Operation-based versus State-based Merging in Asynchronous Graphical Collaborative Editing

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

ABSTRACT
In this paper we present and compare two approaches for asynchronous communication in object-based collaborative graphical editing. The operations allowed to be performed on the shared graphical document are not only simple operations such as create, delete, move, change colour or position, but also group/ungroup operations. We present an operation-based merging approach that was also used in a real-time mode of collaboration, as well as a state-based merging approach and compare the two approaches. The private working space can be synchronised against a central repository or against another private workspace.

Keywords
collaborative graphical editors, consistency maintenance, asynchronous communication, operation-based/state-based merging

INTRODUCTION

Collaborative graphical editing systems support a group of people concurrently editing shared graphical documents. Object-based graphical editing systems are a particular class of collaborative graphical editing systems, where the shared objects subject to concurrent accesses are graphic objects such as lines, rectangles, circles and text boxes.
The collaboration can be synchronous or asynchronous. Synchronous collaboration means that members of the group work at the same time on the same documents and modifications are seen in real-time by the other members of the group. Asynchronous collaboration means that members of the group modify the copies of the documents in isolation, working in parallel and afterwards synchronizing their copies to reestablish a common view of the data.

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

In the *locking* approach [2] concurrent editing is allowed only if users are locking and editing different objects. Aspects [14], Ensemble [9] and GroupDraw [3] 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 [5] and GroupDesign [6] are examples of prototypes implementing this technique. LICRA relies on dependency relations between the operations and on the operational transformation mechanism, while GroupDesign uses an undo/redo mechanism for maintaining consistency. We also implemented a real-time graphical editor [4] that relies on operation serialisation and deals 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.

In the *multi-versioning* approach, for each concurrent operation targeting a common object, a new version of the object is created. GRACE [11] and TIVOLI [8] are two prototype systems that rely on multi-versioning.

Asynchronous collaborative systems have been developed for the case that the shared documents subject to collaboration are text [1,12,10], HTML or CRC cards [7]. The consistency maintenance mechanisms use either a state-based [1,12] or operation-based [10,7] merging. The state-based merging relies on the information about the states of the documents and no information about the evolution of one state into another is used. The operation-based merging approach keeps the information about the evolution of one state of the document into another in a buffer containing the operations performed between the two states of the document. The merging is done by executing the operations performed on a copy of the document onto the other copy of the document to be merged. To our knowledge, there has not yet been implemented any asynchronous collaborative graphical editor.

In this paper we propose a mechanism for maintaining the consistency in the case of the asynchronous object-based graphical editing. We propose two approaches, one relying on operation-based merging and the other on state-based merging. The operation-based approach is based on the same basic ideas that we used for the real-time communication [4].

We start our paper by describing the copy/modify/merge paradigm used in the asynchronous communication. We then present the model of the document and the set of operations that can be performed. We go on to present the operation-based approach and the state-based approach and make a comparison between the two mechanisms. We then compare our approach with some related work. Concluding remarks and the main directions of our future work are presented in the last section.

THE ASYNCHRONOUS COMMUNICATION

Most configuration management tools support the copy/modify/merge paradigm. It consists basically of three operations applied on a shared repository storing multiversioned objects: checkout, commit and update. A *checkout* operation creates a local working copy of an object from the repository. A *commit* operation creates in the repository a new version of the corresponding object by validating the modifications done on the local copy of the object. The condition of performing this operation is that the repository does not contain a more recent version of the object to be committed than the local copy of the object. An *update* operation performs the merging of the local copy of the object with the last version of that object stored in the repository.

In Figure 1 a scenario is illustrated in order to show the functionality of the Copy/Modify/Merge paradigm. *User*$_1$ and *User*$_2$ checkout the document from the repository and create local copies in their private workspaces (operations 1 and 2, respectively). *User*$_1$ modifies the document (operation 3) and afterwards commits the changes (operation 4). *User*$_2$ modifies in parallel with *User*$_1$ the local copy of the document (operation 5). Afterwards, *User*$_2$ attempts to commit their changes (operation 6). But, at this stage, *User*$_2$ is not up-to-date and therefore cannot commit their changes on the document. *User*$_2$ needs to synchronise their version with the last version, so he/she downloads the last version of the document from the repository (operation 7). A merge algorithm will be performed in order to merge the changes performed in parallel by *User*$_1$ and *User*$_2$ (operation 8). Afterwards, *User*$_2$ can commit their changes to the repository (operation 9).

