Draw-Together: Graphical Editor for Collaborative Drawing

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

ABSTRACT
Collaborative object-based graphical editors offer good support for design teams to work concurrently on their design. However, not much research has been done on maintaining consistency when complex operations such as the grouping of objects or working on layers are involved. In this paper, we propose a novel operation serialisation algorithm for consistency maintenance based on the reordering of nodes in a graph. The nodes of a graph represent operations and the edges represent ordering constraints between operations. Users can specify types of conflicts between operations and the policy for the resolution of conflicts.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *Distributed applications*; D.2.2 [**Software Engineering**]: Design 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
Graphical collaborative editing, consistency maintenance, serialisation of operations, topological sort

1. INTRODUCTION
Drawing is a primary activity in many design domains, such as architectural and product design, and computer tools should help design teams manage and work concurrently on design drawings.

Collaborative graphical editing systems support a group of people concurrently editing graphical documents over a computer network. In the case of object-based graphical editing, the shared information space central to the collaboration is a scene of objects. Previous approaches in the collaborative graphical editing field can be classified into locking, serialisation and multi-versioning.

In the *locking* approach adopted by systems such as Aspects [17], Ensemble [10] and GroupDraw [2], concurrency is restricted as concurrent editing is allowed only if users are locking and editing different objects. Moreover, responsiveness is affected due to delays for lock acquisition.

CSCW'06, November 4–8, 2006, Banff, Alberta, Canada.

Serialisation as implemented by LICRA [5] and GroupDesign [6] 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. In [3] an operation serialisation mechanism has been proposed based on the definition of conflicts between the operations and of an order of execution of conflicting operations such that a combined intention of users is obtained.

The *multi-versioning* approach tries to achieve all operation effects, preserving the intentions of all operations. For each concurrent operation targeting a common object as in TIVOLI [7] or a common property of the object as in GRACE [15], a new version of the object is created. However, the multi-versioning approach raises some issues related to the graphical user interface, such as how the versions of an object are related to the base object or the way the navigation through the versions of an object is realised. CoGroup[18] is a multi-versioning approach that adopts the multi-version single display, meaning that only one version is displayed in the user interface according to user assigned priorities.

With the exception of [3] and [18], none of the other approaches have implemented the operations of grouping and ungrouping which are fundamental operations required in the editing process of graphical documents. Moreover, none of the existing approaches discussed the issues concerning concurrency in multi-page documents or documents involving layers.

In this paper, we extend the approach proposed in [3] by providing a novel solution for consistency maintenance that uses an algorithm for the serialisation of operations based on the reordering of nodes in a graph. The nodes of the graph represent executed operations and the edges of the graph represent ordering constraints between these operations. We have classified conflicts into real and resolvable, depending on whether an ordering of execution between pairs of operations can be established or not. We allow an additional set of operations to those proposed in previous systems. The set of operations that we have implemented satisfy the requirements of architectural and product design. For example, in addition to complex operations of group/ungroup, we support concurrent operations targeting different pages and layers of objects. In contrast to [3] and [18], we allow users to define the types of conflicts between the operations and the policy for the resolution of conflicts. In this way, conflict handling can be customised to suit the requirements of specific applications.

We structure the paper as follows. In section 2 we present the model that we adopted for the representation of graphical documents and the set of operations that can be performed on the

scene of objects. We classify the types of conflicts between operations and explain the problem of maintaining the combined effect of the intentions of users. In section 3 we describe our serialisation-based mechanism for maintaining consistency. Comparison of our approach with related work is presented in section 4. Concluding remarks and some future work directions are provided in section 5.

2. MODEL AND PROBLEMATIC

The scene of objects can be modeled by a hierarchical structure: groups are represented as internal nodes, while simple objects are represented as leaves. A group can contain other groups or simple objects.

A node N of a document is a structure of the form $N=<parent, children>$, where

- *parent* is the parent node for the current node. Except for the topmost node, parent is a valid reference to a node in the tree.
- *children* is an unordered list $[child_1,...,child_n]$ of child nodes

The children of an internal node, i.e. a group of objects, are the objects contained in the group. The order between the child nodes of a group does not matter. A node is identified by its identifier and not by its position in the parent structure. A leaf node does not have any children, and it can be any type of simple object.

The simple objects supported by our system are the following: rectangles, circles, ellipses, lines, text boxes, polylines (open/closed), freehand polylines and bitmaps. The freehand polylines are defined by a set of points connected by lines. The large number of points composing the polyline gives the impression of a freehand shape.

The document contains multiple pages and each page contains a set of layers. Each layer has a root object that references the scene of objects containing groups and simple objects. The layers may be set to be visible or not, which determines whether or not the objects that belong to them appear in the displayed scene of objects. The structure of the document is illustrated in Figure 1.

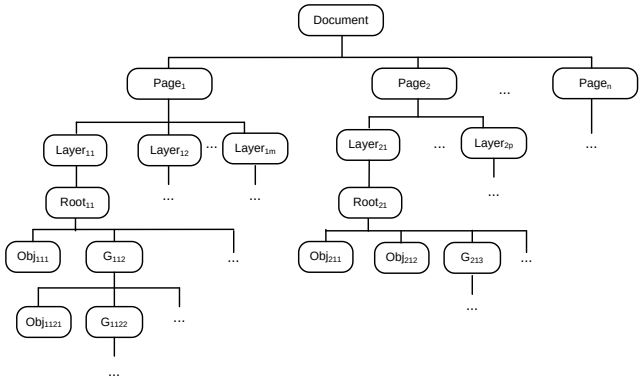

Figure 1. Structure of the graphical document

The operations that can be performed on the scene of objects are as follows:

