Grouping/Ungrouping in Graphical Collaborative Editing Systems

Claudia-Lavinia Ignat
Institute for Information Systems
ETH Zurich
CH-8092 Switzerland
ignat@inf.ethz.ch

Moira C. Norrie
Institute for Information Systems
ETH Zurich
CH-8092 Switzerland
norrie@inf.ethz.ch

ABSTRACT
Often collaborative graphical systems lag behind in features with well accepted single-user applications. 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. In this paper we present an algorithm 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. The system relying on this algorithm is customizable, offering three policy modes for dealing with conflict: null-effect based policy, priority based policy and multi-versioning based policy.

Keywords
collaborative graphical editors, consistency maintenance, group/ungroup, serialization

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

From the perspective of users, a cooperative editor should be an extension of a single user editor with additional functionalities for supporting real-time collaboration. Often collaborative systems lag behind in features with some well accepted single-user applications. 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 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. One of the keys for offering the group/ungroup operations is the use of a hierarchical model of the document.

The algorithms underlying the existing collaborative graphical editing systems are based in isolation on some specific approaches for dealing with conflicts. By using non-optimistic locking [3] some systems prevent the conflicts from occurring. In case that users are freely allowed to edit any part of the document and some conflict occurs, three approaches have been considered. Some systems reject the intentions of all users involved in the conflict. Others maintain the intentions of one of the users arbitrarily chosen by the system. And, finally, some systems generate versions for the objects subject to conflict, in this way, respecting the intentions of all users. Since not all user groups have the same conventions and not all tasks have the same requirements, this implies that it should be possible to customize the collaborative editor at the level of both communities and individual tasks. 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, musical notations etc. 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. Our system offers three policy modes in the case that a set of concurrent operations are conflicting: the *null-effect based policy* where none of the operations in conflict are executed, the *priority based policy* when only the operation issued by the user with the highest priority wins the conflict and the *multi-versioning policy* when the effects of all operations are maintained. The last of these has still to be implemented.

We start our paper by giving an overview of the related work in the field of graphical editing systems. We then go

on by describing the algorithm presenting a classification of the types of conflicting operations and giving an example of the functionality of the algorithm. Some features of the graphical editor application relying on the proposed algorithm are presented in the following section. Concluding remarks and the main directions of our future work are presented in the last section.

RELATED WORK

The existing collaborative graphical editing systems are based on one of the following approaches: locking, serialization 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. Usually, these systems use a server to keep track of the status of the objects, if they are locked or unlocked, so that permission for locking can be denied or granted accordingly. This technique has a very poor responsiveness in internet applications where the network delay is significant. When an operation is generated it needs to wait to send the lock request message to the server and to receive the grant message. *Aspects* [16], *Ensemble* [11] and *GroupDraw* [4] are systems relying on the locking technique.

Serialization 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 Group design [9] are some prototypes implementing this technique.

The LICRA (Lock-free Interactive Concurrency Resolution Algorithm) approach relies on direct dependency relations between generated operations as well as on 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. 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 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 semantic, 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 ORESTE (Optimal RESponse TimE) algorithm underlying the GroupDesign system uses a history buffer with the executed operations and a queue with received operations that apply to nonexistent objects. 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.

A disadvantage of the serialization approaches described above is that, in the case of concurrent operations, the system randomly decides the effect of which operation to maintain.

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. GRACE [14] and TIVOLI [10] are two prototype systems that rely on multi-versioning.

GRACE (GRAphics Collaborative Editing) is an internet based prototype system that uses state vectors to define a total order among the operations and unique identifiers assigned to the objects. In the case that a remote operation is causally ready for execution, the collection of objects in the application scope of this operation is selected. Afterwards, the operation is checked against all the objects in its application scope and, in the case of conflict, it creates new object versions.

TIVOLI is a shared drawing system implementing the whiteboard metaphor and designed to support small group meetings, both in a single room and between remotely connected sites. All data in the system are independent and uniquely identifiable objects. Tivoli adopts an immutable object model, i.e. objects can only be deleted, they cannot be changed. To change an object (e.g. to move it), it must be deleted and a new object created that incorporates the change. If two operations concurrently target the same object, the object will be deleted and two new objects will be created, respecting the intentions of the operations. Each change operation is represented in the history list as an old-new object pair allowing the undoing of the changes, but consuming a lot of space for saving the editing operations. Consequently, the searching process becomes inefficient. In contrast to the GRACE approach where conflict occurs

at the object attribute level, in Tivoli the conflict occurs whenever two concurrent operations target the same object.