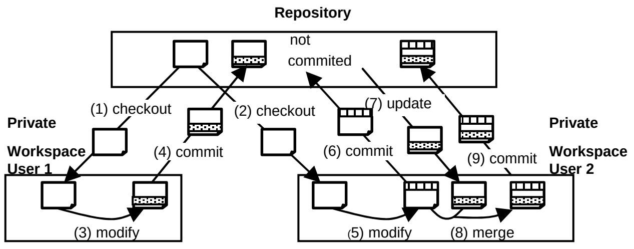

Figure 1. The Copy/Modify/Merge paradigm

DOCUMENT AND OPERATIONS REPRESENTATION

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 primitive objects. Each object has assigned a unique identifier. The primitive objects supported by our system are: lines, rectangles, ellipses and textboxes.

The graphical objects have associated the following attributes: *colour*- the colour of the foreground of the object; *bckColour*- the colour of the background of the object; *depth* – the depth at which the object is drawn, the objects with a higher depth being drawn first; *topLeftPoint* and *bottomRightPoint* – the left upper and bottom right corners of the minimal rectangle that covers the object, these attributes being needed in the scaling process. Additionally to these attributes, the line objects have *startPoint* and *endPoint* representing the starting and ending points that define the line. The *textBox* object contains the additional attribute *content* representing the content of the text box.

The operations performed on the graphical objects are the following ones: *create(P,S)* - adds the shape *S* to the document, as a child of *P*; *delete(P,S)* - removes the shape *S* from the document, *P* being the parent of *S*; *translate*(S,d_x,d_y) - translates shape *S* with (d_x,d_y); *scale*$(S, x_{ref},y_{ref},t_x,t_y)$ - scales shape *S* with the scaling factors t_x and t_y respectively, (x_{ref},y_{ref}) being the reference point; *ungroup*$(G,S_1,\ldots,S_N)$ - ungroups group *G* consisting of the components $S_1,\ldots,S_N$; *group*$(G,S_1,\ldots,S_N)$ - creates a new group *G* by grouping the shapes $S_1,\ldots,S_N$; *setColour(S,c)* - changes the colour of shape *S*, *c* being the new colour; *setBckColour(S,b)* - changes the background color of shape *S*, *b* being the new background colour; *setText(T,s)* - modifies the content of the textBox *T*, *s* being the new text; *setZ(S, z)* - changes the depth of shape *S*, *z* being the new depth of the shape.

In the case of the operation-based approach, the notion of the *inverse* of an operation as well as the notions of *follow* and *masking* relations among operations will be used for describing the algorithms underlying the asynchronous communication. Therefore, in what follows we are going to define these notions.

The *inverse* of an operation is defined as the list of operations that cancel the effects of the initial operation. Table 1 illustrates the inverse function for each type of operation.

O	**inverse**$_O$
create(P,S)	{*delete*(P,S)}
delete(P,S)	{*create*(P,S)}
translate(S,d_x,d_y)	{*translate*$(S,-d_x,-d_y)$}
scale$(S,x_{ref},y_{ref},t_x,t_y)$	{*scale*$(S,x_{ref},y_{ref},1/t_x,1/t_y)$}
ungroup$(G,S_1,\ldots,S_N)$	{*group*$(G,S_1,\ldots,S_N)$}
group$(G,S_1,\ldots,S_N)$	{*ungroup*$(G,S_1,\ldots,S_N)$}
setColour(S,c)	{*setColour*$(L,c_{oldL})\|L\in$ *leaves*(S)}
setBckColour(S,b)	{*setBckColour*$(L,c_{oldL})\|L\in$ *leaves*(S)}
setText(T,s)	{*setText*(T,s_{old})}
setZ(S,z)	{*setZ*(S,z_{old})}

Table 1. Inverse operations

The inverse function for *setColour(S,c)* is formed by the list of the *setColour* operations that reset the original colours for each leaf belonging to *S*. In the particular case of *S* being a primitive object, the result is the list formed by a single operation, i.e. the *setColour* operation that resets the original colour of that object. The inverse function for *setBckColour* is generated in a similar way as *setColour*.