- *create(Obj)* to create an object *Obj*
- *delete(Obj)* to delete an object or group of objects *Obj*
- *group([Obj1,Obj2,…,Objn],G)* to group the objects $Obj_1,Obj_2,\ldots,Obj_n$ into the group *G*. Obj_1, Obj_2,…, Obj_n can be objects or groups of objects.
- *ungroup(G)* to ungroup the group *G*
- *move(Obj,dx,dy)* to move object *Obj* to a relative position given by the distances d_x and d_y on the *x* and *y* axes respectively
- *scale(Obj,rx,ry)* to scale the object or group *Obj* with the ratio r_x and r_y on the *x* and *y* axes respectively
- *rotate(Obj,angle)* to rotate the object or group *Obj* in counterclockwise direction with angle *angle*
- *chgColour(Obj,colour)* to change the fill colour of the object or group *Obj* to colour *colour*
- *chgLineColour(Obj,colour)* to change the line colour of the object or group *Obj* to colour *colour*
- *chgStroke(Obj,width)* to set the width of the line of the object or group *Obj* with the value *width*
- *chgText(Obj,text)* to change the text contained in the textbox *Obj* to *text*
- *chgTextSize(Obj,size)* to change the font size of the text contained in the textbox to *size*
- *sendToBack(Obj)* and *sendToFront(Obj)* operations that move the object or group *Obj* to the back or to the front of the scene of objects. These operations use an auxiliary function *setZPosition(Obj,zPos)* to set the depth of the object or group *Obj* to the value *zPos*.
- *movePoint(Obj,index,dx,dy)* to move the point given by the index *index* belonging to the polyline *Obj* to a relative position given by the distances d_x and d_y on the *x* and *y* axes respectively.
- *createPage(P,name,nextPage)* to create a page identified by *P* with the name *name* before the page identified by *nextPage*.
- *removePage(P)* to remove the page identified by *P*.
- *createLayer(L,P,name)* to create a layer identified by *L* in the page *P* with the name *name*.
- *removeLayer(L)* to remove the layer identified by *L*
- *moveToLayer(Obj,L)* to move the object or group identified by *Obj* to the layer *L*.
- *createAnnotation(A,Obj)* creates an annotation *A* that is attached to the object *Obj*.

Each operation has an associated state vector [1] and an identifier of the site which generated the operation.

A replicated architecture has been used where each user works on a copy of the document. Local operations are executed on the local copy of the document immediately after their generation and then transmitted to the other sites. When a remote operation arrives at a site, some of the operations that have been performed at that site might be undone and re-executed together with the remote operation in order to satisfy a combined effect of the concurrent operations. Two operations generated at different sites are said to be concurrent if, at the moment of generation of one of the operations, the other operation was not executed at that site.

As specified in [3], we consider that two concurrent operations are *conflicting* if they modify the same property of a common target object to different values or one operation targets an object that is destroyed by deletion or ungrouping by the other operation.

In order to illustrate the types of conflict that we defined, let us consider the following scenario where two users concurrently edit a scene of objects. The first user groups a group *G* with another object *O* and the second user ungroups the group *G*. A solution to this scenario is to consider that the two operations are in conflict as they target the same group and that only one of them can be executed. If this solution is desired to be obtained, the two operations have to be defined as being in a *real conflict* relation. Another solution is to obtain a combined effect of the two operations such as the grouping of individual objects in *G* with the object *O*. This solution can be obtained by specifying that the two operations are in a *resolvable conflict* and that the group operation should be executed first followed by the ungroup operation. In the case that the ungroup operation would be executed first, the group operation would target the group *G* that does not belong to the structure of the document. Each application can specify according to its needs the types of conflict between operations. In what follows we define the real and resolvable conflicts between operations used in our approach.

Real conflicting operations are those conflicting operations for which a combined effect of their intentions is not desired or cannot be established. As we have seen, the scenario presented above gives an example of two concurrent operations that can be defined as real conflicting. The class of real conflicting operations includes those operations for which a serialisation order of execution of these operations cannot be obtained to preserve the intentions of the operations: executing one operation will make the execution of the other operation impossible or will completely mask the execution of the other one. An example of this kind of real conflicting operations is the two concurrent operations *chgColour(*id_1*,red)* and *chgColour(*id_1*,blue)*, both targeting the same object and changing the colour of that object to different values. The operation that wins the conflict is decided according to a priority scheme, in our case according to priorities assigned to users, with the operation generated by the user with the highest priority being the one that wins the conflict.

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 the two concurrent operations. Any two resolvable conflicting operations can be defined as being in the right order or in the reverse order. Note that conflicting operations that can be classified as resolvable conflicting operations may be defined as being real conflicting operations by certain applications.

For maintaining consistency between the copies of the document we adopted the operation serialisation mechanism based on the reordering of the operations from the history buffer. In the process of reordering operations, the precedence relation between the operations has to be maintained. An operation O_x *precedes* O_y if O_y was generated after O_x was executed.

An operation O_2 from the history buffer is said to *depend on* O_1 (O_2 *depends on* O_1) if O_1 creates an object or group that belongs to the target list of O_2. An operation cannot be executed before the operation on which it depends. Moreover, an operation has to be cancelled if the operation it depends on is cancelled. If two operations are in a *depends on* relation, they are also in a *precedes* relation.

3. SERIALISATION-BASED MECHANISM FOR MAINTAINING CONSISTENCY

In this section we present the operation serialisation mechanism that we adopted for maintaining consistency. We first give an intuitive explanation of the issues that occur in the reordering of operations in the presence of real and resolvable conflicts. We then present the algorithms for the integration of an operation into the history buffer containing the previous executed operations at that site. We describe how conflicts are defined in our system, and provide some information about our application.

3.1 An Intuitive Explanation of the Algorithm

Operation serialisation is the mechanism by which operations in the history buffer *HB* are re-executed in an order such that the partial combined effect of the intentions of users is achieved. The serialisation order takes into account the ordering constraints between the operations. The conflicts as well as the *precedes* relations between the operations have to be considered. In the case of two real conflicting operations, depending on the policy for resolving conflicts, at most one of them can be executed. In the case of resolvable conflicting operations, operations are executed in the order defined by the ordering relation between the operations. The serialisation order has to conform to the *precedes* relations between the operations.

The main idea of the serialisation mechanism that we used for maintaining consistency can be described as follows. Given the current history buffer $HB=[O_1,O_2,\ldots,O_n]$, the remote operation O_{new} has to be integrated into *HB* such that, by re-executing the operations in the *HB* in a certain order, the partial combined intention of the users is obtained. We reduced the task of finding a serialisation order between $O_1,O_2,\ldots,O_n$ and O_{new} to a graph problem. The operations are represented as nodes of a directed graph. Between two operations O_x and O_y there is a directed arc from O_x to O_y if O_x has to be executed before O_y. The resolvable conflicting operations are therefore represented by means of arcs in the graph. The real conflicting operations are cancelled according to the resolution policy. If an operation is cancelled, its dependent operations also have to be cancelled. In their turn, the cancelled dependent operations cancel their dependent operations. Due to the fact that relations of real conflict between the operations do not impose any ordering between the operations, as opposed to the precedes and resolvable conflict relations between the operations, two directed graphs are constructed: the real-conflict and serialisation graphs. The real-conflict graph determines which operations have to be cancelled due to real conflicts between operations. If there is a real conflict between operations O_1 and O_2 and operation O_2 has a lower priority than O_1, then the real conflict graph contains an edge directed from O_1 to O_2. The serialisation graph determines the order of execution of the operations.

