Collaborative Object Grouping in Graphics Editing Systems

Steven Xia
Griffith University
Brisbane, QLD 4111, Australia

David Sun
University of California at Berkeley
Berkeley, CA, USA

Chengzheng Sun
Nanyang Technological University
Singapore 639798

David Chen
Griffith University
Brisbane, QLD 4111, Australia

Abstract

Object grouping is an effective means for managing the complexity in graphics editing. However, research on collaborative object grouping has not been adequate. In this paper, we contribute a novel collaborative object grouping technique, called CoGroup. CoGroup can achieve maximal combined effects among compatible operations and preserve all users' work in the face of conflict without the overhead of undoing and redoing conflict operations as in existing serialization approaches. CoGroup has been implemented in collaborative word processing (CoWord) and slide authoring (CoPowerPoint) systems and is generally applicable to a range of off-the-shelf commercial graphics applications, particularly CAD/CASE tools.

1. Introduction

Object-based graphics editing systems are a special type of graphics editing systems. A document of such systems consists of a collection of graphic objects like lines, circles, text boxes, etc. Each object has a set of attributes, such as color, size, position, etc. Users are allowed to create, delete objects or modify object attributes. Object-based graphics editing is the foundation of a wide range of off-the-shelf commercial applications, including slide authoring systems (e.g. Microsoft PowerPoint), word processors (e.g. Microsoft Word), CAD systems (e.g. AutoCAD) and CASE systems (e.g. Rational Rose). The goal of our research is to apply the *Transparent Adaptation* (TA) approach [16] to convert existing single-user graphics editing applications into real-time collaborative versions without changing their source code.

Documents of graphics editing applications (e.g. CAD systems) often contain a large number of objects with complex logical structures. Managing complex structures on the basis of individual object would cost significant efforts or sometimes may be infeasible. Object grouping, which packs multiple logically related objects into a single *group-object* and vice versa, is an effective means to help manage the complexity of graphics editing. When objects are grouped, they behave like a single object in response to modifications to any attribute. At the same time, some attributes (e.g. fill color) of group members can still be modified individually. Furthermore, a group-object can be a member in another group-object, which provides a multi-level hierarchical structure for managing complex documents. In summary, object grouping can not only prevent mistaken actions from breaking the logical relationship among group members, but also provide the convenience of modifying group members individually. Object grouping is a practically useful and frequently used function in existing single-user graphics editing applications, and thus must also be supported in TA-based multi-user collaborative versions.

Supporting collaborative object grouping is nontrivial due to the increased complexity in both the data model and the operation model of the collaborative graphics editing technique. First, existing collaborative graphics editing techniques often treat graphic objects as independent entities, but object grouping introduces the group relationships among graphic objects. Second, existing collaborative graphics editing techniques focus on supporting three types of *basic* operations: a *CreateObj* operation creates a new object (e.g. a line, circle, square, or textbox); a *DeleteObj* operation removes an existing object; and a *ChangeAtt* operation changes an attribute (e.g. size, color, or position) of an existing object. Object grouping requires support for two additional operations: a *Group* operation packs a collection of objects into a single group-object; and an *Ungroup* operation unpacks a group-object into a collection of member objects. We shall use the term *grouping operation* to mean either a *Group* or an *Ungroup* operation. The main technical challenge here is conflict resolution and consistency maintenance in the presence of group-objects and grouping operations in a TA-based real-time collaborative environment.

While collaborative editing has been an active area of research in the past decades, little has been done on the techniques for supporting collaborative object grouping. To our knowledge, there is only one prior work on collaborative object grouping in graphics editing systems, which is based on *operation serialization* [5]. In this work, undo and redo strategies are used to reorder operations for consistency maintenance in the face of conflict. Apart from the inherent high complexity and overhead involved in operation serialization, the work in [5] is incapable of preserving all users' work in the face of conflict and is not suitable for application to existing graphics editing systems (detailed analysis and comparison shall be given in Section 5). In this paper, we contribute a novel collaborative object grouping technique, called *CoGroup*, which is based on the *Operational Transformation* (OT) technique [12][14][15] and on the TA approach [16]. The use of OT enables the *CoGroup* technique to resolve conflicts among grouping operations without using internal undo/redo; and the TA approach makes the *CoGroup* technique applicable to a wide range of existing graphics editing systems without changing their source code. The *CoGroup* work is the first collaborative

1-4244-0030-9/05/$20.00 ©2005 IEEE

object grouping technique based on OT and TA, and has been implemented in the CoWord and CoPowerPoint systems [16]. This paper reports the main research findings in designing and implementing the *CoGroup* technique.

The rest of this paper is organized as follows. Section 2 introduces background knowledge about OT and TA. Section 3 defines the conflict relations and MVSD combined effects among conflict and/or compatible basic and grouping operations. Section 4 discusses technical issues and solutions in supporting object grouping in the TA framework. In Section 5, the *CoGroup* approach is compared with related work. Finally, contributions and future work are summarized in Section 6.

2. Background on OT and TA

2.1 Basics of the OT Technique

OT was originally designed to support multiple users to insert and delete characters in replicated text documents concurrently and consistently [3][12][14]. The basic idea of OT is to transform an editing operation defined on a previous document state according to the effects of executed concurrent operations, so that the transformed operation can achieve the correct effect in the current document state. Despite its text editing origine, OT is independent of text documents and text editing, and has been applied to support consistency maintenance and user-initiated *undo* in collaborative editing of both text and graphics documents [1][8][10][15][16].

There are two underlying models in the OT technique: one is the *data address model* which defines the way data objects in a document are addressed by operations; the other is the *operation model* which defines the set of operations that can be directly transformed by OT functions. Different OT techniques may have different data and operation models.

In this paper, we assume the OT data address model is a tree of multiple linear address domains [2], as shown in Figure 1. In this model, a data object is mapped to a position in a linear addressing domain only if it has the position number as its address in this domain. A data object is a *terminal* object if it has no internal data structure or its internal data structure is not addressable. A data object is an intermediate object if it has an addressable internal data structure. A terminal object has no link out of it, but an *intermediate* object has a link leading to a lower level addressing domain, which represents this object's internal addressing space. An object in this data address model can be uniquely addressed by a vector of position integers: $vp = [p_0, p_1, ..., p_i, ..., p_k]$ where $vp[i] = p_i, 0 \le i \le k$, represents one addressing point at level i.