A disadvantage of the multi-versioning technique is that there is no correlation between the different versions of the same object and the base object. It is not clear what interface would be suitable for such systems to allow the user to navigate between the versions. And at the end of the editing process, it is not clear how the versions will be merged.

The dARB [7] algorithm uses a tree model for the document representation, 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.

THE ALGORITHM

Document model

A tree model was 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 assigned a unique identifier. The identifiers are uniquely generated at each site and are sent to the other sites together with the information related to the create operation.

Principles for consistency maintenance

Our algorithm follows the same principles for consistency maintenance as presented in [15]. Before presenting the consistency model underlying our algorithm, we will define the notions of operation, causal ordering relation, dependent and independent operations and conflicting operations.

The operation types offered by our graphical editor are the following:

- Create
- Delete
- Group
- Ungroup
- ChangeColour
- ChangeBckColour
- ChangePosition
- ChangeSize
- BringToFront
- SendToBack
- ChangeText

We define the generic operation as being a structure of the form:

Operation = <*type, initiatorID, stateVector, targetList, outputList, priority, nop, undoFlag*>, where

-*type* : is the type of the operation, *type*∈{*create*, *delete*, *group*, *ungroup*, *changeColour*, *changePosition*, *changeSize*, *bringToFront*, *sendToBack*, *changeText*, *changeBckColour*}

-*initiatorID* : the initiator site identifier

-*stateVector* : the state vector of the generating site [2]

-*targetList* : the list containing the identifiers of the target objects of the operation

-*outputList* : the list of identifiers of the objects created by the operation

-*priority* : the priority of the generating site, priority∈N*

-*nop* : boolean showing if the operation is valid or if it was cancelled

-*undoFlag* : used for performing local/global undo

In the case of the *create* operation, the *targetList* is empty. The *outputList* is not empty only in the case of *create* and *group* operations.

Some types of operations have additional attributes, as for example the colour in the case of *changeColour* and *changeBckColour*, the position in the case of *changePosition*, the size in the case of *changeSize* and the text in the case of *changeText*.

Causal ordering relation "→": 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 $O_x \rightarrow O_b$.

Dependent and independent operations: Given any two operations O_a and O_b, (1) O_b is *dependent* on O_a iff $O_a \rightarrow O_b$; (2) O_a and O_b are said to be independent or 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 two cases occur:

- O_1 and O_2 intend to modify the same attribute of a common target object to different values
- 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. Also note that if one of the operations is a *create* operation there

is no kind of conflict, since no other operation can concurrently target the object created by create.

We now go on by presenting the consistency model underlying the algorithm. The model comprises of 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.

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

To achieve causality preservation, we used a time-stamping scheme based on a data structure called state vector [2]. To achieve convergence, a serialization process is used. Intention preservation is problematic in the case of the conflicting operations, its meaning being bound to the policy chosen by the algorithm. In the case of the null-based policy, none of the intentions of the concurrent operations will be maintained. In the case of priority base operations, the combined effect of as many as possible operations is obtained. And finally, in the case of multi-versioning based policy, the intentions of all operations are preserved.

Concurrency policies

The collaborative graphical editing system we have implemented integrates the approaches that other existing systems have adopted for dealing with concurrency. Our system offers three policy modes: null-effect based policy, priority based policy and multi-versioning policy.

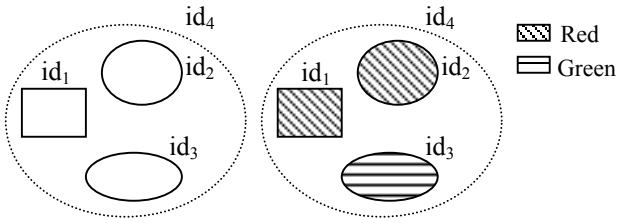
Null-effect based policy is characterized by the fact that, in the case of conflicting operations, the effect of all operations involved in the conflict as well as of the ones depending on the operations in conflict is cancelled.

Priority based policy means that, in the case of conflicting operations, the operation having the highest priority will be executed and the operations with lower priority will be cancelled. The effect of all other operations following these operations and depending on the cancelled operations is also cancelled. Users are assigned priorities according to their roles in the process of editing. The priority of a user is associated to a document at the moment the document is opened. So, a user can have higher priorities when editing one document and lower priorities when editing another. For instance, in the case of a collaborative architectural design, an architect is expert when working on certain projects but not an expert when working on some other projects. The assigned priority for a user related to a document will be the priority assigned to all operations generated by that user.