Additional care has to be taken concerning conflicting operations in order to maintain consistency. Suppose that three users concurrently edit the scene of objects illustrated in Figure 2. Suppose that the first user groups the objects identified by id_4 and id_6 into the group identified by id_7. Further, the user groups the

newly created group id_7 with the object id_8, the result being the group id_9. Concurrently, the second user groups the objects identified by id_1 and id_4 into the group id_5 and changes the colour of this group to red. Concurrently with the operations executed by the first two users, the third user groups the objects identified by id_1 and id_2 into the group identified by id_3 and changes the colour of group id_3 to blue. Suppose that the priorities of the sites $Site_1$, $Site_2$ and $Site_3$ are 1, 2 and 3 respectively and, in the case of conflict, the concurrent operation generated from the site with the highest priority wins the conflict. The highest priority corresponds to the lowest integer value assigned. For instance, priority 1 is higher than priority 2. The operations O_1 and O_3 as well as O_3 and O_5 are real conflicting operations as they target common objects. The editing scenario is illustrated in Figure 3.

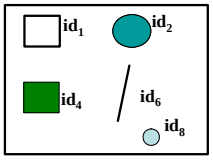

Figure 2. Scene of objects

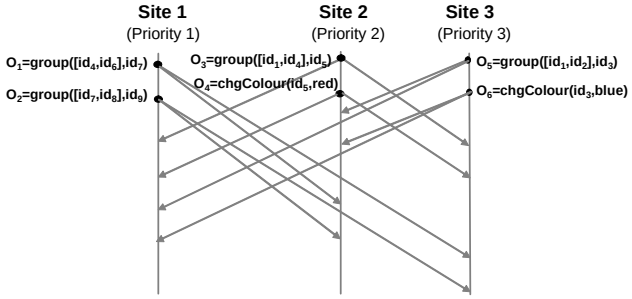

Figure 3. Scenario involving real conflicting operations

Let us analyse the steps for the construction of the graphs at $Site_1$. Due to space limitations, we have drawn the edges of both the real conflict and serialisation graphs for sites 1 and 3, respectively, as single graphs in Figures 4 and 5. The edges of the serialisation graph are drawn using continuous lines, while the edges of the real conflict graph are drawn using dashed lines. After O_1 and O_2 are generated, the corresponding graph illustrated in Figure 4a) contains the edge from O_1 to O_2 as O_2 depends on O_1. When operation O_3 arrives at the site, the real conflict between O_3 and O_1 is detected. As O_1 has a higher priority than O_3, O_3 is cancelled, as illustrated in Figure 4b). When operation O_4 arrives at the site, it is cancelled, as O_3, the operation it depends on, was cancelled. The resulting graph is illustrated in Figure 4c). When O_5 arrives at the site no conflict is detected. When O_6 is received, as illustrated in Figure 4d), the edge from O_5 to O_6 is added to the graph due to the fact that O_6 depends on O_5.

At $Site_3$, after O_5 and O_6 are generated, the edge between O_5 and O_6 is added to the graph, as illustrated in Figure 5a). When O_3 arrives at the site, the real conflict between O_3 and O_5 is detected and operation O_5 is cancelled as O_3 has a higher priority than O_5. As O_5 is cancelled, its dependent operation O_6 is also cancelled. After O_4 arrives at the site, the dependent edge between O_3 and O_4 is added to the graph, as illustrated in Figure 5b). If cancelled operations are not reconsidered, when operation O_1 arrives at the site, the real conflict between O_1 and O_3 is detected and operation O_3 is cancelled as O_1 has a higher priority than O_3. Due to the fact that O_3 is cancelled, O_4 is cancelled too, as O_4 depends on O_3. The graph obtained at this step is illustrated in Figure 5c). When O_2 arrives at the site, the dependent edge between O_1 and O_2 is added to the graph, as illustrated in Figure 5d).