The inverse function applied on a sequence of operations returns the list of operations that need to be executed right after the execution of the given sequence of operations in order to cancel all their effects. Therefore,

$inverse([O_1,...,O_N])=[inverse(O_N),...,inverse(O_1)]$

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 $setColour(id_4, red)$.

OPERATION-BASED APPROACH

In the operation-based approach, all the operations performed by the user are kept in a log at the client side. The repository keeps for each version the list of operations that transform the previous version into the current version.

The repository stores the following information: *version*, representing the version of the document stored in the repository; *delta[1..version]*, where *delta[i]* is a list of operations representing the difference between version $i+1$ and version i.

On the client side, the following data structures are stored: *version*, representing the version of the local document; *doc*, representing the local document; *lastSynchDoc*, representing the last version from the repository that was integrated into doc; *log*, consisting of the list of operations executed locally and not yet committed, representing the difference between *doc* and *lastSynchDoc*.

In what follows we are going to describe the basic operations of *checkout, commit* and *update* used for the asynchronous communication.

Checkout, Commit and Update

The *checkout* operation creates a local working copy of the requested version of the document from the repository. The repository sends all the operations needed to transform an empty document into the document associated with the requested version.

The *commit* operation creates a new version of the corresponding document in the repository by sending the log of the client to the repository. This operation is performed only if the local document is up-to-date, i.e. the repository does not contain a new version committed by other users.

The *update* operation performs the merge between the local copy of the document and the last version of the document from the repository.

In the following, the update procedure, both from the repository and client side, is presented.

```
procedure client.update{
      for i = version to repository.version{
            ops = repository.update(i);
            mops = merge(log, ops)
            doc = lastSynchDoc;
            doc.apply(mops);
            lastSynchDoc.apply(ops);
            log = compress( inverse(ops) + mops )}
}
function repository.update(vrs){
   return delta[vrs];
}
```

The list of operations received from the repository is merged with the list of operations from the local log, after solving the possible conflicts between operations, by using the *merge* procedure. The merge procedure integrates one by one the operations from the repository by using a serialisation mechanism, i.e. a reordering of the operations in order to achieve a combined effect of the intentions of the operations. The local document *doc* is modified to include the effects of the operations from the repository. *lastSynchDoc* must also be modified since a new version from the repository is integrated into the local document. The modifications on both *doc* and *lastSynchDoc* trigger the change of *log*, since *log* represents the difference between *doc* and *lastSynchDoc*. *log* is the list of operations that will be sent to the repository in the case of a commit operation, as representing the difference between the new version that will be committed to the repository and the last committed version. Therefore, *log* should include first the operations that cancel the effects of the operations from the repository, followed by the operations from *mops*. The *compress* procedure presented in the next subsection is applied for compressing the *log*.

Compression

The compression procedure transforms a list of operations into another list with a smaller number of operations that would have the same effects as the original list.

Various types of compression routines can be identified.

Operations with no effects can be removed. The operations belonging to this category are: $translate(S,0,0)$, $scale(S,x_{ref},y_{ref},1,1)$, $setColour(S,c)$ and $setBckColour(S,c)$ in the case that the colour of *S* is already *c*, $setZ(S,z)$ in the case that the depth of S is already z and *setText(T,s)* in the case that T already contains text s.

A pair of inverse operations can be removed. The pairs of inverse operations that can be removed are the following: *group*$(G,S_1,\ldots,S_N)$ and *ungroup*$(G,S_1,\ldots,S_N)$ with the condition that there are no operations in the local log referring to G between the given operations; *ungroup*$(G,S_1,\ldots,S_N)$ and *group*$(G,S_1,\ldots,S_N)$, the *group* operation being the inverse of the *ungroup* operation; *create(P,S)* and *delete(P,S)*; *delete(P,S)* and *create(P,S)*, the *create* operation being the result of the inversion of the *delete* operation; *translate*(S,d_x,d_y) and *translate*$(S,-d_x,-d_y)$; *scale*$(S,x_{ref},y_{ref},t_x,t_y)$ and *scale*$(S,x_{ref},y_{ref},1/t_x,1/t_y)$.

Operations masked by other operations can be removed.

