A Multi-version Approach to Conflict Resolution in Distributed Groupware Systems

Chengzheng Sun and David Chen
School of Computing and Information Technology
Griffith University
Brisbane, Qld 4111, Australia
Email: {C.Sun, D.Chen}@cit.gu.edu.au
URL: http://www.cit.gu.edu.au/∼scz

Abstract – *Groupware systems are a special class of distributed computing systems which support human-computer-human interaction. Real-time collaborative graphics editors allow a group of users to view and edit the same graphics document at the same time from geographically dispersed sites connected by communication networks. Resolving conflict access to shared objects is one of the core issues in the design of this type of systems. This paper proposes a novel distributed multi-version approach to conflict resolution. This approach aims to preserve the work concurrently produced by multiple users in the face of conflicts, and to minimize the number of object versions for accommodating combined effects of conflicting and compatible operations. Major technical contributions of this work include a formal specification of a unique combined effect for any group of conflicting and compatible operations, a distributed algorithm for incremental creation of multiple object versions, and a consistent object identification scheme for multi-version and multi-replica graphics editing systems. All algorithms and schemes presented in this paper have been used in the GRACE (GRAphics Collaborative Editing) system implemented in Java.*

Keywords
Collaborative graphics editors, consistence maintenance, multiple object versions, real-time groupware systems, distributed computing.

I. Introduction

Groupware systems are a special class of distributed computing systems which support *human-computer-human* interaction [2, 4, 13]. A commonly used groupware system is the real-time collaborative editor which allows a group of users to view and edit the same document at the same time from geographically dispersed sites connected by communication networks. Collaborative editors are very useful facilities in advanced Computer-Supported Cooperative Work (CSCW) applications [1], such as electronic conference/meeting, collaborative CAD/CASE, and collaborative documentation systems.

The goal of our research is to investigate the principles and techniques underlying the construction of collaborative editors with the following three major characteristics [13, 18]. (1) *Real-time:* The response to local user actions should be quick (without noticeable delay) and the latency for remote user actions should be low. The key performance parameter here is the response time observable by the user, rather than the number of operations per second as in non-interactive application systems. (2) *Distributed:* Collaborating users may reside on different machines connected by the Internet with non-negligible and non-deterministic latency. While there is no limit on the bandwidth increase of the Internet using fiber optic communication technologies, the communication latency over an inter-continental connection cannot be reduced considerably below 100 milliseconds (the threshold value for user noticeable delay) due to the speed limit of electronic signals. It is the communication latency, rather than the bandwidth, which presents a major challenge to achieving high responsivness for Internet-based collaborative editing systems. (3) *Unconstrained:* Multiple users are allowed to concurrently and freely edit any parts of the document at any time, in order to facilitate free and natural information flow among collaborating users. The major challenge of supporting unconstrained collaborative editing is the management of the multiple streams of concurrent activities so that system consistency can be maintained in the face of conflicts.

Appeared in **Proceedings of the 20th IEEE International Conference on Distributed Computing Systems,** *pp. 316-325, April 10-14, 2000.*

The requirements for high responsiveness and for supporting unconstrained collaboration over the Internet have led us to adopt a *replicated architecture* for the storage of shared documents: the shared documents are replicated at the local storage of each participating site, so editing operations can be performed at local sites immediately and then propagated to remote sites. Because of concurrent generation of operations and non-negligible and non-deterministic communication latency of the Internet, there exist three major inconsistency problems associated with the replicated architecture [13]: (1) *divergence* - operations may arrive and be executed at different sites in different orders, resulting in different final documents at different sites; (2) *causality violation* – operations may arrive and be executed out of their natural cause-effect order, causing confusion to both the system and its users; and (3) *intention violation* - the *actual* effect of an operation at the time of its execution may be different from the *intended* effect of this operation at the time of its generation. To address these inconsistency problems systematically, a consistency model has been proposed in the context of the REDUCE (REal-time Distributed Unconstrained Cooperative Editing) project [13]. The REDUCE consistency model has been applied to the collaborative text editing domain for solving various challenging technical problems [14, 15, 16, 18]. In this paper, we will report new research findings in applying the REDUCE framework to the GRACE (GRAphics Collaborative Editing) project.