The *multi-versioning based policy* preserves the intentions of all operations in the case of a conflict. According to this technique, each time a conflict occurs, the target objects involved in the conflict will be versioned.

Conflicting operations

Group/ungroup operations increase the complexity of the algorithm for maintaining consistency in the case of collaborative graphical editors. For instance, suppose a scene of objects consists of a group containing a set of objects as shown in Figure 1 a). Suppose two users concurrently edit this scene of objects under the priority based policy.



a) before editing b) after editing

Fig. 1. Scene of objects illustrating the combined effect of two concurrent operations, one for changing the colour of a group and the other for changing the colour of an object from the group

The first user wants to change the colour of the group of objects with the identifier id_4 to red, while the second user concurrently wants to change the colour of the object having the identifier id_3 to green. There are some alternatives to deal with such a situation. One alternative is to consider that the two operations are in conflict because first user intends to change the colour of the group id_4, i.e. also of the object id_3, to red, while the second user intends to change the colour of the object id_3 to green. In this case only one of the two operations can be performed, the one having the highest priority. Another alternative would be to consider a 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 1 b). 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 the object belongs to. The combined effect is obtained by the serialization 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. Note that a similar approach was applied to the collaborative text editor we implemented [5,6]. For example, in the case that one operation deletes a text and another concurrent operation inserts another text in the middle of the concurrently deleted text, both intentions of the operations will be maintained: the text selected for the deletion will be deleted and the inserted text will be inserted. Note that our

algorithms maintain the syntactic consistency both in the case of text and graphical collaborative editors, do not dealing with the semantic consistency.

After highlighting the sort of complexities introduced by group/ungroup operations by using the above example, we can go on by describing the types of conflicts between the operations.

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

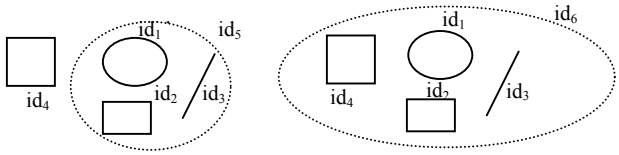
Real conflicting operations are those conflicting operations for which a combined effect of their intentions cannot be established. A serialization order of execution of these operations cannot be obtained: executing one operation will not make possible the execution of the other operation or will completely mask the execution of the other one. An example of real conflicting operations are the two concurrent operations *changeColour(id_1,Red)* and *changeColour(id_1,Blue)*, both targeting the same object and changing the colour of that object to different values. In the case of *no operation policy* for dealing with conflict, none of the concurrent real conflicting operations will be executed. In the case of the *priority based policy*, given a set of concurrent real conflicting operations, only the operation with the highest priority will be executed, the other operations being cancelled.

In Table 1, a list with the possible 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 HB.

Operation from HB	**Remote operation**	**Condition**
delete(id_1)	*ungroup*(id_2)	$id_1=id_2$
group($inList_1$, gid_1)	*group*($inList_2$,gid_2)	$inList_1 \cap inList_2 \neq \varnothing$
changeColour(id_1, colour)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2)$, $\forall$colour
changeColour(id_1, $colour_1$)	*changeColour*(id_2, $colour_2$)	$id_1=id_2$, $\forall colour_1 \neq colour_2$
changeBckColour(id_1,colour)	*Delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2)$, $\forall$colour
changeBckColour(id_1,$colour_1$)	*changeBckColour*(id_2, $colour_2$)	$id_1=id_2$, $\forall colour_1 \neq colour_2$
changePosition(id_1, x,y)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2)$, $\forall$x,y
changePosition(id_1, x_1,y_1)	*changePosition*(id_2,x_2,y_2)	$id_1=id_2$, $\forall(x_1,y_1) \neq (x_2,y_2)$
changeSize(id_1,Δx, Δy)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2)$, $\forall \Delta x, \Delta y$
changeSize(id_1,Δx_1, Δy_1)	*changeSize*(id_2,Δx_2,Δy_2)	$id_1=id_2$, $\forall(\Delta x_1,\Delta y_1) \neq (\Delta x_2,\Delta y_2)$
changeText(id_1, text)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2)$, $\forall$text
changeText(id_1, $text_1$)	*changeText*(id_2,$text_2$)	$id_1=id_2$, $\forall text_1 \neq text_2$
bringToFront(id_1, z)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2)$, $\forall$z
bringToFront(id_1, z_1)	*bringToFront*(id_2,z_2)	$id_1=id_2, \forall z_1 \neq z_2$
sendToBack(id_1, z)	*delete*(id_2)	$id_1=id_2 \vee id_1 \in subTree(id_2)$, $\forall$z
sendToBack (id_1, z_1)	*sendToBack*(id_2, z_2)	$id_1=id_2, \forall z_1 \neq z_2$