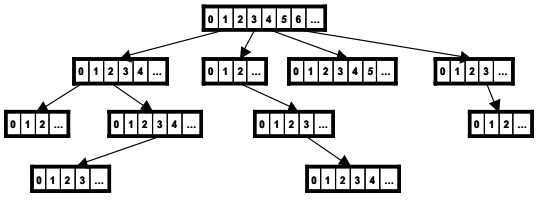


Figure 1. The OT address model.

The *OT operation model* assumed in this paper consists of three generic *Primitive Operations* (PO) [15]:

1. *Insert [pos, obj]* denotes inserting object *obj* at position *pos.*
2. *Delete [pos, obj]* denotes removing object *obj* at position *pos.*
3. *Update [pos, key, old_value, new_value]* denotes changing the attribute *key,* from *old_value* to *new_value,* of an object at position *pos.*

These POs are generic in the sense that they are independent of object types. With these POs, OT does not need any application-specific knowledge to do its work.

2.2 Basics of the TA Approach

TA is an innovative approach to converting single-user applications for multi-user real-time collaboration, without changing the source code of the original application [16]. The TA approach is based on a replicated system architecture where the shared single-user application is replicated at all collaborating sites, the use of the single-user application's API (Application Programming Interface) to intercept and replay the user's interactions with the shared application, and the use of the OT technique to manipulate the intercepted user operations for supporting responsive and unconstrained (i.e. concurrent and free) multi-user interactions with the shared application. The central idea of the TA approach is to adapt the data address and operation models of the shared application's API to that of the OT technique.

More precisely, the TA approach can be described by a reference model, as shown in Figure 2. This reference model consists of three components: *Single-user Application* (SA), *Collaboration Adaptor* (CA), and *Generic Collaboration Engine* (GCE). The main functionalities of these components are sketched below.

The SA component provides conventional single-user interface features and functionalities. This component can be either an existing commercial off-the-shelf single-user application, or a new single-user functionality component in a multi-user collaborative system, but this component itself has no knowledge about multi-user collaboration.

The CA component provides application-specific collaboration capabilities and plays a central role in adapting the SA for collaboration. This component has the knowledge of the SA API but not its internals. At the center of this component is the module of *Adapted Operation* (AO), which represents the SA functionalities exposed by the API. The AO can be generated by the *Local Operation Handler* (LOH) module by intercepting local user's interactions, or received by the *Remote Operation Handler* (ROH) module from remote users. With the AO residing between the API and OT, the task of adaptation between the API and OT is decomposed into two modules:

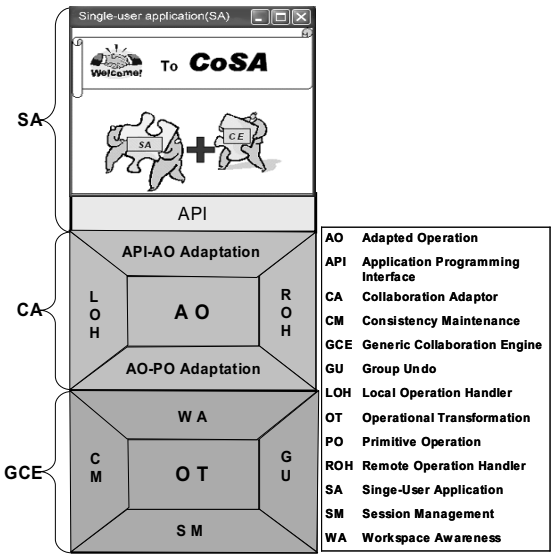


Figure 2: The TA reference model.

1. The *API-AO Adaptation* module is responsible for bridging the semantic gap between the API and the AO so that the AO can be correctly replayed on the SA.
2. The *AO-PO Adaptation* module is responsible for mapping between the AO and OT-supported PO so that the underlying OT technique can be used to ensure the correctness of the AO parameters in the presence of concurrency.

The GCE component provides application-independent collaboration capabilities. This component has no knowledge of the single-user application functionality and therefore can be used in adapting different applications. This component encapsulates a package of collaboration supporting techniques, including *Consistency Maintenance* (CM), *Group Undo* (GU), *Workspace Awareness* (WA), and *Session Management* (SM), etc. OT is at the core of this component for supporting consistency maintenance, user-initiated undo, and workspace awareness in a collaborative environment.

3. Conflict Resolution in the Presence of Grouping Operations

3.1 Conflict Relations Among Operations

In an unconstrained collaborative environment, operations can be generated concurrently, and concurrent operations may conflict with each other if they target common objects and their effects cannot be accommodated in the target or result objects at the same time. For example, multiple users may simultaneously generate *ChangeAtt* operations to change the same attribute (e.g. size, color, or position, etc.) of the same existing object. These concurrent *ChangeAtt* operations conflict since their effects cannot be accommodated within the same target object at the same time. Moreover, two concurrent *Group* operations may also conflict with each other if they target common objects since these common objects cannot belong to two different result group-objects at the same time. In [11] and [15], we have discussed in detail how to define and resolve conflicts among *ChangeAtt* operations. In this paper, we extend our prior work on conflict definition and resolution to grouping operations.

To define the conflict relation, we use the following notions: (1) *Type*(*O*) denotes the type of operation *O*; (2) *Target*(*O*) denotes the set of identifiers of target objects of operation *O*; and (3) *Att.Key*(*O*) denotes the attribute type of operation *O* if *O* is a *ChangeAtt* operation.

Definition 1. Conflict relation "$\otimes$". Two operations *O1* and *O2* conflict with each other, expressed as *O1* $\otimes$ *O2*, if and only if (1) *O1* and *O2* are concurrent; (2) *Target*(*O1*) $\cap$ *Target*(*O2*) $\neq \Phi$; and (3)

a. *Type*(*O1*) = *Type*(*O2*) = *Group*; or

b. *Type*(*O1*) = *Type*(*O2*) = *ChangeAtt* and *Att.Key*(*O1*) = *Att.Key*(*O2*).

Definition 2. Compatible relation "$\odot$". Two operations *O1* and *O2* are compatible, expressed as *O1* $\odot$ *O2*, if and only if they do not conflict with each other; that is, $\neg$(*O1* $\otimes$ *O2*).

According to the above definitions, sequential operations are compatible; operations without common target objects are compatible; and operations of different types are compatible. Conflict relations occur only between a pair of *Group* operations or a pair of *ChangeAtt* operations under the conditions specified in *Definition 1*. The conflict/compatible relations among the three basic operations and the two grouping operations are summarized in Table I (called a conflict relation triangle in [11]). The meaning of shaded cells will be explained in Section 4.6.