Collaborative graphics editing systems can be classified into two types: object-based and bitmap-based. This paper is confined to the issues associated with object-based collaborative graphics editing systems only. Graphic objects such as lines, rectangles, circles, etc., can be created and updated. Each object is represented by attributes such as type, size, position, color, group, etc.. A *Create* operation is used to create an object. After an object has been created, updating operations can be applied to change the attributes of that object. For example, a *Move* operation changes the position attribute of the target object. In a collaborative editing environment, operation conflict may occur when multiple concurrent operations try to update the same object in different ways. Resolving conflict accesses to shared objects is one of the core issues in the design of this type of systems and will be the focus of this paper.

The rest of this paper is organized as follows. In Section II, some

background information about the REDUCE framework is briefly discussed. In Section III, a multiple version strategy for conflict resolution is proposed, and the rules for determining combined effects of conflicting and compatible operations are derived. Then, a formal specification of a unique combined effect for any group of conflicting and compatible operations is presented in Section IV. A distributed algorithm for incremental creation of multiple object versions is described in Section V. A consistent object identification scheme for multi-version and multi-replica graphics editing systems is presented in Section VI. Our work is compared to related work in Section VII. Lastly, major results are summarized and further work is discussed in Section VIII.

II. Previous work

In this section, the basic concepts, definitions, and techniques adopted from our previous work are briefly described. For details, the reader is referred to [13].

A. A consistency model

Following Lamport [8], we define a causal (partial) ordering relation of operations in terms of their generation and execution sequences as follows.

Definition 1: Causal ordering relation "$\rightarrow$"
Given two operations O_1 and O_2, generated at sites i and j, then $O_1 \rightarrow O_2$, *iff*: (1) $i = j$ and the generation of O_1 *happened before* the generation of O_2, or (2) $i \neq j$ and the execution of O_1 at site j *happened before* the generation of O_2, or (3) there exists an operation O_x, such that $O_1 \rightarrow O_x$ and $O_x \rightarrow O_2$. □

Definition 2: Dependent and independent operations
Given any two operations O_1 and O_2. (1) O_2 is *dependent* on O_1 *iff* $O_1 \rightarrow O_2$. (2) O_1 and O_2 are *independent* (or *concurrent*), expressed as $O_1 \parallel O_2$, *iff* neither $O_1 \rightarrow O_2$, nor $O_2 \rightarrow O_1$. □

Definition 3: Intention of an operation
Given an operation O, the intention of O is the execution effect which could be achieved by applying O on the document state from which O was generated. □

Definition 4: A consistency model
A collaborative editing system is said to be consistent if it always maintains the following properties: (1) **Convergence**: when all sites have executed the same set of operations, the copies of the shared document at all sites are identical. (2) **Causality-preservation**: for any pair of operations O_1 and O_2, if $O_1 \rightarrow O_2$, then O_1 is executed before O_2 at all sites. (3) **Intention-preservation**: for any operation O, both the local and remote execution effects of O equal to O's intention, and if there exists an operation O_x such that $O_x \parallel O$, then the execution effect of O_x does not interfere with the execution effect of O, and vice versa. □

It should be highlighted that the consistency model imposes an execution order constraint only on dependent operations, but leaves it open for the execution order of independent operations as long as the convergence and intention-preservation properties are maintained. This feature of the consistency model lays the theoretical foundation for achieving good responsiveness by permitting local operations to be executed immediately after their generation. Moreover, the intention-preservation property makes a further promise to the users that their individual operations' effects can be protected against each other's interference. Finally, it should be pointed out that the three properties are *independent* in the sense that the maintenance of any two of them does not automatically ensure the other one [13, 14].

B. Concurrency control techniques

The consistency model specifies, on the one hand, what assurance a collaborative editing system promises to its users, and on the other hand, what properties the underlying concurrency control mechanisms must support. To capture the causal relationships among all operations in the system, a timestamping scheme based on *vector logical clock* can be used [13, 16]. Causality-preservation can be achieved by using either a distributed algorithm [13] or a central notification server [16]. Since causality is an issue without any relationship with the semantics of operations, causality-preservation techniques are generic and applicable to both text and graphics editors.

For supporting convergence and intention-preservation, however, different editing domains require different techniques. In the text editing domain, an optimistic concurrency control technique, called operational transformation has been devised [14]. In the GRACE project, convergence and intention-preserving techniques for the graphics editing domain have been investigated. Since achieving convergence is a relatively simple and independent issue, this paper will focus on the issues and results related to achieving intention-preservation only.

III. Operation conflicts and multiple versions

A. Conflict and compatible relations

In the graphics editing domain, concurrent operations may target the same object and may conflict with each other. For example, suppose user 1 generates operation $O_1 = Move(G, X)$ to move object G to position X, and user 2 concurrently generates operation $O_2 = Move(G, Y)$ to move G to position Y, where $X \neq Y$. Both operations will be executed at their local sites immediately to give a quick response, and then propagated to the other sites. Since O_1 and O_2 are moving the same object G to two different positions, it is impossible to accommodate their conflicting effects in the same target object. In general, two concurrent operations are in conflict if they are targeting the same object but changing the same attribute to different values.