Operations of the same type could be combined into one operation: *translate*(S,d_{x1},d_{y1}) and *translate*(S,d_{x2},d_{y2}) can be combined into *translate*$(S,d_{x1}+d_{x2},d_{y1}+d_{y2})$. *scale*$(S,x_{ref},y_{ref},t_{x1},t_{y1})$ and *scale*$(S,x_{ref},y_{ref},t_{x2},t_{y2})$ can be combined into *scale*$(S,x_{ref},y_{ref},t_{x1}\cdot t_{x2},t_{y1}\cdot t_{y2})$.

Direct User Synchronisation

According to the direct user synchronisation process, two clients can synchronise their workspaces without using the repository. The direct synchronisation process presumes that a direct connection between the clients can be established and that the users work on the same version.

The direct synchronisation process is one way. If client C_1 wants to synchronise with client C_2, it must initiate the process by sending a synchronisation request to C_2. If the request is accepted, C_1 receives the log of C_2 and merges it with its local log.

After the execution of the synchronisation routine, the operations from the other client will be stored in the local log as if they were locally generated. This issue raises the following problem. If the client C_1 receives an operation O from client C_2 as a result of the direct synchronisation process and then commits the changes, operation O would be stored in the repository. When C_2 is performing an update, operation O needs to be executed in the workspace of client C_2. But operation O had already been executed. Therefore, the re-execution of operation O would yield an incorrect result. The solution that we have adopted in order to resolve this problem is that each local workspace stores the identifiers of the executed operations and, before the execution of an operation, a test is performed to check whether that operation has already been executed.

Another problem arising in the direct synchronisation process is that the compression routine could combine an operation that has already been sent to a peer with other local operations. Let us assume that client C_1 synchronises with client C_2, receives operation O_1 and executes it. A new operation O_2 is generated by client C_2. The compression routine combines O_1 with O_2 resulting in O_3. When client C_2 commits the changes, the operation O_3 is copied in the repository. When client C_1 updates the version of the document from the local workspace, operation O_3 is received from the repository and executed. The result of the execution would yield an incorrect result since the operation O_3 contains the effect of O_1 which has already been executed on the copy of client C_1. In order to solve this problem, each operation stores the identifiers of the operations that have been combined to form the current operation, in a list called *COMBINED_OPS*. Previous to the execution of an operation the effect of all the operations included in the associated *COMBINED_OPS* list must be cancelled.

Merging

The merging routine takes as arguments two lists of operations and generates a new list by integrating the operations from the second list into the first list of operations and resolving the possible conflicts. The merging is done by reordering the operations in order to preserve the effects of all non-conflicting operations.

We start by defining the notion of conflict and giving a classification of the existent types of conflicts.

Two concurrent operations O_1 and O_2 are *conflicting* if one of the following 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 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 operation that either destroys an object from the group, intends to modify an object from the group or uses it in a grouping operation.

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. One of the following cases occurs:

- executing any of the two operations will not make possible the execution of the other one
- executing any of the two operations will completely mask the effect of the other one
- executing one of the operations will not make possible the execution of the other operation and executing the

other operation will make completely invisible the effect of the previous operation

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 Table 2, a list of the real conflicting operations is given. The first column contains the local operations, the second column the remote operations and the third column the condition that a real conflict occurs between the local and remote operations. The same real conflicting situations occur if the first column of the table represents the remote operations and the second column the local operations.

Local operation	**Remote operation**	**Condition**
delete(P,S)	*ungroup*(G,S_1,…,S_N)	S=G
group(G_1,S_1,…,S_N)	*group*(G_1,R_1,…,R_M)	$\{S_1,\ldots,S_N\}\cap\{R_1,\ldots,R_M\}\neq\emptyset$
setColor(S_1,c)	*delete*(P,S_2)	$S_1=S_2 \vee S_1\in S_2$
setColor(S,c_1)	*setColor*(S,c_2)	$c_1\neq c_2$
setBckColor(S_1,c)	*delete*(P,S_2)	$S_1=S_2 \vee S_1\in S_2$
setBckColor(S,c_1)	*setBckColour*(S,c_2)	$c_1\neq c_2$
translate(S_1,d_x,d_y)	*delete*(P,S_2)	$S_1=S_2 \vee S_1\in S_2$
translate(S,d_{x1},d_{y1})	*translate*(S,d_{x2},d_{y2})	$(d_{x1},d_{y1})\neq(d_{x2},d_{y2})$
scale(S_1,x_{ref},y_{ref},t_x,t_y)	*delete*(P,S_2)	$S_1=S_2 \vee S_1\in S_2$
*scale(*S,x_{ref1},y_{ref1},t_{x1},t_{y1})	*scale(*S,x_{ref2},y_{ref2},t_{x2},t_{y2})	$(x_{ref1},y_{ref1},d_{x1},d_{y1})\neq(x_{ref2},y_{ref2},d_{x2},d_{y2})$
setText(S_1,text)	*delete*(P,S_2)	$S_1=S_2 \vee S_1\in S_2$
setText(S,$text_1$)	*setText*(S,$text_2$)	$text_1\neq text_2$
setZ(S_1,z)	*delete*(P,S_2)	$S_1=S_2 \vee S_1\in S_2$
setZ(S,z_1)	*setZ*(S,z_2)	$z_1\neq z_2$