Table I. Conflict relation triangle of five operation types

	CreateObj	***DeleteObj***	***ChangeAtt***	***Group***	***Ungroup***
CreateObj	$\odot$	$\odot$	$\odot$	$\odot$	$\odot$
DeleteObj		$\odot$	$\odot$	$\odot$	$\odot$
ChangeAtt			$\otimes$/$\odot$	$\odot$	$\odot$
Group				$\otimes$/$\odot$	$\odot$
Ungroup					$\odot$

3.2 Conflict Resolution by MVSD

For compatible operations, they can be applied without any special treatment and their effects can be combined in the target objects even if they target common objects. For conflict operations, however, special treatment is needed to resolve their conflict and maintain system consistency.

There are three possible ways of resolving operation conflict while maintaining consistency [11][15]:

1. *Null-effect*: none of the conflict operations has any final effect on the target object.
2. *Single-operation-effect*: only one operation has a final effect on the target object.
3. *All-operations-effect*: all operations have final effects on the target objects.

In [11] and [15], a *Multi-Version Single-Display* (MVSD) technique has been devised to achieve the *all-operations-effect*: multiple versions of the common target objects are created to accommodate the effects of all conflict operations, but only one version is displayed at the user interface. Users are allowed to

choose to display any version at a time by using the system undo facility or a multi-version management tool [15].

The multi-versioning technique is capable of preserving all users' work even in the face of conflict; the single-display strategy matches the single-user interface of existing graphic editing applications, and lends itself to integration with OT [15].

3.3 Combined Effects for Conflict and Compatible Operations

Based on the conflict/compatible relations given in Table I and the MVSD technique, we specify the combined effects among the five operations: *CreateObj, DeleteObj, ChangeAtt, Group*, and *Ungroup,* in this subsection.

According to Table I, a *CreateObj* operation is always compatible with all operations, including another *CreateObj* operation because the object to be created cannot be targeted by another concurrent operation.

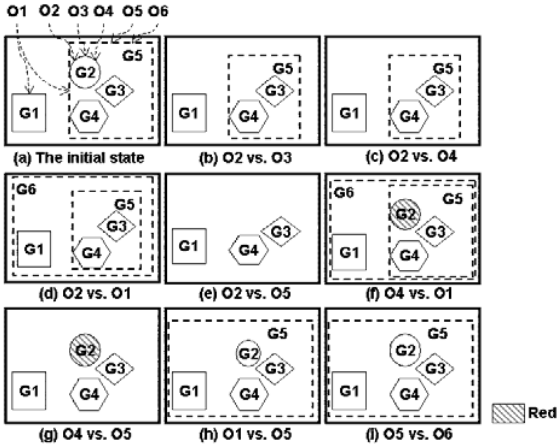


Figure 3. Combined effects between graphics editing operations. O1 = Group (G1, G5), O2 = DeleteObj (G2), O3 = DeleteObj (G2), O4 = ChangeAtt (G2, FillColor, red), O5 = Ungroup (G5), O6 = Ungroup (G5).

A *DeleteObj* operation is always compatible with all other operations as well because the effect of a *DeleteObj* operation can be combined with the effect of any other concurrent operation targeting the same object.

1. The combined effect with another *DeleteObj* operation is the deletion of the target object (Figure 3-(b)). Their effects have been combined in the sense that the deleted object can be recovered only after undoing both operations [15].
2. The combined effect with a *ChangeAtt* operation is the change of the attribute and the deletion of the target object (Figure 3-(c)).
3. The combined effect with a *Group* operation is the creation of a group-object containing all member objects targeted by the *Group* operation, except the member object targeted by the *DeleteObj* operation (Figure 3-(d)).
4. The combined effect with an *Ungroup* operation is the unpacking of the member objects in the group-object targeted by the *Ungroup* operation and the deletion of the member object targeted by the *DeleteObj* (Figure 3-(e)).

A *ChangeAtt* operation may conflict with another *ChangeAtt* operation under the condition specified in *Definition 1*; but it is always compatible with other operations because the effect of a *ChangeAtt* operation can be combined with the effect of any other concurrent operation targeting the same object.

1. The combined effect with a *DeleteObj* operation is illustrated in Figure 3-(c).
2. The combined effect with a *Group* operation is the creation of a group-object containing all target member objects, and the change of the attribute of one member object targeted by the *ChangeAtt* operation (Figure 3-(f))
3. The combined effect with an *Ungroup* operation is the unpacking of all member objects inside the target group-object, and the change of attribute of the member object targeted by the *ChangeAtt* operation (Figure 3-(g)).

Examples for illustrating the combined MVSD effects of conflicting *ChangeAtt* operations targeting common *non-group* objects can be found in [11] and [15]. An example for the combined MVSD effects of conflicting *ChangeAtt* operations targeting group-objects shall be given in Section 4.2.

A *Group* operation may conflict with another concurrent *Group* operation if they target common objects; but it is always compatible with other operations because the effect of a *Group* operation can be combined with the effect of any other concurrent operation targeting the same object.

1. The combined effect with a *DeleteObj* or a *ChangeAtt* operation has been illustrated in Figure 3-(d) and Figure 3-(f), respectively.
2. The combined effect with an *Ungroup* operation is the creation of a group-object containing all member objects targeted by the *Group* operation and the unpacking of the group-object (a member object targeted by the *Group* operation as well) targeted by the *Ungroup* operation (Figure 3-(h)).

An example for illustrating the combined MVSD effects of two conflict *Group* operations is given in Figure 4. Initially, the document contains five objects: G1, G2, …, G5, and suppose two operations *O1* = *Group*(G1, G2, G3) and *O2* = *Group*(G3, G4, G5) are generated concurrently, as shown in Figure 4-(a). Since *O1* and *O2* target a common object G3, they conflict with each other. To achieve the MVSD effect, two versions $G3_{O1}$ and $G3_{O2}$ should be created to accommodate the effects of both *O1* and *O2*, but only $G3_{O1}$ is displayed in the group-object created by *O1* (Figure 4-(b)), provided that *O1* has a higher priority than *O2* [15]. The version $G3_{O2}$ is maintained internally in the group-object created by *O2* but is invisible at the user interface due to the single-display strategy. However, after *O1* is undone, $G3_{O2}$ shall become visible as shown in Figure 4-(c).