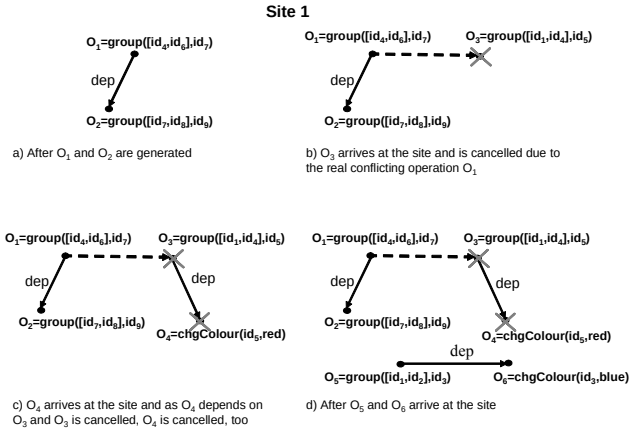

Figure 4. Steps in the construction of the graph at $Site_1$

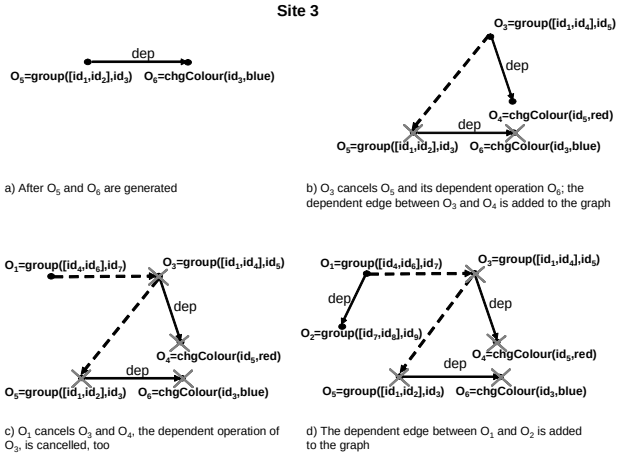

Figure 5. Steps in the construction of the graph at $Site_3$

As we can see, the set of operations that are not cancelled at $Site_1$ and $Site_3$ are not the same, which leads to inconsistency of the shared document at the two sites. The reason is that, due to the order of execution of the real conflicting operations, the maximum set of real conflicting operations that can be executed according to the priorities of the operations has not been considered. The solution to this issue is that the operations that are real conflicting with an operation have to be considered even if they have been cancelled. If an operation is in real conflict with other operations and it has the highest priority, the real conflicting operations having lower priorities have to be cancelled. By cancelling an operation, their dependent operations have to be cancelled recursively. If an operation is cancelled, the operations in real conflict with it have to be reconsidered as some of the cancelled operations in real conflict might be reactivated. By uncancelling an operation, its dependent operations might be reaccepted if they are not cancelled by other operations. The process for the cancellation and reactivation of an operation is repeatedly applied throughout the graph.

In our example, when operation O_1 arrives at $Site_3$, it cancels operation O_3 and its dependent operation O_4, as shown in Figure 5c). By the cancellation of O_3, the real conflicting operation O_5 previously cancelled due to O_3 should be reactivated. Due to the reactivation of O_5, O_6 should be reactivated too. In this way, the set of operations executed at $Site_1$ and $Site_3$ are the same and therefore consistency is achieved.

3.2 Integration of an Operation

We next present the procedure for the integration of an operation into the history buffer.

```
Procedure integrate(O,SGraph,RCGraph,History){
  addNode(O,SGraph);
  addNode(O,RCGraph);
  //add the edges corresponding to the relations
  //between O and the other operations in History
  for(i:=0;i<size(History);i++)
    if(concurrent(O,History[i]))
      if(realConflict(O,History[i]))
        if(priority(O)>priority(History[i]))
             addEdge(O,History[i],RCGraph);
        else
             addEdge(History[i],O,RCGraph);
      else
         if(rightOrderConflict(O,History[i]))
             addEdge(O,History[i],SGraph);
         else
            if(reverseOrderConflict(O,History[i]))
                addEdge(History[i],O,SGraph);
    else
       if(dependsOn(O,History[i])){
           addEdge(History[i],O,SGraph);
           markDependentEdge(History[i],O,SGraph);
       }
       else
           if(precedes(History[i],O))
               addEdge(History[i],O,SGraph);
  //recursively cancel and reactivate nodes in the graph
  propagateCancelOps(O,SGraph,RCGraph);
  //compute the topological sort
  TopSort:=topologicalSort(SGraph,History);
  //find the maximum set of operations from History that
  //conform to the computed topological order
  i:=0;
  while(i<size(History) and History[i]==TopSort[i] and
        not(changedStatus(History[i])))
    i:=i+1;
  //undo the last operations in History that need to be
  //reordered
  for(j=size(History)-1;j>=i;j--) undo(History[j]);
  //redo the undone operations in the order specified by
  //the computed topological sort
  for(j=i;j<size(TopSort);j++) redo(TopSort[j]);
  setHistory(TopSort);
}
```

The procedure *integrate* integrates operation *O* into the history buffer *History*, by taking into account the real conflict graph *RCGraph* and the serialisation graph *SGraph* constructed from the nodes in *History*. The new node *O* is added to *RCGraph* and *SGraph*. For each concurrent operation in *History*, *History[i]*, an edge between *O* and *History[i]* is added in *RCGraph* or *SGraph*, depending on the type of conflict between them. If the two operations *O* and *History[i]* are not concurrent, a check is done whether *O* depends on *History[i]* or whether *History[i]* precedes *O* and the corresponding edges are added to *SGraph*. Due to the insertion of node *O* and the insertion of various types of edges between *O* and other nodes in the graph, some nodes might change their status from accepted to cancelled or the other way around. The recursive propagation of the cancellation and reactivation of nodes is performed in the procedure *propagateCancelOps* presented later on in this subsection.

The topological sort of the nodes in *SGraph* that maintains the maximum set of ordered operations in *History* starting with the first operation is saved into the list *TopSort*. Therefore, the first parts of the lists *History* and *TopSort* are common and the remaining operations in *History* have to be undone and re-executed in the order specified by *TopSort*. The starting index for the process of undoing the operations is determined by traversing the list *History* from left to right and finding the first operation that does not conform to the ordering in *TopSort* or changed its status from accept to reject or the other way around.

We next present the procedure *propagateCancelOps* that recursively propagates the cancellation and reactivation of nodes.

```
Procedure propagateCancelOps(O_new,SGraph,RCGraph) {
  Nodes:=[];
  addFirst(O_new,Nodes);
  setStatus(O_new,cancel);
  //process each node from the list Nodes and check if
  //its status has to be changed
  while(!isEmpty(Nodes)) {
    O:=removeFirst(Nodes);
    changed:=false;
    //if O has status accept
    if(getStatus(O)==accept){
      //suppose the final status of O remains accept
      isFinalState:=true;
      //if other operations cancel O, the status of O
      //becomes cancel
      DepOps:=getNeighbours(filterDependent(
                            getInboundEdges(O,SGraph)));
      for(i=0;i<size(DepOps);i++)
        if(getStatus(DepOps[i])==cancel) {
          setStatus(O,cancel);
          isFinalState:=false;
          break;
        }
      if(isFinalState) {
       RCOps:=getNeighbours(getInboundEdges(O,RCGraph));
       for(i=0;i<size(RCOps);i++)
          if(getStatus(RCOps[i])==accept) {
            setStatus(O,cancel);
            isFinalState:=false;
            break;
          }
      }
      //if an operation cancelled O, O changed its status
      if(!isFinalState) changed:=true;
   }
   else {
    //if O has status reject, assume O changes its status
    isFinalState:=false;
    //if other operations cancel O, O keeps its status
    DepOps:=getNeighbours(filterDependent(
                          getInboundEdges(O,SGraph)));
    for(i:=0;i<size(DepOps);i++)
       if(getStatus(DepOps[i])==cancel) {
         isFinalState:=true;
         break;
       }
    if(!isFinalState) {
       RCOps:=getNeighbours(getInboundEdges(O,RCGraph));
       for(i=0;i<size(RCOps);i++)
          if(getStatus(RCOps[i])==accept) {
             isFinalState:=true;
             break;
          }
    }
    //if no operation canceled O, O has to change its
    //status to accept
```

```
    if(!isFinalState){
       changed:=true;
       setStatus(O,accept);
    }
   }
   //if status of O changed, add to Nodes the operations
   //that might change their status due to O
   if(changed) {
     setChangedStatus(O);
     for(O in getNeighbours(filterDependent(
                           getOutboundEdges(O,SGraph))))
       addLast(O,Nodes);
     for(O in getNeighbours(getOutboundEdges(O,RCGraph)))
       addLast(O,Nodes);
   }
 }
}
```

The first argument of procedure *propagateCancelOps* is operation O_{new} that was added to the *SGraph* and *RCGraph* graphs. It might be cancelled or generate the cancelling of other operations. The second and third arguments of the procedure are the graphs *SGraph* containing the ordering relations between operations and *RCGraph* containing the real conflicting operations.