Table 2. Real conflicting operations

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. In the case of two *setColour* operations: *setColour(group$_1$,red)* and *setColour(id$_1$,green)*, where the object with *id$_1$* belongs to the group identified by *group$_1$*, a combined effect could be obtained by executing first the *setColour(group$_1$,red)* followed by *setColour(id$_1$,green)*. The result would be that the object identified by *id$_1$* will have *green* color and the other objects belonging to the group *group$_1$* except object *id$_1$* will have *red* colour.

In Table 3, a list with resolvable conflicting operations is given. The first column contains the local operations, the second column the remote operations and the third column the condition that a resolvable conflict occurs between the local and the remote operations. 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 local operation and reverse order indicates that the remote operation should be executed before the local operation. 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 local operations, but the order of serialisation will be the reverse order specified in the fourth column.

Local Operation	**Remote Operation**	**Condition**	**Order**
delete(P,S)	*group*(G,S_1,…,S_N)	$S\in\{S_1,\ldots,S_N\}$	Reverse
delete(P_1,S_1)	*delete*(P_2,S_2)	$S_2\in S_1$	Reverse
ungroup(G,S_1,…,S_N)	*setColor*(G,c)		Reverse
ungroup(G,S_1,…,S_N)	*setBckColor*(G,c)		Reverse
ungroup(G,S_1,…,S_N)	*translate*(G,d_x,d_y)		Reverse
ungroup(G,S_1,…,S_N)	*scale*(G,x_{ref},y_{ref},t_x,t_y)		Reverse
ungroup(G,S_1,…,S_N)	*setZ*(G,z)		Reverse
setColor(S,c_1)	*setColor*(G,c_2)	$S\in G$, $c_1\neq c_2$	Reverse
setBckColor(S, c_1)	*setBckColor*(G, c_2)	$S\in G$, $c_1\neq c_2$	Reverse
translate(S,d_{x1},d_{y1})	*translate*(G,d_{x2},d_{y2})	$S\in G$, $(d_{x1},d_{y1})\neq(d_{x2},d_{y2})$	Reverse
scale(S,x_{ref1},y_{ref1},t_{x1},t_{y1})	*scale*(G,x_{ref2},y_{ref2},t_{x2},t_{y2})	$S\in G$, $(x_{ref1},y_{ref1},d_{x1},d_{y1})\neq(x_{ref2},y_{ref2},d_{x2},d_{y2})$	Reverse
setZ(S,z$_1$)	*setZ(G,z$_2$)*	$S\in G$, $z_1\neq z_2$	Reverse

Table 3. Resolvable conflicting operations

The merging procedure consists of the integration of each operation from the repository into the local log. The integration of the remote operation into the local log is done using the same algorithm that has been used for real-time collaboration [4]. Given the local history buffer *HB=[O$_1$,…,O$_m$,…,O$_n$]*, the steps of the integration of the operation *O$_{new}$* from the repository into *HB* are presented in what follows.

Firstly, *O$_{new}$* is checked for whether it depends on any cancelled operation from the repository. If it is the case, *O$_{new}$* is cancelled too and it is not inserted into HB. Otherwise, *O$_{new}$* needs to be integrated into *HB* and the operations from *HB* need to be reordered in order to preserve the intentions of as many as possible operations among *O$_{new}$* and the operations from *HB*. A list *Real_Conflict* containing the operations from *HB* in real conflict with *O$_{new}$* is created. A list called *Right_Order* is created to contain all the operations belonging to *HB* that have to be executed before *O$_{new}$*, i.e. those operations that are in a right order resolvable conflict relation with *O$_{new}$*. The list of operations *Reverse_Order* is created to contain all the operations belonging to *HB* that have to be executed after *O$_{new}$*, i.e. the operations from *HB* that are in a reverse order relation with *O$_{new}$*, the operations that are in a follow relationship with 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 merging mechanism. We have implemented three main policies for merging: to always execute the local operations, to always execute the operations from the repository or to let the user manually choose which of the conflicting operations is to be executed.