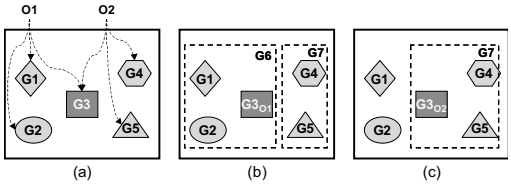


Figure 4. An example for illustrating the combined MVSD effect of two conflict Group operations.

An *Ungroup* operation is always compatible with other operations for the reasons explained above and illustrated in

Figure 3-(e), Figure 3-(g) and Figure 3-(h) respectively. The combined effect of two concurrent *Ungroup* operations targeting the same group-object is the unpacking of the target group-object (Figure 3-(i)). Both *Ungroup* operations have been combined in the sense that the group-object can be recovered only after undoing both operations [15].

4. Supporting object grouping in the TA framework

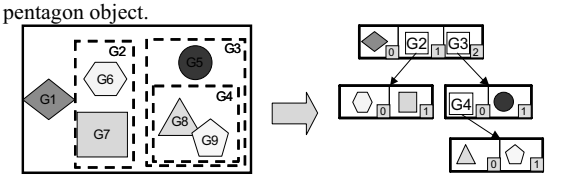
In this section, we shall discuss the technical issues and solutions involved in supporting object grouping by means of OT in the TA framework.

4.1 The Group Objects Address Model

The first issue is how to map graphics objects, particularly group-objects, into an object address model that is compatible with that of OT as shown in Figure 1.

A wide range of graphics editing applications have provided varieties of mechanisms (in their APIs) for mapping any graphic objects, including group-objects, into a tree of linear addressing domains [16]. To illustrate this address mapping, consider the following example: Figure 5-(a) shows a graphic document when viewed from the user interface; and Figure 5-(b) shows the mapping of the graphic objects in this document to a tree of linear addressing domains when viewed from the API. In this example, the top three objects (G1, G2, and G3) are mapped into the top-level linear addressing domain in the tree; the member objects in the two group-objects G2 and G3 are mapped into two second-level addressing domains, respectively; and the member objects in group-object G4 are further mapped into a third-level addressing domain. As shown in this example, member objects of a group-object forms a separate linear addressing domain; a group-object (e.g. G4) can be a member object of a higher level group-object (e.g. G2), allowing multiple levels of object grouping.

Under the address model in Figure 5-(b), any graphic object can be accessed with the vector address used in OT (see Figure 1). For example, the address of the pentagon can be expressed as a vector address [2, 0, 1], where "2" refers to the group-object G3, "0" refers to the group-object G4; and "1" refers to the pentagon object.



(a) The user interface representation (b) The address model in the API

Figure 5. The group objects address model.

4.2 Basic AOs targeting Group-Objects

In the TA framework (see Section 2.2), the user's interactions with the single-user application are intercepted and expressed as *Adapted Operations* (AO). For the three basic operations, we have three corresponding basic AOs: *CreateObjAO, DeleteObjAO,* and *ChangeAttAO.* Effects of these basic AOs in the group-object address model can be fully captured by POs, so the built-in mechanisms of OT are capable of resolving conflicts among basic AOs without any additional mechanisms at the AO level.

A example of resolving conflicts among *ChangeAttAOs* targeting group-objects is shown in Figure 6. From the initial document state (Figure 6-(a)), three operations are generated concurrently: O1 = *ChangeAttAO*([0, 0, 0], FillColor, red) to change the color of non-group object G1 into red, O2 = *ChangeAttAO*([0, 0], FillColor, green) to change the color of group-object G5 to green, and O3 = *ChangeAttAO*([0], FillColor, blue) to change group-object G6 to blue. According to the conflict definition, these three AOs conflict. Assume their priority relation is O1 > O2 > O3.

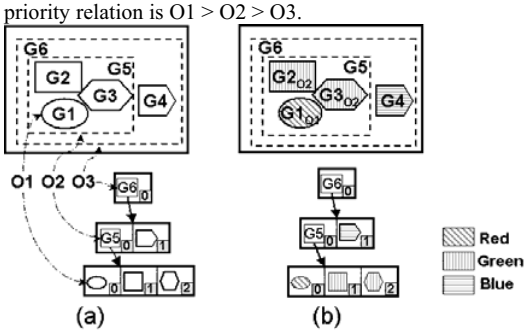


Figure 6. A scenario of three conflict ChangeAttAOs.

The conflicts among these AOs can be detected in OT from their common PO types (all are type *Update)*, target attribute types (all are FillColor), and overlapping addresses, (O3.addr is the prefix of O2/O1.addr, and O2.addr is the prefix of O1.addr). These conflicts are solved with the conflict resolution algorithm for the *Update* PO [15] and the combined MVSD effects, shown in Figure 6-(b), is achieved. In this result, multiple versions for objects targeted by conflict AOs are created, but only the versions created by AOs with the highest priorities (e.g. $G1_{O1}$, $G2_{O2}$ and $G3_{O2}$) are displayed.

4.3 Grouping AO Representation

For object grouping, we have two grouping AOs, named as *GroupAO* and *UngroupAO*, respectively. To determine the representation of these grouping AOs, it is necessary to analyze their effects on both the real objects (visible from the user interface) and the object address model (visible from the API).

As illustrated in Figure 7, the effect of a *GroupAO* on the real objects is to pack multiple target objects into a single group-object; and its effects on the internal addressing model include: (1) inserting a group-object in the current addressing domain (at the position before the first target object); and (2) moving all target objects into a lower level addressing domain (linked to the group-object). In moving these target objects, their original relative sequence relationships are preserved (see Figure 7-(b)).

The effect of an *UngroupAO* on the real objects is to unpack the target group-object into multiple member objects; and its effects on the address model include: (1) moving all member objects to the position of the target group-object in the higher

level addressing domain; and (2) deleting the target group-object (see Figure 7-(c)).

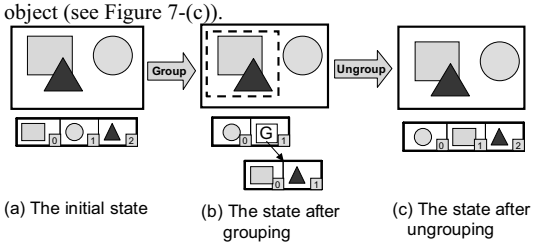


Figure 7. Effects of GroupAO and UngroupAO.

It should be pointed out that after executing the *UngroupAO* operation, the document state returns to the previous state before the execution of the *GroupAO* operation at the user interface; but the internal addresses of these objects are not restored, as can be seen by comparing Figure 7-(a) and (c). These object grouping effects are supported by the APIs of all existing single-users applications we have investigated, including MS Word, MS PowerPoint and OpenOffice Presentation.