To give a precise definition of operation conflict, the following notations are introduced: (1) $Tgt(O)$ denotes the target object of operation O; (2) $Att.Type(O)$ denotes the attribute type of O; and (3) $Att.Value(O)$ denotes the attribute value of O.

Definition 5: Conflict relation "$\otimes$"
Given two operations O_1 and O_2, they conflict with each other, expressed as $O_1 \otimes O_2$, *iff* (1) $O_1 \parallel O_2$; (2) $Tgt(O_1) = Tgt(O_2)$; (3) $Att.Type(O_1) = Att.Type(O_2)$; and (4) $Att.Value(O_1) \neq Att.Value(O_2)$. □

In contrast, if a pair of operations are not conflicting, then they are compatible, as defined below.

Definition 6: Compatible relation "$\odot$"
Given two operations O_1 and O_2, if they do not conflict with each other, they are compatible, expressed as $O_1 \odot O_2$. □

B. Accommodating all operation effects

For compatible operations, if they are targeting the same object, they can be applied to the same object. For conflicting operations, what combined effects could they have without violating their intentions?

One possible combined effect is the *null-effect*, which means none of the two conflicting operations has any final effect on the target object. This can be achieved by rejecting/undoing an operation when it is found to be conflicting with another operation, as shown in Fig 1. The final results at both sites are identical (empty). However, this null effect does not preserve the intentions of the two operations since none of the two operations has any effect at the remote site and the effect of one operation has been undone by another independent operation. The consequence of this intention violation is that whenever there is a conflict, the work concurrently done by involved users will be destroyed. This effect is highly undesirable in the collaborative working environment because users involved in a conflict are provided with no explicit information about what other users intended to do, and hence may not be able to take proper actions to resolve their conflict.

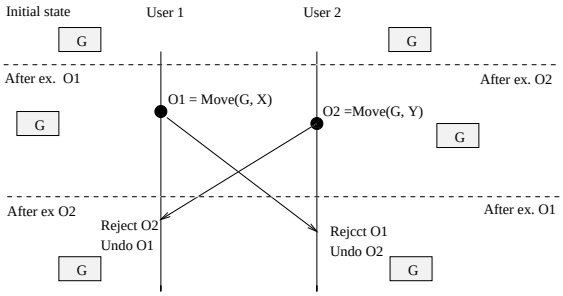

Fig. 1. The null-effect for conflicting operations

The second possible combined effect is the *single-operation-effect*, which is to retain the effect of only one operation, either O_1 or O_2. This can be achieve by enforcing a serialized effect among all operations. As shown in Fig. 2, when O_2 arrives at user 1, it moves G to position Y (effectively undoing O_1); when O_1 arrives at user 2, it is rejected. The final results at both users sites are identical. However, this single operation effect violates the intentions of both operations since one operation (O_1) has no effect at user 2, and the other operation (O_2) has changed the effect of an independent operation (O_1) at user 1. One consequence of this intention violation is that whenever there is a conflict, only one user's work can be preserved. Another consequence is that users are not ensured to see the effects of the same set of operations: e.g., user 1 sees the effects of both O_1 and O_2, but user 2 never sees the effect of O_1. Generally, when there are multiple conflicting operations, each user may see the effects of arbitrary number of operations, depending on the order in which operations arrive at each site. Therefore, when a conflict occurs, users may not see a consistent and explicit picture about what other users intended to do, and hence they may not be able to take proper actions to resolve their conflict.

To preserve all work concurrently produced by multiple users in the face of conflicts, we propose an *all-operations-effect* based on a *multiple versions* strategy: two versions of G, G_1 and G_2, will be created, with O_1 and O_2 being applied to G_1 and G_2, respectively. In this way, the effects of both operations are accommodated in two separate versions, as shown in Fig. 3.

