conflict between O_5 and O_1 and the reverse order resolvable conflict between O_3 and O_5 are detected. All operations from *HB* are undone. Because the priority of O_1 is higher than the priority of O_5, O_5 becomes NOP operation, the operations from *HB* are redone and O_5 will be added at the end of the history buffer. So, finally *HB*=[O_1,O_2,O_3, O_4,O_5=NOP]. Note that according to an optimisation of the undo/redo mechanism, the operations from the history buffer need to be undone only in the case of the reordering of the operations.

At *$Site_2$*, after O_3 and O_4 are performed, *HB*=[O_3,O_4]. When O_1 is received, the algorithm detects that O_3 is in a reverse order resolvable conflict with O_1, so O_4 and O_3 are undone. The *Reverse_Order* list contains only the operation O3, so O_4 is redone, O_1 executed and O_3 redone. So, *HB*=[O_4,O_1,O_3]. When O_2 is received, the reverse order resolvable conflict between O_4 and O_2 is detected, O_3, O_1 and O_4 are undone. The *Reverse_Order* list contains only the operation O_4. O_1 and O_3 are redone, O_2 executed and O_4 redone, the history buffer becoming *HB*=[O_1,O_3,O_2,O_4]. When O_5 is received, the real conflict between O_5 and O_1 is detected and all operations from *HB* are undone. Because the priority of O_1 is higher than the priority of O_5, O_5 becomes NOP, the operations from *HB* are redone and O_5 will be added at the end of the history buffer. So, finally *HB*=[O_1,O_3,O_2,O_4,O_5=NOP].

At *$Site_3$*, after the execution of O_5, *HB*=[O_5]. When O_4 is received, its execution will be delayed, because it is not a causally ready operation. When O_3 is received, the algorithm detects that O_5 and O_3 are in a right order conflict, O_5 is undone, then redone, followed by the execution of O_3. Afterwards, O_4 becomes ready for execution. No conflict between O_4 and the operations in the history buffer is detected, so O_4 is added at the end of the history buffer. So, *HB*=[O_5,O_3,O_4]. When O_1 is received, the real conflict between O_1 and O_5 is detected, as well as the reverse order resolvable conflict between O_3 and O_1, the *Reverse_Order* list containing only O_3. The operations in the history buffer are undone, O_5 becomes NOP, O_4 is redone, O_1 is executed, followed by the re-execution of O_3. So, *HB*=[O_5=NOP, O_4,O_1,O_3]. When O_2 is received, the reverse order resolvable conflict between O_2 and O_4 is detected, O_3, O_1 and O_4 are undone. Since *Reverse_Order* list contains only O_4, then O_1 and O_3 are redone, O_2 is executed and O_4 is redone. So, *HB*=[O_5=NOP,O_1,O_3,O_2,O_4].

In the case of an application that adopts the null-effect policy, the algorithm is simpler. When a concurrent remote operation in conflict with some of the operations from the *HB* is received, all the conflicting operations from *HB* will become NOP operations as well as their dependent operations. The remote operation is also cancelled.

We have implemented a multi-document real-time collaborative graphical editor relying on the algorithm presented in the previous section [6]. Users can join or leave the editing of any document or of the whole session whenever they want. At any moment of time, a user is aware of the other users concurrently editing a document. Users are also informed by means of messages that appear on the lower part of the editor in the case that a conflict cancelled their operations.

Our real-time graphical editor is customizable, offering to the users two policy modes for dealing with concurrency: the null-effect based policy that ensures that none of the real conflicting operations is executed and priority based policy according to which a remote operation is executed only if it has the highest priority among a set of conflicting operations. Given that users have assigned priorities according to the application domain, the priority of an operation equals the priority of the user that issued the operation.

A garbage collection scheme has been implemented for removing those operations from the history buffer that are no longer used in the serialisation process. The garbage collection scheme is similar to the one implemented in REDUCE [16].

3. RELATED WORK

In this section we compare our approach for maintaining consistency in collaborative graphical editing with some related works. From the existing collaborative graphical editing systems, the closest to our work are GroupDesign and LICRA that use the serialisation mechanism for maintaining consistency and the dARB approach that uses a hierarchical document model.