The reordering of the operations from *HB* should be done in such a way that the operations from *Right_Order* are positioned before O_{new} and the operations from *Reverse_Order* are positioned after O_{new}.

If *Real_Conflict* is empty, O_{new} is added at the end of *HB* and afterwards the operations from the *Reverse_Order* are moved to the end of *HB* respecting their initial order in *HB*.

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 nothing has to be done.

b) If the winning operation is the remote operation, all operations from the list *Real_Conflict* as well as the operations that depend on them are removed from *HB*, O_{new} is added at the end of *HB* and afterwards the operations from the *Reverse_Order* are moved at the end of *HB* respecting their initial order in *HB*.

The difference between the algorithm for the asynchronous communication and the one for the real-time communication [4] is that, in the case of the asynchronous communication, the masked operations are not considered since they are eliminated by the compression procedure. The reader is referred to [4] for a discussion on the correctness of the algorithm.

STATE-BASED APPROACH

In the state based approach the *checkout* routine consists of the sending by the repository of the document corresponding to the requested version. In the *commit* routine, the client sends the local document to the repository. In the *update* routine, the last version of the document from the repository is merged with the local document. In what follows we are going to describe the merging procedure.

The *merging* routine has, as arguments, two graphical documents - a copy of the local document D_{local} and the remote document from the repository D_{remote} and returns a new document resulting from the process of combining the two given documents. The merging process is done relative to the *lastSynchDoc* value, i.e. the last version from the repository that was integrated into the local document. Two different merging algorithms are used: one for the attributes of the objects and the second for the tree structures. In the following both algorithms are presented.

The *merging of the object attributes* is done only for the case that the leaves are present in all three documents *lastSynchDoc*, D_{local} and D_{remote}. Otherwise, a change in the tree structure is detected and the case is handled by the algorithm for merging the structures of the trees. In the case that the structures of the trees are the same, if a certain attribute from a leaf S was modified only in one document D_{local} or D_{remote}, the modification is simply kept in the resulting document. The changing of an attribute in both documents is considered a conflict. In this case only one modification is kept depending on the chosen policy.

The *tree merging algorithm* handles the changes related to the tree structures. The graphical documents D_{local}, D_{remote} and *lastSynchDoc* are transformed into directed acyclic graphs (DAG), the graphical objects becoming vertices, the edges being oriented from parents to children.

Three graphs, $G_L=(V_L,E_L)$, $G_R=(V_R,E_R)$ and $G_{LS}=(V_{LS}, E_{LS})$ are obtained as result of the transformation of D_{local}, D_{remote} and *lastSynchDoc* documents, respectively.

For handling the conflicts that occur during the merging process, a priority function is defined:

$$\text{priority}(doc_1, doc_2) = \begin{cases} 0, \text{if the changes in } doc_2 \text{ have priority} \\ 1, \text{if the changes in } doc_1 \text{ have priority} \end{cases}$$

A cost is associated with each edge from G_L and G_R:

$$\text{cost}(u) = \begin{cases} 0, \text{if } u \in E_{LS} \\ 1 + \text{priority}(D_{local}, D_{remote}), \text{if } u \notin E_{LS} \end{cases} \text{where } u \in E_L$$

$$\text{cost}(u) = \begin{cases} 0, \text{if } u \in E_{LS} \\ 1 + \text{priority}(D_{remote}, D_{local}), \text{if } u \notin E_{LS} \end{cases} \text{where } u \in E_R$$

In what follows we describe the algorithm for merging.

Firstly, a new graph $G=(V_L \cup V_R, E_L \cup E_R)$ is constructed containing all the vertices and edges from both G_L and G_R. In order to convert G to a graphical document, some edges must be removed. A DAG is a tree if the in-degree of all vertices is 1, i.e. every vertex has only one parent.