This all-operations-effect preserves the intentions of both operations since the effects of O_1 and O_2 at their local sites are the same

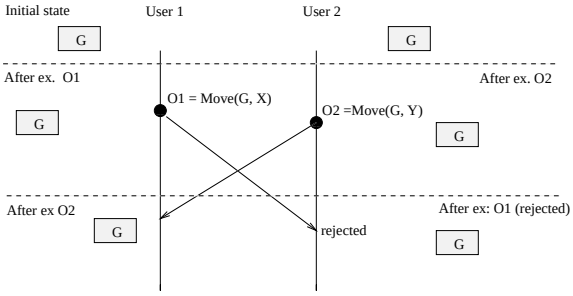

Fig. 2. Single operation effect for conflicting operations

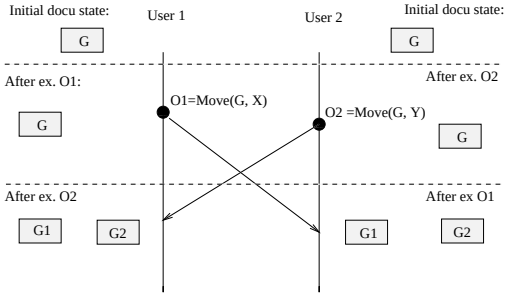

Fig. 3. All operations effect for conflicting operations

as their effects at the remote sites and they do not change the effects of each other. With this all-operations-effect, the system is able to ensure that the work produced by all users be always retained regardless whether there is a conflict or not. The only side effect of this approach is that the single version object may be converted to multiple versions if a conflict occurs. The system could notify the users that there is a conflict, e.g., by highlighting the multiple versions of the same object. Since all users are provided with a consistent and explicit picture about what other users intended to do, they could make better assessment of the situation and may decide to keep one of the versions or even all of them if that is desired.

It is worth pointing out that a similar all-operations-effect strategy has also been used in the collaborative text editing domain [13, 14]: when there are two concurrent *Insert* operations inserting two strings S_1 and S_2 at the same position, even if S_1 is a substring of S_2, both strings are maintained in the document (one after the other) rather than being merged into one. In general, we advocate a groupware design principle: In the face of a conflict, it is usually better to preserve and display all users' work to facilitate a user-decided solution to the conflict, rather than to destroy or hide users' work to impose a system-decided solution to the conflict. Because it is generally infeasible for the system to have the knowledge to properly resolve conflicts among concurrent users, conflicts are best resolved by collaborative users, with the system providing explicit information about other users' actions.

C. Combined effect rules

Given a group of N operations targeting the same object, if they are all mutually compatible with each other, then they can be applied to the original target object without creating new versions; and if they are all mutually conflicting with each other, then N versions can be created to accommodate each operation's effect in a separate version. However, if there is a mixture of compatible and conflicting operations in the group, it becomes non-trivial to determine how many versions to create and how to apply which operations to which versions. In the following discussion, the notation $G\{O_x\}$ will be used to represent an object G with the effect of O_x and G{ } represents its initial state.

To start with, consider a simple scenario with three operations: O_1, O_2, and O_3. Suppose they are targeting the same object G, and their mutual conflict relations are: $O_1 \otimes O_2$, $O_1 \otimes O_3$, and $O_2 \odot O_3$. What combined effects should these three operations have?

Since $O_1 \otimes O_2$, they must be separately applied to two versions $G\{O_1\}$ and $G\{O_2\}$ according to the multiple versions strategy. In general, we have the following combined effect rule:

Combined Effect Rule 1 (CER1): Given two operations O_1, and O_2 targeting object G. If $O_1 \otimes O_2$, they must be applied to different versions $G\{O_1\}$ and $G\{O_2\}$ made from G.

The question is: how to combine O_3's effect? One possibility is to make a separate version $G\{O_3\}$. The problem with this approach is that it unnecessarily creates two versions $G\{O_2\}$ and $G\{O_3\}$ for two compatible operations. To avoid unnecessary versions, we propose to combine two compatible operations O_2 and O_3 in a common version $G\{O_2, O_3\}$. In general, to minimize the number of versions for an object, the following combined effect rule is used to justify the creation of different versions.

Combined Effect Rule 2 (CER2): Given any two versions G_1 and G_2 made from the same object G, there must be at least one operation O_1 applied to G_1, and at least one operation O_2 applied to G_2, such that $O_1 \otimes O_2$.

Furthermore, consider another scenario with three operations: O_1, O_2, and O_3, targeting the same object G. Suppose their mutual conflict relations are: $O_1 \otimes O_2$, $O_1 \odot O_3$, and $O_2 \odot O_3$. Since $O_1 \otimes O_2$, two versions $G\{O_1\}$ and $G\{O_2\}$ need to be created according to **CER1**. The question is: which one of the two versions should O_3 be applied to?