The ORESTE (Optimal RESponse TimE) algorithm [9] underlying the GroupDesign system uses a history buffer with the executed operations and a queue with received operations that cannot be executed because the object to which they are applied is not yet created. Timestamps are used to define the total order among operations. When a remote operation is received, the operation is checked to see whether it applies to an object that was not yet created. If this is the case, the operation is queued into the list of operations applying to nonexistent objects. Otherwise, the local logical time is compared with the timestamp of the operation: if the time of the received operation is greater than the local time, then the operation is executed immediately, otherwise all operations from the history buffer more recent than the received one are undone. The received operation is executed and the undone operations are redone. Relationships of commutation and mask are used in order to reduce the undo/redo number of operations. The ORESTE algorithm does not check if an operation is causally ready. Two operations of the same type, different than of type *create*, generated in a certain order at a site, are executed in their order of arrival at another site. In the GroupDesign approach a masked operation is not inserted into the history buffer. Therefore, the masked operation cannot be undone, which is a restriction for the undo mechanism. A *delete* operation masks a concurrent operation modifying a property of the object to be deleted, such as *setColour* and *translate*. This is not always the expected behaviour. We consider that these two types of operations have equal priorities and define them as real conflicting operations, the application deciding, according to the policy adopted, which of these operations to execute.

The LICRA (Lock-free Interactive Concurrency Resolution Algorithm) approach [8] relies on direct dependency relations between generated operations as well as on an operation transformation mechanism. The direct dependency relations between generated operations are used instead of state vectors for causality preservation, i.e. when an operation is propagated to the other sites, the message contains also the identifier of the last operation executed at the site. Relationships of commutation, masking and conflict between operations are used in the operational transformation process, the limitations being the same as the ones mentioned in the GroupDesign approach. Besides a history buffer containing the list of executed operations, each site maintains a set of operation lists which hold the received operations that cannot be executed because some precedent operations were not yet received and also a list of waiting operations. A disadvantage of LICRA is that the number of sites involved in the editing process needs to be constant, so a user cannot dynamically join/leave the group. In addition to the operation semantics, operation transformation depends also on the priority of originators sites. The priority function is simply defined as a total order over the set of site identifiers.

The main difference between our approach and GroupDesign and LICRA is that we deal with operations of grouping/ungrouping and we try to satisfy the intentions of most users issuing concurrent operations including group/ungroup operations. The intentions for concurrent operations involving not only objects, but also groups of objects, cannot be preserved using the mechanisms proposed by GroupDesign and LICRA.

The dARB [7] algorithm uses a tree model for document representation, as in our approach, however it is not able to automatically resolve all concurrent accesses to documents and, in some cases, must resort to asking the users to manually resolve inconsistencies. Their approach is similar to the dependency detection approach for concurrency control in multi-user systems where operation timestamps are used to detect conflicting operations and the conflict is then resolved through human intervention [13]. Further, there are cases when one site wins the arbitration and it needs to send, not only the state of the vertex itself, but maybe also the state of the parent or grandparent of the vertex. The dARB algorithm was presented applied to text documents. To prove its generality, the dARB algorithm was described as applicable also for a scene of objects. However, grouping and ungrouping features were not implemented, the level of the tree modeling the scene of objects being 2.

In what follows we will compare the techniques we have used for maintaining the consistency in the case of the collaborative graphical editor and the ones we have used in the collaborative text editor that we have implemented [5].

The principles for maintaining the consistency for the text and graphical collaborative editors are the same: causality preservation, convergence and intention preservation. For achieving causality preservation, state vectors have been used in both approaches. However, for achieving convergence and intention preservation, different techniques have been used: operation transformation in the case of the text editor and operation serialisation in the case of the graphical editor. In the case of the collaborative text editor, we have modeled the document also as a tree, consisting of a set of paragraphs, each paragraph as a set of sentences, each sentence as a set of words and each word as a set of characters. Each semantic unit (paragraph, sentence, word and character) can be uniquely identified by its position in the sequence of the child elements of its parent. The operations allowed to be performed on the semantic units are insertion and deletion. For achieving consistency, the insertion and deletion operations on these elements may shift the positions of the elements in the sequence of the child elements of their parent for adapting to the effect of concurrent operations. In the case of the graphical documents, the objects are not organised into sequences and identified by their position in the sequence. Rather, they are identified by unique identifiers and there is no need to adapt the identifiers due to the concurrent operations.

Graphical objects in the case of graphical documents have associated properties such as colour and position that are subject to concurrent operations. In the case of text documents, the elements in the tree model have no associated properties. In the case we want to associate different fonts and styles to elements of the text document, we could represent those elements using object identifiers and introduce corresponding operations such as *setFontSize* and *setStyle*.

As explained in the paper, due to grouping/ungrouping operations and our aim to maintain a partial combined effect in the case of conflicting operations, the operational transformation approach was not suitable for graphical editing.

Generally, we can say that operational transformation is suitable for systems that adopt a linear or hierarchical representation for data organization and the operations that can be performed on the data are insertions and deletions that may shift the positions of the elements. Operation serialisation is suitable for systems in which data objects do not require a sequential representation and the operations that can be performed modify the properties of the objects. Moreover, an order of execution of concurrent operations with overlapping effect needs to be specified in order to achieve convergence and intention preservation.