To facilitate grouping AOs adaptation, their representations must capture their effects on both the data objects (needed for replaying their effects in *AO-API Adaptation* (see Figure 2)), and on the object addressing space (needed for OT-processing in *AO-PO Adaptation* (see Figure 2)). Since both *GroupAO* and *UngroupAO* have the effect of moving existing objects between different addressing domains, we introduce a new operation, named *MoveAO,* to represent this effect. The *MoveAO* can be represented as follows:

- *MoveAO(from, to, obj)* denotes the effects of deleting the object *obj* at the address *from* and inserting the same *obj* at the address *to*.

Based on the basic AOs and *MoveAO*, the two grouping AOs can be represented as follows:

1. *GroupAO(CreateObjAO(addr, go), MoveAO(from-1, to-1, obj-1)*, …, *MoveAO(from-n, to-n, obj-n))* denotes the effects of creating a group-object *go* at address *addr* and moving the target member objects *obj-1, ..., obj-n* from addresses *from-1, ..., from-n*, to new addresses *to-1, ..., to-n* at a lower level addressing domain.
2. *UngroupAO(DeleteObjAO(addr, go), MoveAO(from-1, to-1, obj-1)*, …, *MoveAO(from-n, to-n, obj-n))* denotes the effects of deleting the target group-object *go* at address *addr* and moving the member objects *obj-1, ..., obj-n* from addresses *from-1, ..., from-n*, to new addresses *to-1, ..., to-n* at a higher level addressing domain.

It should be stressed that the object addresses used in all AOs are positional references in the tree of linear addressing domains (see Figure 1), rather than the visual locations of the data objects at the user interface.

4.4 Grouping AO Translation

OT works on an operation model that includes only three POs (see Section 2.1). For conflict resolution and consistency maintenance by OT, all AOs, including grouping AOs, must be translated into suitable POs according to their effects on the OT-related address model. Translation of the basic AOs is straightforward: a *CreateObjAO* has the effect of inserting an object in the address model, so it can be translated into an *Insert* PO; a *DeleteObjAO* has the effect of deleting an object from the address model, so it can be translated into a *Delete* PO; a *ChangeAttAO* has the effect of changing an attribute of an object in the address model, so it can be translated into an *Update* PO.

On the other hand, *GroupAO* and *UngroupAO* are compound AOs in the sense that they cannot be translated into a single PO. The translation of a compound AO consists of translating each composing AO into a list of POs.

Definition 3. Translation Rules for Grouping AOs. For each composing AO in a grouping AO, it is translated as follows:

1. if the composing AO is a basic AO: *CreateObjAO/ DeleteObjAO/ChangeAttAO,* then it is translated into a single *PO: Insert/Delete/Update PO;*
2. if the composing AO is *MoveAO*, then it is translated into a pair of POs: *Delete* and *Insert*, where the two POs must refer to the same object (which is different from a pair of independent *Delete* and *Insert*).

Let *GroupAO-POList* denote the translated PO list for *GroupAO*, *UnGroupAO-POList* denote the translated PO list for *UngroupAO*. Based on the translation rules in *Definition 3*, we have:

1. *GroupAO-POList = [Insert(go-addr, go-ref), Delete(from-1, moref-1), Insert(to-1, moref-1), ..., Delete(from-n, moref-n), Insert(to-n, moref-n))*.
2. *UngroupAO-POList=[Delete(go-addr, go-ref), Delete(from-1, moref-1), Insert(to-1, moref-1), ..., Delete(from-n, moref-n), Insert(to-n, moref-n))*.

It should be stressed that the translated PO list captures only part of the grouping AO effects (including the timestamps for detecting concurrency [14] and priorities), which are needed for generic OT processing. Additional application-specific mechanisms are needed to detect and resolve operation conflict at the AO level, which are discussed in the following subsections.

4.5 Grouping AO Conflict Detection

Based on the AO representation and translation schemes discussed in Sections 4.3 and 4.4, conflicts among basic AOs can be fully detected and resolved by the mechanisms built in the generic OT technique [15]. However, detection of conflicts among *GroupAO*s requires the knowledge of operation type *Group* (see *Definition 1*), which is unknown to the OT component in GCE (see Figure 2). Therefore, conflict detection in the presence of grouping AOs requires additional mechanisms at the AO level.

According to *Definition 1*, a pair of *GroupAOs* may conflict under three conditions: (1) they are concurrent; (2) they have overlapping target objects; and (3) they have the same operation type *GroupAO*. OT is able to detect the first two conditions by examining the POs translated from *GroupAOs*, but the third condition must be checked at the AO level. To facilitate the check of the third condition and to propagate the concurrency and overlapping conditions result from the PO level to the AO level, we have established bi-directional references between each AO and its translated POs. A routine *GetAO(PO)* is provided to get the AO associated with the PO. Moreover, the underlying OT functions have been extended as follows: when a *PO1* is transformed against a concurrent *PO2* and found to have

overlapping target objects with *PO2*, this finding and *PO2*'s reference to its associated AO must be recorded in the transformed *PO1*. At the AO level, a routine *POConcurrentAndOverlapping(PO1)* is provided to check whether *PO1* has been found to be concurrent and overlapping with another operation, and another routine *GetCOAO(PO1)* is provided to get the AO associated with *PO2*. Based on the above extensions, we are able to determine whether a *GroupAO* is in conflict with another *GroupAO* by invoking the *GAOConflictDetection*( ) routine defined in Figure 8.

```
GAOConflictDetection(TPO)
{
   if(POConcurrentAndOverlapping (TPO) == true)
   {
      if(GetAO(TPO).type == GetCOAO(TPO).type == Group)
         return true;
   }
   return false;
}
```

Figure 8. The routines for detecting grouping AO conflicts.

4.6 Conflict Resolution and Combined Effects

4.6.1. The Need for AO-level mechanisms. OT is able to resolve conflicts among basic AOs, but additional mechanisms at the AO level are needed to resolve conflicts among *GroupAOs*. This is because resolving *GroupAO* conflicts requires semantic knowledge of the *GroupAO* and its representation, which are not captured by individual POs and hence unknown to the OT component in GCE. For the same reason, to achieve combined effects among compatible AOs in the presence of grouping AOs, additional mechanisms at the AO level are also needed. In other words, resolving conflicts among conflict operations and achieving the combined effects among compatible operations require the interaction and collaboration between the underlying generic OT component and the *AO-PO Adaptation* module in the TA framework (Figure 2).