One possibility is to combine O_3 with either O_1 (i.e., $G\{O_1, O_3\}$) or O_2 (i.e., $G\{O_2, O_3\}$), chosen by the system (randomly or by using their total ordering). This approach does not produce any unnecessary version (according to **CER2**), but may have an abnormal phenomenon at the user interface, as shown in Fig. 4.

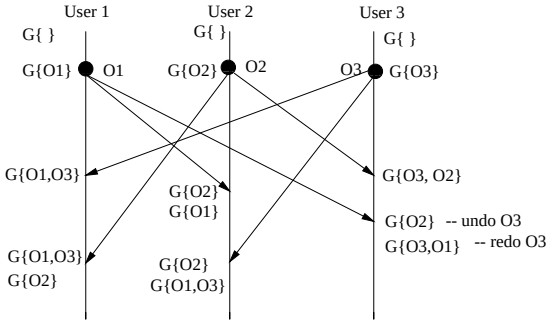

Fig. 4. A scenario for motivating CER3

Suppose the system has chosen to combine O_3 with O_1. At the site of user 3, the following abnormal phenomenon occurs: O_3 is first applied to its target object G to produce $G\{O_3\}$; then O_2 arrives and is combined with O_3 to produce $G\{O_3, O_2\}$ since they are compatible (site 3 has no knowledge about O_1 at this stage); finally O_1 arrives and is found to be conflicting with O_2, so O_3 has to be undone to produce $G\{O_2\}$, and then redone in a new version to produce $G\{O_3, O_1\}$ (to achieve the system chosen combined effect). In this scenario, user 3 will observe that O_3's effect is changing from one version to another version, due to the inconsistence between its initial effect and its final effect. This abnormal effect is undesirable, and also violates the intentions of operations since one operation (e.g., O_1) changes (by undoing) the effect of another independent operation (e.g., O_3). It should be pointed out that no matter which combined effect (O_3 combined with O_1 or O_2) the system chooses, at least one user (user 1 or user 3) in this scenario will observe that the original execution effect of O_3 is undone and then redone in another object.

To avoid this abnormal interface effect, we propose to combine O_3 with both O_1 and O_2 to produce $G\{O_1, O_3\}$ and $G\{O_2, O_3\}$. In this way, no matter which orders these three operations are executed, the final combined effect will be the same at all sites, without any abnormal interface effect. In general, we have the following additional rule to determine the combined effects of compatible operations in the face of mixed compatible and conflict operations.

Combined Effect Rule 3 (CER3): Given any group of operations, if they are mutually compatible and target the same object, then their effects must be combined in at least one common version of the target object.

In summary, **CER1**, **CER2** and **CER3** are the three criteria for judging whether a combined effect for a group of operations targeting the same object is correct or not. By applying these criteria, the following combined effects can be achieved: (1) conflicting operations are accommodated in different versions; (2) compatible operations are combined in common versions; (3) there is at least one pair of conflicting operations between any pair of versions; and (4) there is at least one version combining the effects of any group of compatible operations.

IV. Combined effects for any group of operations

In the previous section, simple scenarios have been used to derive and illustrate the criteria (i.e., **CER1**, **CER2** and **CER3**) to determine the combined effects of conflicting and compatible operations. However, in a highly concurrent real-time collaborative editing environment, a group of operations may have rather arbitrary and complex conflict relationships among them. A major technical problem here is: given an arbitrary group of operations targeting the same object, how to determine their combined effect, which is complying with **CER1**, **CER2** and **CER3**?

A. Conflict relation matrix and triangle

To solve this problem, we first introduce the *conflict relation matrix* to capture the complete picture of conflict relationships among any group of operations targeting the same object.

Given a group of n operations, $O_1, O_2, ..., O_n$, targeting the same object, their conflict relationships can be fully and uniquely expressed by a $n \times n$ *Conflict Relation Matrix (CRM)*, in which element $CRM[i, j], 1 \leq i, j \leq n$ is filled with "$\otimes$" if $O_i \otimes O_j$, otherwise it is filled with "$\odot$". For example, a 3×3 *CRM* for three operations is shown in Fig. 5-(a).

OP	O_1	O_2	O_3
O_1	$\odot$	$\otimes$	$\odot$
O_2	$\otimes$	$\odot$	$\odot$
O_3	$\odot$	$\odot$	$\odot$

(a) CRM

OP	O_2	O_3
O_1	$\otimes$	$\odot$
O_2		$\odot$

(b) CRT

Fig. 5. CRM versus CRT

Since $\otimes$ and $\odot$ relations are symmetric (i.e., $CRM[i, j] = CRM[j, i]$), and an operation is always compatible with itself (i.e.,

$CRM[i,i] = \odot$), by omitting these redundant and constant relation elements, the conflict matrix can be compressed to a $(n-1)\times(n-1)$ *Conflict Relation Triangle (CRT)*. For example, the 3×3 *CRM* in Fig 5-(a) can be compressed into an equivalent 2×2 *CRT* in Fig. 5-(b).