The idea of the algorithm is to add the nodes that might change their status to a list and check, for each node in the list, whether it can keep its current status and then add to the list the nodes that it might in turn cause to change their status.

The list *Nodes* contains the nodes whose status has to be checked. At the beginning, operation O_{new} is added to the list with the status of a cancelled operation.

A set of iterations is performed over the list *Nodes* and, at each step, the first element in the list is checked to determine if it can keep its status. If the operation has to change its status from accepted to cancelled or the other way around, its dependent nodes and its real conflicting nodes that have a lower priority are added to the list *Nodes*. No more iterations have to be performed when the list *Nodes* is empty. For the first operation *O* in the list *Nodes*, two cases are distinguished depending on whether the status of *O* is accept or cancel.

The flag *isFinalState* indicates whether operation *O* can keep its status. In the case that the status of *O* is accept, we make the assumption that the final state of *O* is accept and therefore set the flag *isFinalState* to *true*. A check has to be done whether *SGraph* contains a cancelled operation on which *O* depends that cancels *O* or *RCGraph* contains an active conflicting operation that cancels *O*. If it is the case, *O* has to be cancelled and therefore *isFinalState* has to be set to *false*. This means that *O* changed its status and therefore the flag *changed* is set to *true*.

In the case that the status of *O* is cancel we make the assumption that this is not the final state of *O* and therefore set the flag *isFinalState* to *false*. A check is done whether *SGraph* contains a cancelled operation on which *O* depends that cancels *O* or *RCGraph* contains an active conflicting operation that cancels *O*. If it is the case, *O* has to be cancelled. Therefore, *O* keeps its original state and the flag *isFinalState* has to be set to *true*. If there is no operation that cancels *O*, i.e. *isFinalState* remains set to *false*, it means that our assumption that the final state of *O* is accept holds. Therefore, *O* changed its status and flag *changed* has to be set to *true*.

If, as result of the verifications, operation *O* changed its status, all operations that might change their status due to the changing of the status of *O* are the dependent operations on *O* and the real conflicting operations of a lower priority than the priority of *O*.

The serialisation order of the operations is the topological sort of the serialisation graph. If the graph has a cycle there is no solution for the serialisation order. As a graph has a set of corresponding topological sort orders, there are different ways of reordering the operations. To find an order between O_1, O_2,…,O_n, O_{new}, we chose the topological sort that maintains the maximum set of ordered operations in $HB=[O_1,O_2,\ldots,O_n]$ starting with the first operation. Therefore, a minimum number of operations from *HB* have to be undone in order to perform the reordering of operations.

The *topologicalSort* function together with the auxiliary procedure *addResult&Update* that it calls are now presented.

```
Function topologicalSort(Graph,History):Result {
  Result:=[];
  Nodes:=getNodes(Graph);
  //NoEdges is a hashmap containing pairs between nodes
  //and the in-degrees of those nodes
  for (i=0;i<size(Nodes);i++)
    put(NoEdges,(Nodes[i],getInDegree(Nodes[i],Graph)));
  k:=0;
  //add those nodes belonging to History to Result
  //if their in-degrees=0 and their status did not change
  while(k<size(History) and get(NoEdges,History[k])==0
        and not(changedStatus(History[k]))){
    addResult&Update(Result,History[k],Graph,NoEdges);
    k++;
  }
  //sort the remaining nodes in their topological order
  //iterate size(Nodes)-k times over Nodes and choose
  //each time a node of in-degree 0
  for(i=0; i<size(Nodes)-k; i++) {
     ind=-1;
     for(j=0; j<size(Nodes); j++)
       if(get(NoEdges,Nodes[j])==0)
         if(ind==-1) ind:=j;
         else
           if(getId(Nodes[j])<getId(Nodes[ind])) ind:=j;
     //if there is no node having the in-degree=0,
     //no topological order exists
     if (ind==-1){
        Result:=null;
        return Result;
     }
     addResult&Update(Result,History[ind],Graph,NoEdges);
  }
  return Result;
}

Procedure addResult&Update(Result,Node,Graph,NoEdges) {
    append(Result,Node);
    //mark Node to not be considered in the next step of
    //the topological order
    put(NoEdges,(Node,-1));
    //decrease the in-degree of the neighboring nodes
    //corresponding to the outgoing edges of Node
    for(NeighbourNode in getOutNeighbours(Graph,Node)) {
       NeighbourInDegree:=get(NoEdges,NeighbourNode);
       put(NoEdges,(NeighbourNode,NeighbourInDegree-1));
    }
}
```

The *topologicalSort* function takes as arguments the serialisation graph *Graph* that is used for the reordering of operations and the old history buffer *History* before the integration of the new operation. The function reorders the nodes in the graph according to the ordering relations between the operations represented by the edges in the graph. Note that *Graph* contains the new operation that has to be integrated in the history buffer, while *History* does not contain it. The function returns the reordered list

of operations in the serialisation graph. Therefore, *Result* will represent the new history buffer.

The process of building a topological sort of a graph implies considering, in turn, the nodes that have the in-degree 0. After a node that has the in-degree 0 is considered, the edges from that node to the neighbouring nodes are removed.