An overall picture of the responsibility distribution between these two components is shown in Table I (see Section 3.1): the non-shaded cells indicate the sole responsibility areas of the generic OT component for resolving conflicts and achieving the defined combined effects among basic AOs; the shaded cells correspond to joint responsibility areas of OT plus additional AO-level mechanisms (in the *AO-PO Adaptation* module) for resolving conflict and achieving combined effects in the presence of grouping *AOs*.

The rest of this section shall focus on mechanisms for resolving conflicts and achieving combined effects for the shaded cells in Table I.

In the following discussion, we shall use the following auxiliary functions: (1) *GetMove*(*POx*) returns the composing *MoveAO* from which the PO *POx* is translated; and (2) *GetCOMove*(*POx*) returns the composing *MoveAO* of the grouping AO whose reference is recorded in the PO *POx*. Implementation of these functions is straightforward based on the AO-PO association and AO reference recorded in a transformed PO. Furthermore, we use the term *Common Target MoveAO* (CT-MoveAO) to mean a composing *MoveAO* of a grouping AO that moves a common target object targeted by another concurrent AO.

4.6.2. Resolving GroupAO Conflicts. According to the MVSD combined effect defined in Section 3.3, the conflict between two *GroupAOs* is resolved based on their priorities. Given two conflict *GroupAOs*: *O1* with a higher priority and *O2* with a lower priority, their common target objects should be packed in the group-object created by *O1* and excluded from the group-object created by *O2*.

In the *GroupAO* representation, the effects of moving target objects are represented by composing *MoveAOs*. Therefore, for a pair of conflicting *GroupAOs O1* and *O2*, there must be a *CT-MoveAO* in each of them, which target a common target object. Based on this observation, the strategy of resolving the conflict between *O1* and *O2* is as follows:

1. if the *O1* is executed after *O2*, the *from* parameter of the *CT-MoveAO* of *O1* should be set to the *to* parameter of the *CT-MoveAO* of *O2*, so that the common target object shall be moved to the group-object created by O1.
2. if *O2* is executed after *O1*, the *CT-MoveAO* of *O2* should be cancelled so that the common target object is excluded from the group-object created by *O2*.

Based on the above strategy, the routine *GAOConflictResolution(TPO)* is defined (Figure 9) for resolving the conflict between the *GroupAO* (obtained by calling *GetAO*) from which the *TPO* was translated and the *GroupAO* (obtained by calling *GetCOAO*) with which *TPO* was associated due to concurrency and overlapping relationship.

```
GAOConflictResolution(TPO)
{
   if(GetAO(TPO).priority > GetCOAO(TPO).priority)
      GetMove(TPO).from = GetCOMove(TPO).to;
   else
      GetMove(TPO).cancelled = true;
}
```

Figure 9. The routine for resolving conflicts among GroupAOs.

Based on the MVSD effect, our conflict resolution approach also supports selectively displaying versions that are hidden by default. Assume that between the two conflict *GroupAOs O1* and *O2*, *O1* has a higher priority than *O2*. According to the MVSD effect, two versions of the common target object are created, but only the version created by O1 is displayed. To display the version created by O2, a simple strategy is to undo *O1*. The disadvantage of this strategy is that all *O1's* object-packing effects are unnecessarily discarded, including those non-common objects that are not targeted by *O2*. To preserve *O1's* effects to the maximum extent, a better strategy is to partially undo the composing *CT-MoveAO* of *O1*. From the adjustment to this *MoveAO* while resolving the conflict between *O1* and *O2*, it is clear that the effect of this undo is only to move the common target object from *O1's* group-object into *O2's*, while all other member objects in *O1's* group-object are intact. A detailed discussion on this partial-undo based version selection scheme is beyond the scope and space limitation of this paper. The reader is referred to [15] for a detailed discussion on a full-undo based version selection scheme.

4.6.3. Achieving Combined Effects for Compatible Operations in the Presence of GroupAOs. According to the combined effects of concurrent and compatible operations

defined in Section 3.3 (see Figure 3), their effects should be accommodated on the common target object at the same time.

Here we shall focus in scenarios in which two concurrent and overlapping compatible AOs are involved and at least one of them is a grouping AO. Given a pair of AOs, *O1* and *O2* involved in such a scenario, suppose *O1* is executed after *O2*. When *O1* is executed, its parameters need to be adjusted according to the changes caused by *O2* to achieve the combined effect. Next, we shall discuss adjustment strategies for different AO type combinations.

In the routines discussed in this section, the input parameter *TPO* is the transformed PO of the currently processed AO (i.e. O1). With *TPO*, *O1* can be obtained by calling *GetAO; O2* can be obtained by calling *GetCOAO*; the *CT-MoveAO* of *O1* can be obtained by calling *GetMove* if *O1* is a grouping AO; and the *CT-MoveAO* of *O2* can be obtained by calling *GetCOMove* if *O2* is a grouping AO.

Consider the scenario in which *O1* is a *GroupAO* and *O2* is a *DeleteObjAO*. When *O1* is executed, the common target object has been deleted by *O2*. Therefore, this object should be excluded from the group-object created by *O1*. From the *GroupAO* representation, we know that the effect of moving the common target object is represented by the *CT-MoveAO* of *O1*, so our strategy for this scenario is to cancel the *CT-MoveAO* of *O1*. This strategy also applies to the AO combinations of *UngroupAO* versus *DeleteObjAO* [1](the *DeleteObjAO* targets a member object of the *UngroupAO's* target group-object) and *UngroupAO* versus *UngroupAO*.

On the other hand, if *O1* is a *DeleteObjAO* and *O2* is a *GroupAO*, when *O1* is executed, its target object has been moved into the group-object created by *O1*. Based on the *GroupAO* representation, we know that the current address of the common target object is indicated by the *to* parameter of *O2's CT-MoveAO*, so our strategy for this scenario is to set *O1's* address to the *to* parameter of *O2's CT-MoveAO*. This strategy also applies to AO combinations *ChangeAttAO/DeleteObjAO* versus *UngroupAO* (the *ChangeAttAO/DeleteObjAO* targets a member object of the *UngroupAO's* target group-object), *DeleteObjAO* versus *GroupAO*, and *UngroupAO* versus *GroupAO*.