B. Compatible groups set

An alternative way of expressing the conflict/compatible relationships for a group of operations is called *Compatible Groups Set (CGS)*, which is defined as follows.

Definition 7: Compatible Groups Set
Given a group of operations GO, its corresponding Compatible Groups Set (CGS) is expressed as follows:

$$CGS = \{CG_1, CG_2, ..., CG_n\}$$

where $CG_i = \{O_1, O_2, ..., O_k\}$, and (1) all operations in any CG_i must be mutually compatible; (2) for any operation $O \in GO$, there must be at least one $CG_i \in CGS$, such that $O \in CG_i$; and (3) for any pair of operations O_x, $O_y \in GO$, if $O_x \odot O_y$, there must be at least one $CG_i \in CGS$, such that $O_x, O_y \in CG_i$. □

For example, the conflict relation expressed by the CRT in Fig. 5-(b) can also be captured by: $CGS = \{\{O_1, O_3\}, \{O_2, O_3\}\}$. In general, given a CRT, a CGS can be derived by using the following algorithm.

Algorithm 1: Given a CRT for a group of N operations GO, a CGS corresponding to this CRT can be obtained as follows:

1. $CGS = \{\ \}$;
2. For $1 \le i \le N-1$, and $i < j \le N$
   If $CRT[i, j-1] = \odot$
   Then $CGS = CGS + \{\{O_i, O_j\}\}$;
3. For $1 \le i \le N$,
   If $O_i \notin CG_k$ for all $k \in \{1, 2, ..., |CGS|\}$
   Then $CGS = CGS + \{\{O_i\}\}$. □

For example, as shown in Fig. 6, $\{O_2, O_3\} \in CGS$ since $CRT[2, 3-1] = \odot$, and $\{O_1\} \in CGS$ since O_1 is not in any other CG in CGS.

OP	O_2	O_3
O_1	$\otimes$	$\otimes$
O_2		$\odot$

$$CGS = \{\{O_1\}, \{O_2, O_3\}\}$$

Fig. 6. A CRT and its corresponding CGS.

It should be noted that in the CGS, the compatible relationships among operations are *explicitly* expressed by their co-existence in at least one CG. However, the conflict relationships among operations are *implicitly* expressed by their non-coexistence in any CG.

C. Equivalent CGS

If two compatible groups sets CGS_i and CGS_j capture the same compatible relationships for the same group of operations, then they are *equivalent*, denoted as $CGS_i \equiv CGS_j$. There exist some transformation rules which can be used to transform a CGS into another equivalent CGS.

In the following, we use the notation $CG_i \odot CG_j$ to mean that all operations in both CG_i and CG_j are mutually compatible.

Rule 1: Given a CGS, for any pair $CG_i, CG_j \in CGS$, if $CG_i \not\subseteq CG_j$, $CG_j \not\subseteq CG_i$, and $CG_i \odot CG_j$, then $CGS \equiv CGS - \{CG_i, CG_j\} + \{CG_i \cup CG_j\}$. □

Rule 1 says that if none of the two groups embraces the other (non-embracing groups) and all operations in the two groups are mutually compatible (mutually-compatible groups), then these two groups can be replaced by their union. This rule can be extended to any $m(>2)$ non-embracing but mutually-compatible groups. With this rule, multiple small groups can be merged into a single big group which includes all mutually compatible operations.

Rule 2: Given a CGS, if there exist $CG_i, CG_j \in CGS$, $i \neq j$, such that $CG_i \subseteq CG_j$, then $CGS \equiv CGS - \{CG_i\}$. □

Rule 2 says that if one group is a subgroup of another group in a CGS, then the subgroup can be removed.

D. Normalized CGS

We are particularly interested in a special form of CGS, called *Normalized Compatible Groups Set (NCGS)*, which is defined below.

Definition 8: Normalized Compatible Groups Set
Given a CGS for any group of operations GO targeting the same object, the CGS is a *Normalized* CGS ($NCGS$), *iff:* (1) for any group of mutually compatible operations in GO, there must be at least one $CG \in CGS$, such that all these compatible operations coexist in CG; and (2) for any pair $CG_i, CG_j \in CGS$, there must be at least one $O_x \in CG_i$, and one $O_y \in CG_j$, such that $O_x \otimes O_y$. □

By using Rules 1 and 2, a CGS can always be transformed into a $NCGS$. The following algorithm can be used to obtain a $NCGS$ from a given CRT for any group of operations targeting the same object.