Nodes is initialised with the list of nodes in the graph and *NoEdges* is a hash map containing, for each node in the graph, the in-degree of the node, i.e. the number of edges having that node as target. The history is traversed from left to right and, as long as an operation *History[k]* has the in-degree 0 and has not changed its status in the recursive process of cancellation of operations, it is considered in the topological sort. During the process of cancellation, some operations might have changed their status from being cancelled to being accepted or the other way around and, therefore, the ordering between these operations and the other operations might have changed and it no longer conforms to the ordering in *History*. The procedure *addResult&Update* appends the operation *History[k]* to *Result* and, in order that *History[k]* is no longer considered in the ordering process, the operation is marked to have the in-degree -1. The neighbouring operations of *History[k]* in the graph corresponding to the outgoing edges have to have their in-degree decreased by 1, as operation *History[k]* has been already considered in the topological sort and its neighbouring edges have to be eliminated.

After the operations in *History* have all been considered or an operation is encountered that either has in-degree≠0 or has its status changed, the other operations in the graph that were not considered have to be added to the topological sort. The number of nodes that still have to be added to the topological sort is equal to the number of total nodes minus the number of operations in *History* that have been included in the topological sort.

A node can be added to the topological sort list when it has an in-degree equal to 0. But, there are more nodes that may have an in-degree equal to 0. The history buffers that would be obtained by different topological sorts would be equivalent. Obtaining the same history that contains the operations in a global order is useful in the undo process of global operations. In order to obtain the same history at all sites, we use the criteria that, when two nodes have their in-degree equal to 0, we choose to execute first the operation that was generated from the site with the lowest identifier. Note that it is not possible for two operations generated at the same site to have an in-degree equal to 0, as, between the two operations, a precedes relation exists and thus one of the two operations has an in-degree greater or equal to 1.

A number of iterations equal to the difference between the total number of nodes in the graph and the number of operations in *History* that have been included in the topological sort have to be performed. In each iteration, a node that has an in-degree equal to 0 has to be chosen and it has to be the operation generated from the site with the lowest identifier from the set of operations that have not been considered and have an in-degree equal to 0. In the case that, in one of these iterations, no operation with in-degree 0 is found, the returned result is null, meaning that there exists no topological sort. This case occurs if the set of conflicts between the operations was not correctly defined. In the case that an operation satisfying the above-mentioned conditions is found during the iteration, the procedure *addResult&Update* is called. By the call of the procedure, the operation is appended to *Result*, its corresponding entry in the hashtable *NoEdges* is updated with value -1 and the in-degree of its neighbours in the graph is updated as result of the deletion of the outgoing edges of the operation.

In what follows we point out some issues encountered for maintaining consistency in graphical editing. The first issue was working with pages. The pages of the document conform to a linear structure and special attention had to be given to maintaining consistency in the presence of concurrent operations that insert and delete pages. We defined the operation of deletion of a page as *DeletePage(PageId)*, where *PageId* is the identifier of the page to be deleted. The operation of insertion of a page is defined as *InsertPage(BeforePageId)*, where *BeforePageId* is the identifier of the page before which the insertion has to be performed. The problem occurs if one user inserts a page, while another user concurrently deletes the page before which the insertion has been performed. The solution to this problem was to find a serialisation order between the two concurrent operations of insertion and deletion of a page, such that an insertion of a page is performed before the deletion of a page.

The same solution that we applied for maintaining consistency over the pages of a document could be applied for maintaining consistency over text documents. Text documents are viewed as a sequence of characters, each character having assigned a unique identifier. The set of operations that can be performed on text documents are insertions and deletions of characters. The operation of deletion of a character specifies as argument the character to be deleted and the operation of insertion of a character takes as arguments the character to be inserted and the character identifier before which the insertion has to be performed.

As for the consistency maintenance for document pages, the cases that need special attention are the concurrent insert operation of a character and the deletion of the character before which the insertion has to be performed. The solution is to establish a serialisation order between the two operations, to execute first the insertion of the character followed by the deletion of the character. Another special case is the one when two concurrent operations insert characters at the same position. The solution is to establish a serialisation order between the two operations, to execute first the insert operation generated from the site with the lower identifier followed by the insert operation generated from the site with the higher identifier.

Another issue that we mention here is that objects and groups have an associated z-order. The case that needs special attention is when two concurrent operations create a new object. The solution is to execute these operations in a certain order at all sites. We adopted a serialisation order between concurrent operations based on the identifier of the sites where the operations were generated. Layers also have an associated z-order and the same issues and solutions apply.

3.3 Definition of Conflicts

The list of types of conflicts is specified by users in a separate document and can be modified according to various applications. A conflict is given by the specification of the type of the two operations in conflict, the condition for conflict and the type of conflict. We illustrate this by means of some examples showing how conflicts can be defined.

A delete operation *o1* is in a resolvable conflict with a group operation *o2* if the target object of *o1* belongs to the target list of *o2*. The conflict is resolved by executing first the *o2* operation followed by the *o1* operation.

```
conflict {
   operation o1:delete o2:group
   condition o1.id in o2.inlist
   resolution reverse
}
```

Two concurrent operations moving the same point of a polyline are in real conflict. The resolution policy is specified to be none, meaning that the conflict is a real conflict.

```
conflict {
  operation o1:pointmove o2:pointmove
  condition o1.id=o2.id and o1.pointid = o2.pointid
  resolution none
}
```

Two change colour operations *o1* and *o2* are in conflict if the target of operation *o1* belongs to the group targeted by operation *o2*. The resolution policy is to execute operation *o2* first followed by operation *o1*. An additional condition has to be stated that the target objects of the two operations are different. The case when the two change colour operations target the same object is specified as being a real conflict in another conflict rule.

```
conflict {
  operation o1:changecolor o2:changecolor
  condition o1.id != o2.id and o1.id childof o2.id
  resolution reverse
}
```

3.4 Draw-Together Application

An editor based on the algorithms described in this paper has been implemented. A screenshot of our Draw-Together application interface is given in Figure 6. The interface illustrates the collaborative work of three users performing a brainstorming session about the organisation of the Collaborative Editing Workshop. One can notice the set of primitives that can be used for drawing, such as rectangles, lines, ellipses, polylines, text boxes, bitmaps and annotations and the set of operations that can be performed by means of the various buttons included in the toolbars. Note also the use of multi-pages and layers. The *Workshops* and *Organisers* layers can be set visible or invisible, depending on whether the details about the related workshops and organisers of the workshop want to be shown or hidden.