Based on the above strategies, the routine for achieving combined effects for concurrent and overlapping *GroupAO* and *DeleteObjAO* is shown in Figure 10.

```
CE_GroupDeleteObj(TPO)
{
   if(GetAO(TPO).type == GroupAO)
      GetMove(TPO).cancelled = true;
   else
      GetAO(TPO).addr = GetCOMove(TPO).to;
}
```

Figure 10. The routine for achieving combined effects for GroupAO and DeleteObjAO.

Consider the scenario in which *O1* is a *ChangeAttAO*, *O2* is an *UngroupAO* and they both target the same group-object. When *O1* is executed, the common target group-object has been unpacked into a continuous range of multiple objects by *O2* (see Figure 7-(c)). From the *UngroupAO* representation, we know that the address and length of the unpacked object range are indicated by *O2's* composing *MoveAOs*. Therefore, our strategy for this scenario is to set *O1's* effect range (i.e. address and length) to cover all unpacked objects. This strategy also applies to AO combinations *DeleteObjAO* versus *UngroupAO*(the *DeleteObjAO* targets the same group-object as the *UngroupAO*) and *GroupAO* versus *UngroupAO*.

In the scenario in which *O1* is an *UngroupAO* and *O2* is a *ChangeAttAO*, when *O1* is executed, *O2* has applied its effect on all member objects of the target group-object. To make sure that after ungrouping, all the unpacked objects will still have *O2's* effect, our strategy is to apply *O2's* effect to data objects of all *O1's* composing *MoveAOs*.

```
CE_UngroupChangeAttGO(TPO)
{
   if(GetAO(TPO).type == ChangeAttAO)
      SetEffectRange(GetAO(TPO), GetCOAO(TPO));
   else
   {
      for(i = 0; i < GetAO(TPO).MoveAOList.count; i ++)
         ApplyChangeAtt(GetAO(TPO).MoveAOList[i].obj, GetCOAO(TPO));
   }
}
```

Figure 11. The routine for achieving combined effects for UngroupAO and ChangeAttAO (targeting the group-object).

Based on the above strategies, the routine for achieving combined effects for concurrent *UngroupAO* and *ChangeAttAO* targeting the same group-object is shown in Figure 11.

4.7 Grouping AO-PO Adaptation Algorithm

With the routines discussed above, the *AO-PO Adaptation* in the TA framework can be extended to support grouping AOs, as shown in Figure 12.

First, the input AO is translated into a series of POs saved in a PO list. Then, each PO in the list is processed as follows. The PO is first transformed in OT. Then, if this AO involves in a *GroupAO* conflict, the conflict resolution routine is invoked. Otherwise, if this AO is overlapping with another concurrent compatible AO and at least one of them is a grouping AO, the *CompatibleGAOCombinedEffects* routine is invoked to apply AO level mechanisms for achieving combined effects for compatible AOs. In the *CompatibleGAOCombinedEffects* routine, suitable routines discussed in Section 0 are invoked according to AO type combinations.

```
GAO-POAdaptation(AO)
{
   POList = TranslateAO(AO);
   for(i = 0; i <POList.count; i++)
   {
      TransformPO(POList[i]);
      if(GAOConflictDetection(POList[i]) == true)
         GAOConflictResolution(POList[i]);
      else if (POConcurrentAndOverlapping(POList[i]) == true &&
            IncludingGroupingAO(GetAO(POList[i]), GetCOAO(POList[i])) == true)
         CompatibleGAOCombinedEffects(POList[i]);
   }
}
```

Figure 12. The routines for adapting AOs in the presence of grouping AOs.

5. Comparison to Related Work

[1] In this pair, the former AO is the AO current being processed (i.e. *O1*), and the latter AO is the one concurrent and overlapping with the former (i.e. *O2*).

The *CoGroup* technique reported in this paper is the first collaborative object grouping technique based on the OT technique and designed in the TA framework. This work made important extensions to our prior work on OT and TA in order to support collaborative object grouping. Particularly, this work contributes a new definition of conflict/compatible relations among graphic editing operations in the presence of object grouping operations, new definitions of desirable combined effects among a mixture of basic and grouping operations to maximize the natural combination of compatible operations and to preserve all users work in the face of conflict, and novel data and operation adaptation techniques to bridge the gap between grouping operations and OT-supported primitive operations in the TA framework. These extensions are essential to apply OT and TA to a wider range of commercial off-the-shelf editing systems, particularly CAD/CASE applications.

To the best of our knowledge, the operation serialization technique reported in [5] is the only prior work on collaborative object grouping in graphic editing systems. Both the *CoGroup* work in this paper and the work in [5] address similar issues involved in conflict resolution for a similar collection of graphics editing operations, but these two work are very different in their approaches to conflict definitions, combined effects among conflicting/compatible operations, and techniques for conflict resolution.

The notion of conflict in *CoGroup* is based on the conditions that operations are concurrent, target common objects, and cannot be accommodated in the common target objects. Under this conflict definition, conflict may occur only between *ChangeAtt* operations or between *Group* operations, and the relations among all other operations are compatible (as shown in Table I). Operation conflicts are resolved by an *all-operations-effect* technique: multiple versions of the common target objects are created to preserve the effects of all operations, but one version at a time is displayed at the user interface (the *MVSD* technique). *CoGroup* is based on and extends OT for conflict resolution and consistency maintenance.

The notion of conflict in [5] is based on the conditions that operations are concurrent and do not commute. Under this conflict definition, conflict may occur not only between *ChangeAtt* operations and between *Group* operations, as in the *CoGroup* technique (see Table I), but also among other operations, as shown Table II (in which the *ChangeAtt* operation represents the *setColor, SetBckColour, setZ, SetText, translate, scale* operations in [5]).

Table II. Conflict relation triangle of five operation types in the prior work [5].

	CreateObj	***DeleteObj***	***ChangeAtt***	***Group***	***Ungroup***
CreateObj	⊙	⊙	⊙	⊙	⊙
DeleteObj		⊗/⊙	⊗/⊙	⊗/⊙	⊗/⊙
ChangeAtt			⊗/⊙	⊗/⊙	⊗/⊙
Group				⊗/⊙	⊗/⊙
Ungroup					⊗/⊙

For the purpose of resolving operation conflict, two types of conflict are further distinguished in [5]: *real conflicts* are those which can be resolved by preserving the effect of one of the conflict operations (or none of them); and *resolvable conflicts* are those which can be resolved by combining partial effects of conflict operations. Regardless whether the conflict is real or resolvable, conflict resolution is based on *operation serialization*, which achieves the defined effects either by using operation-specific ordering rules (specified in [5]) for resolvable conflicts, or by using any priority scheme for real conflicts. Serialization is essentially a *single-operation-effect* or *null-effect* conflict resolution technique [11].