Algorithm 2: Given a CRT for a group of operations, a $NCGS$ corresponding to this CRT can be obtained as follows:

1. Obtain a CGS for this CRT by using Algorithm 1.
2. Apply Rule 1 to transform CGS into CGS', so that all non-embracing but mutually-compatible groups are replaced by their unions.
3. Apply Rule 2 to transform CGS' into CGS'', so that all subgroups are removed.
4. Return $NCGS = CGS''$. □

An example of applying Algorithm 2 to transform a CGS into a $NCGS$ is given in Fig 7.

The following theorem establishes the *uniqueness* property of the $NCGS$.

Theorem 1: Given a group of operations GO targeting the same object, the $NCGS$ for this GO is unique.

Proof: Suppose there are two $NCGS_1$ and $NCGS_2$ for the same GO. First, we prove that for $CG_x \in NCGS_1$, there must exist a $CG_y \in NCGS_2$, such that $CG_x = CG_y$. Since both $NCGS_1$ and $NCGS_2$ are for the same GO, all operations in CG_x of $NCGS_1$ must also be in $NCGS_2$. Moreover, since all operations in CG_x are mutually compatible, they must all be in at least one compatible group CG_y in $NCGS_2$ according to Condition (1) of Definition 8.

OP	O_2	O_3	O_4
O_1	$\otimes$	$\odot$	$\odot$
O_2		$\odot$	$\odot$
O_3			$\odot$

$$\begin{aligned} CGS = & \{\{O_1, O_3\}, \{O_1, O_4\}, \{O_2, O_3\}, \{O_2, O_4\}, \{O_3, O_4\}\} \\ \equiv & \{\{O_1, O_3, O_4\}, \{O_2, O_3, O_4\}, \{O_3, O_4\}\} \ (by\ Rule\ 1) \\ \equiv & \{\{O_1, O_3, O_4\}, \{O_2, O_3, O_4\}\} \ (by\ Rule\ 2) \\ \equiv & \ NCGS \end{aligned}$$

Fig. 7. A CRT, and its corresponding CGS and $NCGS$.

Furthermore, it is impossible for CG_y to contain one extra compatible operation O_y. Otherwise, there must be at least one compatible group CG'_x in $NCGS_1$, which contains both O_y and all operations in CG_x according to Condition (1) of Definition 8. Then, CG_x must be subgroup of CG'_x, which is contradicting to Condition (2) of Definition 8. Thus, CG_x and CG_y must contain the same group of compatible operations and hence $CG_x = CG_y$. By the same reasoning, it can be proven that for any $CG_y \in NCGS_2$, there must exists a $CG_x \in NCGS_1$, such that $CG_x = CG_y$. Thus the theorem follows. □

E. Combined effect specified by NCGS

The significance of the $NCGS$ is that it gives a formal specification of the combined effect for any group of operations targeting the same object.

Definition 9: NCGS specified combined effect
Given the $NCGS$ for a group of operations GO targeting object G, the combined effect for GO is as follows: (1) For each $CG \in NCGS$, there is one object version made from G. (2) For all operations in the same CG, they will be applied to the same version corresponding to the CG. □

The combined effect specified by the $NCGS$ is unique because the $NCGS$ for a group of operations GO is unique. Furthermore, the following theorem establishes that this combined effect complies with **CER1**, **CER2**, and **CER3**.

Theorem 2: The combined effects specified by the $NCGS$ satisfy **CER1**, **CER2** and **CER3**.

Proof: (1) For any pair of operations O_1 and O_2 in the $NCGS$, if $O_1 \otimes O_2$, they could never coexist in the same CG in the $NCGS$ according to Condition (1) of Definition 7. and hence they could never be applied to the same object version, which complies with **CER1**. (2) For any pair of compatible groups CG_i and CG_j in the $NCGS$, there must be at least one $O_x \in CG_i$, and one $O_y \in CG_y$, such that $O_x \otimes O_y$ according to Condition (2) of Definition 8. Since there is one-to-one correspondence between the compatible groups in the $NCGS$ and the object versions made according to the $NCGS$ specified combined effect, **CER2** is satisfied. (3) For a group of operations, if they are mutually compatible, they must coexist in at least one common CG according to Condition 1 of Definition 8, so they will be combined in at least one common object version, which complies with **CER3**. □

In summary, the major result in this section is that given a group of operations targeting the same object, their combined effect can be uniquely determined by the *NCGS*, and this combined effect complies with **CER1**, **CER2**, and **CER3**. The following sections will discuss how to achieve this unique and correct combined effect in a distributed, incremental, and consistent way.