Table 1. Real conflicting operations

Resolvable conflicting operations are those conflicting operations for which a combined effect of their intentions can be obtained by serializing those operations. Consequently, ordering relations can be defined between any two resolvable conflicting operations. Any two resolvable conflicting operations can be defined as being in the right order, or in reverse order. Consider that two users concurrently edit the scene of objects containing a rectangle having the identifier id_4 and a group of three objects having the group identifier id_5, as shown in Figure 2. Suppose 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, the second user performs an ungroup operation upon the group with id_5.



a) before editing b) after editing

Figure 2. Scene of objects illustrating the 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 containing 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, i.e. *group({id_4,id_5},id_6)* before *ungroup(id_5)*. In the case that the ungroup operation is issued before the group operation, then the group operation will have an invalid argument, i.e. id_5.

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 represents the operations from the HB, but the order of serialization will be the reverse order specified in the fourth column.

Operation from HB	**Remote Operation**	**Condition**	**Order**
delete(id_1)	*group* ($inList_2$,id_2)	$id_1 \in inList_2$	Reverse order
group($inList_1$,id_1)	*delete*(id_2)	$id_2 \in inList_1$	Right order
ungroup(id_1)	*group*($inList_2$, id_2)	$id_1 \in inList2$	Reverse order
ungroup(id_1)	*changeColour*(id_2, colour)	$id_1=id_2$, $\forall$colour	Reverse order
ungroup(id_1)	*changeBckColour* (id_2, colour)	$id_1=id_2$, $\forall$colour	Reverse order
ungroup(id_1)	*changePosition*(id_2,x,y)	$id_1=id_2$,$\forall$x,y	Reverse order
ungroup(id_1)	*changeSize*(id_2, Δx,Δy)	$id_1=id_2$,$\forall \Delta$x,Δy	Reverse order
ungroup(id_1)	*sendToBack*(id_2, z)	$id_1=id_2$,$\forall$z	Reverse order
ungroup(id_1)	*bringToFront*(id_2, z)	$id_1=id_2$,$\forall$z	Reverse order
changeColour(id_1, $colour_1$)	*changeColour*(id_2, $colour_2$)	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall colour_1 \neq colour_2$	Reverse order
changeBckColour (id_1, $colour_1$)	*changeBckColour* (id_2, $colour_2$)	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall colour_1 \neq colour_2$	Reverse order
changePosition(id_1,x_1,y_1)	*changePosition*(id_2,x_2,y_2)	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall (x_1,y_1) \neq (x_2,y_2)$	Reverse order
changeSize(id_1, Δx_1,Δy_1)	*changeSize*(id_2, Δx_2,Δy_2)	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall (\Delta x_1,\Delta y_1) \neq (\Delta x_2,\Delta y_2)$	Reverse order
bringToFront(id_1, z_1)	*bringToFront*(id_2, z_2)	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall z_1 \neq z_2$	Reverse order
sendToBack(id_1, z_1)	sendToBack(id_2, z_2)	$id_1 \neq id_2 \wedge$ $id_1 \in subTree(id_2)$, $\forall z_1 \neq z_2$	Reverse order

Table 2. Resolvable conflicting operations

Description of the algorithm

After defining the notions of real conflicting and resolvable conflicting operations, we can go on to present the algorithm.

Each site involved in the editing process stores locally a copy of the shared document. As mentioned above, the document has a hierarchical structure. 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 serialization process will be applied. The serialization of the operations is somewhat more difficult than in the case of other algorithms using serialization, because of grouping/ungrouping operations.

Given the history buffer $HB=[O_1,\ldots,O_m,\ldots,O_n]$, 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.