It is well known that the combined effects achievable by an *all-operations-effect* technique cover all combined effects achievable by a *single-operation-effect* technique, but the inverse is not true [11]. Furthermore, some combined effects among conflict *Group* operations achievable by *CoGroup* are not achievable by the serialization work in [5]. For example, when two concurrent *Group* operations target some common and non-common objects, they are regarded as conflict operations in both *CoGroup* and [5] (a *real conflict*). The combined effects in *CoGroup* is following: both *Group* operations shall succeed in creating their result group-objects; both group-objects contain their non-common target objects, but only one of them has the common target objects displayed (see Figure 4). However, the combined effects in [5] is the following: one of the two *Group* operations shall win and create the group-object containing all target objects, but the other one shall lose completely and has no any effect (not even the effect of grouping the non-common target objects).

In [5], achieving the partially combined effects for some resolvable conflicts is the main motivation for disqualifying OT from being applied for this purpose and for devising the new operation serialization technique. As shown in the example in Figure 6, however, the partially combined effect in [5] can be achieved by using the generic OT technique without additional application-level support, and more comprehensive MVSD combined effects can be achieved by extending OT with the application-level adaptation. A major problem with operation serialization is its undoing and redoing conflict operations when they are executed out of the correct conflict resolution order, which may cause potential interface disruption (when the undo/redo effects are visible at the user interface) and major performance overheads. It should be pointed out that the undo/redo involved in operation serialization is different from the collaborative undo capability in OT: the former is initiated by the internal system out of the necessity for resolving conflict among grouping operations, but the latter is initiated by the external user for the purpose of eliminating the effect of error grouping operations [13].

Last but not least, *CoGroup* is designed in the TA framework so it can be applied to a wide range of commercial off-the-shelf graphics editing systems (and this generality has been tested by its successful application in CoWord[2] and CoPowerPoint[3], whereas the work in [5] was designed in the context of a collaboration-aware graphics editing system and its applicability to collaboration-transparent applications is unknown.

It is worth pointing out that there exist other alternative approaches to conflict resolution based on locking (e.g. Ensemble [9] and GroupDraw [4]) or different kinds of serialization (e.g. GroupDesign [7] and LICRA [6]), but none of them addressed the issues related to collaborative object

[2] CoWord Demo. http://reduce.qpsf.edu.au/coword.

[3] CoPowerPoint Demo. http://reduce.qpsf.edu.au/copowerpoint.

grouping. The reader is referred to [11] and [15] for detailed comparisons between the multi-versioning approach, on which *CoGroup* is based, and these alternative approaches.

6. Conclusions and Future Work

In this paper, we have contributed a novel technique, called *CoGroup*, to supporting collaborative object grouping in graphics editing systems. The *CoGroup* technique is the first collaborative object grouping technique based on the OT technique in the TA framework. Major technical contributions of this work include new definitions of conflict/compatible relations among graphics editing operations in the presence of object grouping operations, new definitions of desirable combined effects among basic and grouping operations to maximize the natural combination of compatible operations and to preserve all users work in the face of conflict, and novel data and operation adaptation techniques to bridge the gap between grouping operations and OT-supported primitive operations in the TA framework. The collaborative object group capability provided by *CoGroup* is essential to expand the application scope of OT and TA to a wider range of commercial off-the-shelf editing systems, particularly CAD/CASE applications, which are the new targets of our research plan.

7. Reference

[1] J. Begole, M. Rosson and C. Shaffer. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *ACM Transactions on Computer Human Interaction*, 6(2), 1999, pp. 95 – 132.

[2] A. Davis, C. Sun and J. Lu. Generalizing operational transformation to the standard general markup language. *In Proceedings of the ACM Conference on Computer-Supported Cooperative Work*. November 2002, pp. 58 – 67.

[3] C. A. Ellis and S. J. Gibbs, Concurrency control in groupware systems. *In Proceedings of ACM Conference on Management of Data*. May 1989, pp. 399 – 407.

[4] S. Greenberg and M. Roseman, and D. Webster. Issues and experiences designing and implementing two group drawing tools. *In Proceedings of the 25th Annual Hawaii International Conference on the System Sciences.* January 1992, pp. 139 – 250.

[5] C. Ignat and M. C. Norrie. Grouping in collaborative graphical editors. *In Proceedings of Computer Supported Cooperative Work*. November 2004, pp. 447 – 456.

[6] R. Kanawati. LICRA: A replicated-data management algorithm for distributed synchronous groupware application. *Parallel computing,* 22, 1997, pp. 1733 – 1746.

[7] A. Karsenty, C. Tronche, and M. Beaudouin-Lafon. Groupdesign: shared editing in a heterogeneous environment. *Usenix Journal of Computing Systems*, 6(2), 1993, pp. 167 – 195.

[8] D. Li and R. Li. Transparent sharing and interoperation of heterogeneous single-user applications. *In Proceedings of ACM Conference on Computer Supported Cooperative Work*. November 2002, pp. 246 – 255.

[9] R. E. Newman-Wolfe, M. Webb and M. Montes. Implicit locking in the ensemble concurrent object-oriented graphics editor. *In Proceedings of the Computer Supported Cooperative Work*. New York, 1992, pp. 265-272.

[10] M. Ressel, D. Nitshe-Ruhland and R. Gunzenbauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. *In Proceedings of ACM Conference on Computer Supported Cooperative Work*. November 1996, pp. 288 – 297

[11] C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transaction on Computer-Human Interaction*, 9(1), March 2002, pp. 1 – 41.

[12] C. Sun and C. A. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. *In Proceedings of the ACM Conference on Computer Supported Cooperative Work*. 1998, pp. 59 – 68.

[13] C. Sun. Undo as concurrent inverse in group editors. *ACM Transactions on Computer-Human Interaction*, 9(4), December 2002, pp. 309 – 361.

[14] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1), March 1998, pp. 63 – 108.

[15] D. Sun, S. Xia, C. Sun, and D. Chen. Operational transformation for collaborative word processing. *In Proceedings of ACM Conference on Computer Supported Cooperative Work*. November 2004, pp. 437 – 446.

[16] S. Xia, C. Sun, D. Sun, H. Shen, and D. Chen. Leveraging single-user applications for multi-user collaboration: the coword approach. *In Proceedings of ACM Conference on Computer Supported Cooperative Work*. November 2004, pp. 162 – 171.