The vertices of the graph are connected either by one edge or by two edges with their parent. The vertices connected by one edge with their parent indicate the removal or the addition of a graphical object from/to a document. If the cost of the edge that connects the vertex with its parent is 0, the edge is removed from the graph. Otherwise, no modification is done. For the vertices connected by two edges with their parent, the edge with lower cost that connects the given vertex with its parent is removed from the graph. In the end, all the vertices that are not in the same connected component with the root vertex R are removed from the graph.

Consider a scene of objects consisting of two groups, G_1 and G_2. Consider that group G_1 contains two objects, L_1 and L_2 and group G_2 is formed by group G_3 and object L_5. Group G_3 is composed by the objects L_3 and L_4. Further, suppose that two users, *User*$_1$ and *User*$_2$, checkout the version of the document representing this scene of objects and concurrently modify it. For resolving the conflicts we consider that the local changes have priority over the remote changes. *User*$_1$ adds an object L_6 to the scene of objects and commits the version of the document into the repository. *User*$_2$ groups the groups G_1 and G_2 into a new group G_4, adds the object L_7 to the scene of objects and deletes object L_2. *User*$_2$ tries to commit the changes into the repository, but needs to first update the local workspace with the changes from the repository. Figures 2a), 2b) and 2c) illustrate the graphs G_{LS}, G_R and G_L respectively.

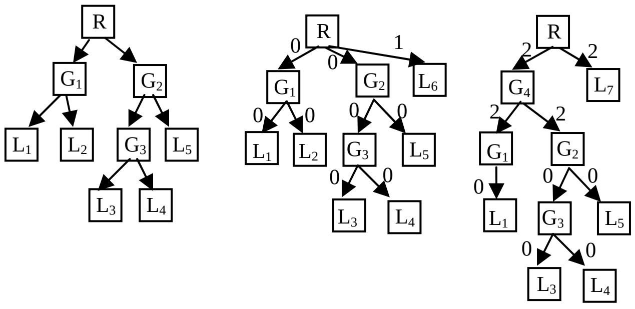

Figure 2a) G_{LS} 2b) G_R 2c) G_L

The merging routine consists of the merging of the graphs G_R and G_L. The intermediate graph resulting from the union of the vertices and the edges of the graphs G_R and G_L is shown in Figure 3a). The intermediate graph is then transformed into the final graph shown in Figure 3b).

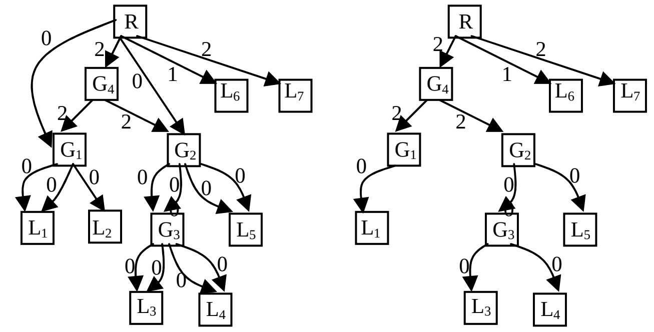

Figure 3a) Intermediate graph 3b) Final graph

OPERATION-BASED VERSUS STATE-BASED APPROACHES IN GRAPHICAL EDITING

The state-based merging algorithm compares the states of the documents to be merged in order to generate the result of the merging, while the operation-based merging algorithm uses the differences between the two documents and a reference document.

The operation-based approach is more efficient than the state-based approach in the case of large documents because the number of operations that transform version *i* into version *i+1* would be statistically smaller than the number of graphical objects.

The state-based approach is more efficient than the operation-based approach for merging in the case of small documents where the number of operations is comparable with the number of objects from the graphical document.

The operation-based approach has some advantages over the state-based approach for merging such as a better resolution for conflicts by partially preserving the intentions of the operations in conflict.