The history buffer is traversed 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_List* containing the operations from $HB_{[m,n]}$ in real conflict with O_{new} is created. Also, the operations belonging to $HB_{[m,n]}$ being in a resolvable conflict relation with O_{new} are separated into two lists: *Right_Order_List* and *Reverse_Order_List*. If *Real_Conflict_List* is empty, then the operations belonging to *Right_Order_List* are redone, O_{new} is executed and afterwards the operations belonging to *Reverse_Order_List* are redone. Otherwise, the operation from *Real_Conflict_List* having the maximum priority is found and its priority is compared with the priority of O_{new}. If O_{new} has a lower priority, the history buffer should be restored, i.e. operations from $HB_{[m,n]}$ are redone and O_{new} is appended to the HB as a NOP operation. If O_{new} has a higher priority, the operations belonging to the *Real_Conflict_List* will be all set to NOP, the operations from *Right_Order_List* will be redone, O_{new} executed and afterwards the operations belonging to the

Reverse_Order_List redone. When dealing with the *Right_Order_List* and with the *Reverse_Order_List*, not only the operations being in resolvable conflict relation (right order and reverse order, respectively) need to considered, but also the operations directly dependent on the operations belonging to these lists.

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.

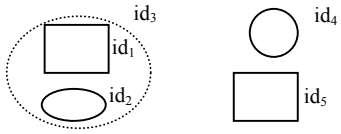


Figure 3. The initial scene of objects

Suppose three users with priorities 3, 2 and 1 respectively concurrently edit this graphical document. Further, suppose that the user at $Site_1$ wants to group the group with the identifier id_3 with the object having the identifier id_5. Concurrently, the user at $Site_2$ wants to ungroup the group id_3 and the user at $Site_3$ wants to perform a grouping of group id_3 with the object id_4.

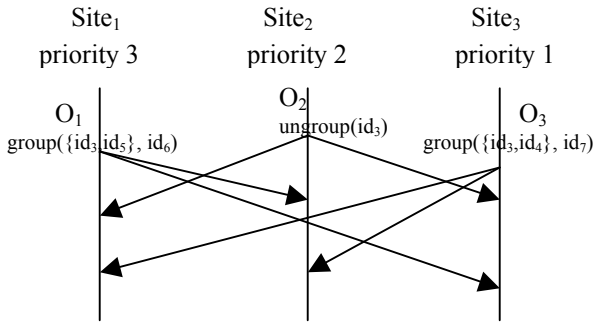


Figure 4. Scenario illustrating the functionality of the algorithm

Let us analyse how the algorithm applies to each of the three sites.

At $Site_1$, after O_1 is executed, HB=[O_1]. When the causally ready operation O_2 is received, the algorithm detects that O_1 is concurrent and in conflict with O_2, so O_1 is undone. The conflict between O_1 and O_2 is a resolvable conflict, the order of execution being O_1 followed by O_2. So, O_1 is redone and afterwards O_2 is executed. So, HB=[O_1, O_2]. When the causally ready operation O_3 is received, it is checked against O_1 and O_2, because both are concurrent operations: O_1 is in real conflict with O_3 and O_2 is in reverse order resolvable conflict with O_3. Because the priority of O_1 is higher than the priority of O_3, O_3 becomes NOP operation and will be added at the end of HB. So, finally HB=[O_1,O_2,O_3=NOP].

At $Site_2$, after the ungroup operation O_2 is performed, HB=[O_2]. When O_1 is received, O_2 is undone and O_1 will be executed followed by O_2 (O_2 and O_1 are in reverse order resolvable conflict). At this moment, HB=[O_1,O_2]. When O_3 is received, both O_2 and O_1 will be undone. O_1 is in real conflict with O_3 and O_2 and O_3 are in resolvable conflict relation (O_2 needs to be executed after O_3). Because the priority of O_1 is higher than the priority of O_3, O_3 will become a NOP operation, O_1 and O_2 will be redone and O_3 appended to the HB. So, finally HB=[O_1,O_2,O_3=NOP].

At $Site_3$, after O_3 is executed HB=[O_3]. When O_2 is received, because O_3 and O_2 are in right order resolvable conflict, HB=[O_3,O_2]. When O_1 is received, both O_2 and O_3 are undone, a real conflict is detected between O_3 and O_1 and a reverse order conflict is detected between O_2 and O_1. Because the priority of O_1 is higher, O_3 will become NOP operation. O_1 will be executed followed by O_2. So, finally HB=[O_3=NOP,O_1,O_2].

In the case of 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 these operations from HB will become NOP operations. All operations directly dependent on the NOP operations as well as the remote operation will also be cancelled.

THE GRAPHICAL EDITOR APPLICATION

We have implemented a multi-document collaborative graphical editor relying on the algorithm presented [1]. The communication part underlying our system is based on a generic model for the communication particularized for LAN and WEB. The LAN communication model is based on the multicast protocol, while the WEB communication model is using a client-server architecture.