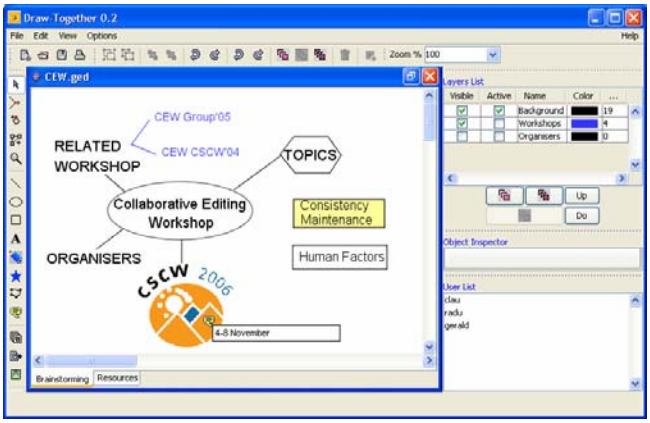

Figure 6. Draw-Together application

The Draw-Together application is based on requirements for collaborative drawing as specified by a research group in the engineering department of our university who are especially interested in supporting the early stages of product development involving product sketching and brainstorming sessions.

4. RELATED WORK

As our algorithm is based on the serialisation mechanism, in the first part of this section, we relate our algorithm to other serialisation approaches, such as Bayou[16], Sync[9], IceCube[11] and the ACF framework[12]. In the second part of this section we relate our approach to other graphical editors.

Bayou[16] is a replicated database system. A site applies operations tentatively as they are received from the local or remote sites. A tentative timestamp is assigned by a site to an operation as it arrives at that site. The final timestamp is the time the operation is accepted by the primary site. Bayou produces a schedule in timestamp order. Operations are first executed by a site in their tentative order and then undone and redone in the final order. Conflicts are detected by an explicit precondition called dependency check attached to an operation and they are resolved by an application-defined merge procedure also attached to each operation. The primary site orders the operations and resolves the conflicts as they arrive and then propagates the decisions to the other sites. As opposed to the Bayou system that requires a primary site for deciding the final order between operations, our approach is distributed and the final order is incrementally built by each site.

IceCube [11] is a generic system for reconciliation which is viewed as an optimisation problem of scheduling a maximum number of concurrent actions, given a set of constraints. The constraints can be static or dynamic. In the case of static constraints, they are evaluated without using the current state of objects. In the case of dynamic constraints, the success or failure of a single action depends on the current state of objects. A scheduling stage produces schedules that satisfy the static constraints. The schedules obtained in this phase are verified in a simulation stage, where actions are executed against a copy of the state to check the dynamic constraints. At the end, a selection stage chooses the schedules that satisfy the dynamic constraints. In order to achieve convergence of the *n* sites involved in the collaboration, in the IceCube approach, a common site is responsible for achieving the reconciliation. All the other sites have to send all operations executed since the last synchronisation to this site. The reconciliation mechanism is applied and the combined log is sent to all sites. The reconciliation phase is NP-hard. In our approach, the constraints are defined by the relations between the operations: preceding, right order, reverse order, dependence and real conflict. We do not distinguish between static and dynamic constraints. As opposed to the centralised approach of IceCube, our approach is distributed. Moreover, the algorithm that we designed is incremental and it integrates a new operation into an already computed schedule by reusing a maximal set of orderings that remain valid after the integration of the remote operation. The new schedule satisfies the total set of constraints between the remote operation and the previously integrated operations. Bayou and IceCube approaches do not describe their application for graphical editing.

Sync[9] application provides high-level primitives in the form of predefined classes that enable programmers to create synchronised, replicated data objects. The Sync approach for merging is based on the merge-model described in [8]. The merge

matrix defines merge functions for the possible set of operations, such as deletion of an element, the insertion of an element after another element in the sequence and the modification of an element. An application of Sync for a drawing-based collaborative tool has been proposed in [9]. Sync allows a merge of a user's change with the server, but requires that the server's change always wins the conflict. Our approach does not require a server for the synchronisation, but an operation generated at a site is propagated immediately to the other sites and at each site an incremental process for the integration of an operation is applied. Moreover, the graphical application proposed in [9] does not deal with operations of grouping and ungrouping that would change the structure of the tree.

In [12] a formalism for modelling replication in a distributed system where users concurrently work on shared data has been proposed. The approach is based on actions representing operations executed by users and a set of constraints representing scheduling relations between actions. The approach offers support for reasoning about consistency properties of replication protocols. The Action-Constraint Framework [13] is a continuation of the work in [12] and it defines the three dimensions for consistency by mapping to three problems on subgraphs of a multilog: conflict breaking, agreement and serialisation. Each site has a local view called a multilog of known actions and constraints. The constraints defined for the multilog are the following:

- "α Before β" indicating that α should be executed before β. A schedule that executes neither α nor β, or only α or only β or both α and β in this order is correct with respect to this constraint.
- "α MustHave β" indicating that if α is executed then β must also be executed, although not necessarily in this order. A schedule that executes only β, or that executes neither α nor β is correct with respect to this constraint.
- "α non-commuting β" indicating that α does not commute with β. Two actions commute if by executing them in either order an equivalent state is obtained.

An initial graph is constructed, where the nodes of the graph represent actions and the edges represent the *before*, *mustHave* and *non-commuting* relations between actions. As previously mentioned, the consistency problem is divided into three subproblems by the division of the initial graph into three other graphs: *before*, *mustHave* and *serialisation* graphs. The before graph contains the before edges from the initial graph, the mustHave graph contains the mustHave edges from the initial graph and the serialisation graph contains the before and non-commuting edges from the initial graph. The processing of these graphs offers support for reasoning about the correctness of different consistency protocols.

We can map our problem for maintaining consistency to the ACF framework. Our precedes and resolvable conflicting relations between the operations can be mapped to the before relations in the ACF framework. A depends relation between two operations in our approach is mapped to both a before and a mustHave relation. The operations excluding the ones that are in a real conflict, resolvable conflict or are dependent are considered commuting. The cancelled operations correspond to the set of dead actions in ACF, i.e. actions that are not executed in any schedule. In our approach, we adopted an incremental way of integrating remote operations in the schedule that was built from the previous set of operations. The approach of ACF framework rather considers the whole set of actions that have to be taken into account in building the schedule. Our approach also deals with a graph that contains the operations as nodes and the relations between nodes as edges, similar to the serialisation graph. Therefore, our approach based on a graph and constraints between operations can be mapped to the ACF framework that represents a theoretical formalism for modeling replication. However, in this paper we provided detailed algorithms for the incremental process for the integration of an operation applied for graphical editing.