Consider a scene of objects consisting of a group *G* that contains a rectangle *R* and an ellipse *E*, all having the colour *white*. Consider that two users concurrently modify this scene of objects. Suppose that *User*$_1$ changes the colour of *G* into *blue* and performs a commit. Concurrently, *User*$_2$ changes the colour of *R* into *green* and tries to commit his changes. *User*$_2$ needs to first update the version of his document. In the case of state-based merging, a conflict is detected between the two operations performed by the two users, because the colour of *R* is modified both locally and remotely. In the case of operation-based merging, the two operations are detected as resolvable conflicting operations and they will be serialised, first the operation changing the colour of the group *G* is performed then the operation of changing the colour of the rectangle *R*. In the case of the operation-based approach, the conflicts are detected only between real conflicting operations and a partial preservation of the intentions of the users is realised by a serialisation of the resolvable conflicting operations. On the other side, in the case of the state-based approach, the conflicts are generated whenever a property of an object has different values in the two documents to be merged. Moreover, in the case of operation-based merging, rules for the definition and resolution of conflicts can be defined. The real and resolvable conflicts can be defined between pairs of operations. In this paper we defined the pairs of operations that are in a real or resolvable conflict, but the pairs of conflicting operations can be defined depending on the application. In the case of state-based merging, there is only one way for the definition and resolution of conflicts, i.e. when a property of an object has different values in the two documents to be merged and the policies for dealing with conflict are either to keep the local or remote changes or to let the user choose manually the modification to be kept.

RELATED WORK

To our knowledge, there are no other collaborative graphical asynchronous editing systems. Therefore, in this section, we are going to relate our operation-based approach with other approaches for maintaining the consistency in real-time graphical editing based on serialisation. Also, we are going to relate our state-based merging with other state-based approaches used in some asynchronous editing systems.

The main difference between our approach and other approaches based on serialization such as GroupDesign[6] and LICRA[5] 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 in the way we propose here using the mechanisms from GroupDesign and LICRA.

The same basic algorithm proposed in this paper has been used for the real-time graphical editing [4].

Some of the version control systems such as CVS [1] and RCS [14] adopt state-based merging for the concurrent editing of text documents. The basic unit for conflict definition and resolution is the line, meaning that the changes performed by two users are in conflict if they refer to the same line. In our state-based approach, the basic unit for the conflict definition and resolution is the property of an object in the scene of objects.

In [13] a state-based merging algorithm for XML documents is proposed. As in our approach, graphs are associated to tree representations of the two documents to be merged. A merged graph is constructed by considering the union of the nodes and edges of the two graphs. The merged graph is then analysed and the result generated. The proposed approach [13] also determines the difference to be stored in the repository. In our approach we keep whole documents in the repository, but we plan to investigate storing only the differences between the documents. However the approach of merging XML documents is more complex than the approach of merging object-based graphical documents, because the children of an XML node are ordered, while there is no order among the objects belonging to a group.

CONCLUSIONS AND FUTURE WORK

In this paper we proposed an approach for maintaining consistency in the case of asynchronous object-based graphical editing, which, to our knowledge, is the first work in this direction. We proposed one merging algorithm based on operations and another based on the states of the documents and compared the two approaches. The operation-based approach that we proposed is the same approach that we have used for real-time communication. The synchronisation of the private working space can be performed against the repository or against another private working space.

We are currently extending the system and adapting the consistency maintenance algorithm to deal with other objects such as polygonal lines, free forms, connecting lines, as well as annotations, such that the system can be used in the architectural or product data management design.

REFERENCES

1. Berliner, B. CVS II:Parallelizing software development. *Proc. of USENIX*, Washington D.C., 1990.
2. 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, October 1994, pp. 207-218.
3. 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 International Conference on the System Science*, 1992, pp. 138-150.
4. Ignat, C. and Norrie, M.C. Grouping in Collaborative Graphical Editors. *Proc. of the CSCW'04*, Chicago, November 2004, to appear
5. Kanwati, R. LICRA: a replicated-data management algorithm for distributed synchronous groupware application. *Parallel Computing* 22, 1992.
6. 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.
7. Molli, P., Skaf-Molli, H., Oster, G. and Jourdain, S. Sams. Synchronous, asynchronous, multi-synchronous environments. *Proc. of the 7th Int. Conf. on CSCW in Design*, Rio de Janeiro, Brazil, Sept. 2002
8. 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, pp. 24-36.
9. 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.
10. Shen, H., Sun, C.: Flexible merging for asynchronous collaborative systems. *Proc. of CoopIS/DOA/ODBASE 2002*, pp. 304-321.
11. 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.
12. Tichy, W.F. RCS-A system for version control. *Software-Practice and Experience*, 15(7),1985.
13. Torii, O., Kimura, T., Segawa, J.: The consistency control system of XML documents. *Symposium on Applications and the Internet*, Jan. 2003.
14. von Biel, V. Groupware Grows Up, in *MacUser*, June 1991, pp. 207-21