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.

A garbage collection scheme has been implemented for removing those operations from the history buffer that are no longer used in the serialization process. The garbage collection scheme is similar to the one implemented in REDUCE [15]. Local and global undo/redo has been implemented, the undone operation being issued as a concurrent inverse operation. There is a challenge between the garbage collection and the undo: the garbage collected operations cannot be undone.

CONCLUSIONS AND FUTURE WORK

In this paper we presented some aspects of our on-going research in developing a collaborative graphical editor. We presented an algorithm in its current state of implementation for consistency maintenance in collaborative graphical editing. It deals not only with simple operations such as create/delete, change colour, change position, change size, but also complex operations of group/ungroup. The algorithm tries to preserve the

intentions of as many users as possible in the case of concurrent conflicting operations. The editor relying on this algorithm offers flexibility in choosing the policy for resolving conflicts.

Extending our graphical collaborative editor with the possibility of locking objects is one of the future functionalities to be integrated in our system. In some applications, the users would like to protect their 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.

Another issue is to introduce voice channel communication facilities in order to prevent or resolve the conflicts.

Our purpose is to develop a general algorithm combining all the three policies for concurrency. In the current prototype, only the priority based policy and the null-effect based policies are implemented. The multi-versioning technique is the third policy for concurrency maintenance to be integrated.

Extending the graphical editor with the facility to assign different priorities to different objects increases the flexibility and adaptability for different types of applications. In this way, the process of assigning priorities will be fine grained.

Our collaborative editor will be one of the applications to be integrated into the Universal Information Platform (UIP) [12] project which provides a complete and rich API that could be used for the development of collaborative space for general document management.

REFERENCES

1. Csaszar, L. Real-time collaborative editor, *Diploma Thesis*, ETH Zurich, 2003
2. Ellis, C.A. and Gibbs, S.J. Concurrency Control in groupware systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data,* May 1989, pp. 399-407
3. Greenberg, S. and Marwood, D. (1994): Real time groupware as a distributed system: Concurrency control and its effect on the interface, *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, North Carolina, October 1994, pp. 207-218.
4. Greenberg, S., Roseman, M., Webster, D. and Bohnet, R. (1992): Issues and experiences designing and implementing two group drawing tools, *Proceedings of the 25th Annual Hawaii International Conference on the System Science*, 1992, pp. 138-150.
5. Ignat, C. and Norrie, M.C. Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems, *Workshop on Collaborative Editing, Computer Supported Cooperative Work (CSCW 2002)*, New Orleans, Louisiana, November 2002
6. Ignat, C.L. and Norrie, M.C. Customizable Collaborative Editor Relying on treeOPT Algorithm, *Europen Computer Supported Cooperative Work (ECSCW 2003)*, Helsinki, Finland, September 2003 (to appear)
7. Ionescu, M. and Marsic, I. An Arbitration Scheme for Concurrency Control in Distributed Groupware, *The Second International Workshop on Collaborative Editing Systems, CSCW 2000*, December 2000.
8. Kanwati, R. LICRA: a replicated-data management algorithm for distributed synchronous groupware application, in *Parallel Computing* 22, 1992, pp. 1733-1746.
9. Karsenty, A., and Beaudouin-Lafon, M. An algorithm for distributed groupware applications, *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993, pp.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, in *Groupware for Real-Time Drawings: A designer's Guide*, S. Greenberg, Ed. McGraw-Hill International(UK), 1995, pp. 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 ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, New York, 1992, pp. 265-272.
12. Rivera, G. From File Pathnames to File Objects: An approach to extending File System Functionality integrating Object-Oriented Database Concepts, *Doctoral Thesis* No. 14377, ETH Zurich, September 2001.
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*, vol. 30, no.1, January 1987, pp.32-47.
14. Sun, C. and Chen, D. Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems, in *ACM Transactions on Computer-Human Interaction*, vol.9, no.1, March 2002, pp. 1-41.
15. Sun, C., Jia, X., Zhang, Y., Yang, Y. and Chen, D. Achieving Convergence, Causality-preservation, and Intention-preservation in Real-time Cooperative Editing Systems, in *ACM. Trans. on Computer-Human Interaction*, vol. 5, no. 1, March 1998, pp.63-108.
16. von Biel, V. Groupware Grows Up, in *MacUser*, June 1991, pp. 207-211.