4. CONCLUSIONS AND FUTURE WORK

In this paper we presented a mechanism for consistency maintenance in collaborative graphical editing that deals with operations of group/ungroup. We have highlighted the complexities that appear for preserving the intentions of as many users as possible in the case of concurrent conflicting operations including the grouping/ungrouping operations. The proposed mechanism consists of operation serialisation according to state vectors but also using some ordering rules in the case of concurrent operations. The algorithm that we presented in this paper can be applied not only for the real-time communication, but also for the asynchronous communication.

Extending our graphical collaborative editor with the possibility of locking objects is one of the future functionalities to be integrated in the real-time collaborative editor. In some applications, the users would like to protect parts of the graphical document from the editing of some other users. The solution for this problem is to permit the users to lock the group of objects to which they want to have exclusive access. The graphical editor application needs to be extended by offering proper local and global undo mechanisms.

We are also investigating ways of optimizing the algorithm for consistency maintenance, such as reducing the number of undo operations that need to be performed when a new remote operation is integrated into the history buffer.

5. REFERENCES

[1] Dourish, P. Consistency guarantees: Exploiting application semantics for consistency management in a collaboration toolkit. *Proc. of ECSCW'95,* Stockholm, Sweden, Sept. 1995.

[2] Ellis, C.A., Gibbs, S.J. Concurrency control in groupware systems. *Proceedings of the ACM SIGMOD Conf. on Management of Data*, May 1989, 399-407

[3] Greenberg, S. and Marwood, D. Real time groupware as a distributed system: Concurrency control and its effect on the interface. *Proc. of the CSCW'94,* North Carolina, Oct. 1994, 207-218.

[4] Greenberg, S., Roseman, M., Webster, D. and Bohnet, R. Issues and experiences designing and implementing two group drawing tools. *Proc. of the 25th Annual Hawaii Intl. Conference on the System Science*, 1992, 138-150.

[5] Ignat, C.L. and Norrie, M.C. Customizable Collaborative Editor Relying on treeOPT Algorithm. *Proc. of ECSCW'03*, Helsinki, Finland, Sept. 2003, 315-334.

[6] Ignat, C.L. and Norrie, M.C. Grouping/Ungrouping in Graphical Collaborative Editing Systems. *The 5th Intl. Workshop on Collaborative Editing, ECSCW'03*, Helsinki, Finland, Sept. 2003.

[7] Ionescu, M. and Marsic, I. An Arbitration Scheme for Concurrency Control in Distributed Groupware. *The 2nd Intl. Workshop on Collaborative Editing Systems, CSCW'00,* Dec. 2000.

[8] Kanwati, R. LICRA: a replicated-data management algorithm for distributed synchronous groupware application. *Parallel Computing*, 22, 1992, 1733-1746.

[9] Karsenty, A., and Beaudouin-Lafon, M. An algorithm for distributed groupware applications. *Proc. of the 13th Intl. Conf. on Distributed Computing Systems*, May 1993, 195-202.

[10] Moran, T., McCall, K., van Melle, B., Pedersen, E. and Halasz, F. Some design principles for sharing in tivoli, a whiteboard meeting-support tool. *Groupware for Real-Time Drawings: A designer's Guide*, S. Greenberg, Ed. McGraw-Hill International(UK), 1995, 24-36.

[11] Newman-Wolfe, R.E., Webb M., and Montes, M. Implicit locking in the Ensemble concurrent object-oriented graphics editor. *Proc. of the CSCW'92*, New York, 1992, 265-272.

[12] Parker, D.S. et al. Detection of mutual inconsistency in distributed systems. *IEEE Trans. on Software Engineering*, vol. SE-9, no.3, 1983, 240-247

[13] Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S. and Suchman, L. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30, 1 (Jan. 1987), 32-47.

[14] Suleiman, M., Cart, M. and Ferrié, J. Concurrent Operations in a Distributed and Mobile Collaborative Environment. *Proeedings of ICDE'98*, Orlando, Feb. 1998, 36-45

[15] Sun, C. and Chen, D. Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems. *ACM Trans. on Computer-Human Interaction*, vol.9, no.1, March 2002, 1-41.

[16] Sun, C., Jia, X., Zhang, Y., Yang, Y. and Chen, D. Achieving Convergence, Causality-preservation, and Intention-preservation in Real-time Cooperative Editing Systems. *ACM. Trans. on Computer-Human Interaction*, vol. 5, no. 1, March 1998, 63-108.

[17] von Biel, V. Groupware Grows Up. *MacUser*, June 1991, 207-211.