Most of the existing collaborative object-based graphical editors such as the ones presented in [4,5,6,7,15] do not consider grouping operations. We therefore do not compare our approach with these systems, but instead relate our work with previous collaborative editing approaches dealing with group/ungroup operations [18,3].

The CoGroup [18] approach is an alternative solution to our operation serialisation mechanism for the grouping of graphical objects. It uses an operational transformation (OT) mechanism proposed in the context of the Transparent Adaptation (TA) approach to convert existing single-user editing applications into real-time collaborative applications without changing their source code. In the TA approach, the shared single-user application is replicated at all sites and the API of the single user applications can intercept the user operations. The intercepted operations are then processed by an OT framework that achieves a combined effect of the multi-user interactions. The transformed operations are then sent back to the API of the single-user applications that generates the corresponding operations for the single-user applications. In the case of two conflicting operations, the MVSD (Multi-Version Single-Display) technique has been applied, according to which, multiple versions of the common target objects are created to accommodate the effects of all conflict operations, but only one version is displayed. Users are allowed to choose to display any version at a time by using the system undo facility. However, no solution is provided for the navigation between the versions associated to an object.

The advantage of the operational transformation approach is that no undo/redo mechanism is required. However, our serialisation mechanism undoes the minimum number of operations in the integration process of a new remote operation in order to satisfy the set of constraints. The advantage of our approach is that it offers a flexible way of defining conflicts according to application needs. Conflicts were classified into real and resolvable. For instance, if an application wants to define that a *group* operation is conflicting with another operation that has a common target, it can do this by defining the two types of operations as real conflicting. A combined effect of these operations can be obtained by defining them as being in a resolvable conflict, i.e. specifying an order of execution, such as *group* operation followed by the other operation.

In the case of two concurrent operations that target a set of objects, versions of the targeted objects are created, but only one of the versions according to operation priorities is displayed. The same solution displayed for solving conflicts between any pairs of operations can be achieved in our approach by defining a serialisation order between operations. However, some applications need to restrict concurrent operations targeting some

common objects and our approach offers the possibility to define these operations as real conflicting.

The work proposed in this paper extends the approach described in [3] where the policies for handling conflicts are fixed, by allowing a flexible definition and resolution of conflicts according to various application needs. To achieve a flexible handling of conflicts and the possibility of defining various ordering constraints between operations, a novel serialisation mechanism has been applied. The serialisation mechanism is based on a graph and performs the reordering of nodes in a graph based on the ordering constraints between operations. The mapping of the serialisation mechanism to a graph problem offers support for a formalisation of the consistency maintenance approach, for an optimisation of the number of operations that have to be undone and for proving correctness [13]. Moreover, the set of primitives subject to collaboration and the operations that can be performed on the scene of objects have been extended compared to [3]. For instance, we allow multi-page documents and working with layers.

5. CONCLUSIONS

In this paper, we propose a novel operation serialisation approach for maintaining consistency in collaborative object-based graphical editing. Our approach relies on the topological sort of nodes in a graph where the nodes of the graph represent executed operations and edges of the graph represent ordering constraints between operations. A collaborative graphical editor has been implemented based on the algorithms described in this paper. According to various applications, the users can specify resolution policies for concurrent operations depending on whether the operations are considered to be in conflict and only the operation with the highest priority should be executed or whether an order of execution between the operations should be specified. The graph approach offers support for proving the correctness of the consistency maintenance algorithm.

In our future work, we are going to identify the set of conditions that have to be fulfilled by the set of defined conflicting operations for the existence of a topological sort. One of our next directions for future work is to perform user studies together with our engineering colleagues on the use of the Draw-Together application in collaborative product design.

6. REFERENCES

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

[2] 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*, Kuwaii, Hawaii, January 1992, 138-150.

[3] Ignat, C.L. and Norrie, M.C. Grouping in Collaborative Graphical Editors. *Proc. of CSCW*, Chicago, Illinois, USA, 2004, 447-456.

[4] Ionescu, M. and Marsic, I. An Arbitration Scheme for Concurrency Control in Distributed Groupware. *The 2nd Intl. Workshop on Collaborative Editing Systems, CSCW*, Philadelphia, Pennsylvania, USA, December 2000.

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

[6] Karsenty, A., and Beaudouin-Lafon, M. An algorithm for distributed groupware applications. *Proc. of Conf. on Distributed Computing Systems*, Pittsburg, Pennsylvania, USA, May 1993, 195-202.

[7] 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.

[8] Munson, J.P. and Dewan, P. A flexible object merging framework. *Proc. of CSCW*, Chapel Hill, North Carolina, USA, 1994, 231-242.

[9] Munson, J.P. and Dewan, P. Sync: A Java Framework for Mobile Collaborative Applications. *Computer*, 30(6), 1997, 59-66.

[10] Newman-Wolfe, R.E., Webb M., and Montes, M. Implicit locking in the Ensemble concurrent object-oriented graphics editor. *Proc. of CSCW*, Toronto, Canada, 1992, 265-272.

[11] Preguica, N., Shapiro, M., Matheson, C. Semantics-Based Reconciliation for Collaborative and Mobile Environments. *Proc. of CoopIS*, Catania, Italy, 2003.

[12] Shapiro, M., Bhargavan, K., Krishna, N. A Constraint-Based Formalism for Consistency in Replicated Systems. *Proc. of OPODIS*, Grenoble, France, December 2004, 331-345.

[13] Shapiro, M., Krishna, N. The three dimensions of data consistency. *Journées Francophones sur la Cohérence des Données en Univers Réparti (CDUR)*, Paris, France, 2005, 54-58.

[14] Suleiman, M., Cart, M. and Ferrié, J. Concurrent Operations in a Distributed and Mobile Collaborative Environment. *Proc. of ICDE*, Orlando, Florida, USA, February 1998, 36-45.

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

[16] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J. and Hauser, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system, *Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, USA, December 1995, 172-182.

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

[18] Xia, S., Sun, D., Sun, C., Chen, D. Collaborative Object Grouping in Graphics Editing Systems. *Proc of CollaborateCom*, San Jose, CA, USA, December 2005.