V. Incremental creation of multiple versions

If the group of operations GO targeting the same object are all known in advance, the $NCGS$ for this GO can be constructed by using Algorithm 2; then multiple versions can be created and operations can be applied to proper versions according to the combined effects specified by the $NCGS$. However, in real-time collaborative editing sessions, operations can be generated concurrently and may arrive at different sites in different orders. Because of high responsiveness consideration, it is not proper (or feasible) to postpone executing an operation until all other potentially concurrent operations have arrived. An operation should be allowed to execute as long as it is in the right causal order. This means that the system has to execute the group of operations one after another to incrementally create versions (if necessary) and combine the effects of all operations. In other words, a distributed algorithm is needed to incrementally construct the $NCGS$ at all sites.

Suppose a group of n operations targeting the same object arrive (and become causally ready for execution) at a site in the following order: O_1, O_2, ..., O_n. The algorithm will construct a sequence of $NCGS$s: $NCGS_1$, $NCGS_2$, ..., $NCGS_n$ in such a way that $NCGS_i$ is the $NCGS$ for the group of operations from O_1 to O_i, and the final $NCGS_n$ is the $NCGS$ for the whole group of operations. To achieve this, two technial problems need to be solved: one is how to apply operation O_i on $NCGS_{i-1}$ to produce $NCGS_i$; and the other is how to identify all object versions corresponding to $NCGS_{i-1}$ at each step. The second problem will be addressed in the next section. In this section, a *Multiple Object Versions Incremental Creation (MOVIC)* algorithm will be proposed to address the first problem.

A. The MOVIC algorithm

The following notations will be used in the description of the MOVIC algorithm: (1) O_i represents the ith operation to execute at any site. (2) $NCGS_{i-1}$ represents the $(i-1)$th $NCGS$ for operations from O_1 to O_{i-1}. (3) $NCGS_i$ represents the ith $NCGS$ for operations from O_1 to O_i. (4) $O_i \odot CG$ means that O_i is compatible with all operations in CG. (5) $O_i \otimes CG$ means that O_i is conflicting with all operations in CG.

The objective of the MOVIC algorithm is to apply O_i to the $NCGS_{i-1}$ (i.e., to add O_i to proper existing compatible groups in the $NCGS_{i-1}$ or to create new compatible groups if necessary) to produce the $NCGS_i$.

Algorithm 3: $MOVIC(O_i, NCGS_{i-1}) : NCGS_i$
1. $NCGS_i := \{\ \}$; $C := |NCGS_{i-1}|$;
2. Repeat until $NCGS_{i-1} = \{\ \}$:
   (a) Remove one CG from $NCGS_{i-1}$;
   (b) If $O_i \odot CG$, then $CG := CG + \{O_i\}$;
   (c) Else if $O_i \otimes CG$, then $C := C - 1$;
   (d) Else
   - $CG_{new} := \{O | (O \in CG) \wedge (O \odot O_i)\}$;
   - $CG_{new} := CG_{new} + \{O_i\}$;
   - $NCGS_i := NCGS_i + \{CG_{new}\}$.

   (e) $NCGS_i := NCGS_i + \{CG\}$;
3. If $C = 0$, then
   (a) $CG_{new} := \{O_i\}$;
   (b) $NCGS_i := NCGS_i + \{CG_{new}\}$;

4. For any $CG_{new} \in NCGS_i$, if there is another $CG \in NCGS_i$, such that $CG_{new} \subseteq CG$, then $NGCS_i := NCGS_i - \{CG_{new}\}$. □

In the MOVIC algorithm, the $NCGS_i$ is first initialized to an empty set, and *C* (a counter for the number of CGs which are not fully conflicting with O_i) is initialized to the size of the current $NCGS_{i-1}$.

Then, O_i is checked against every CG in the $NCGS_{i-1}$ one by one (Note: the order is not significant). If O_i is compatible with all operations in CG, then O_i is added to CG, which means that O_i can be directly applied to that object version. Else if O_i is conflicting with all operations in CG, then O_i is not added to CG, which means O_i cannot be applied to that object version. In this case, the counter C is decremented. Otherwise, O_i must be partially compatible with some operations in CG. In this case, a new group CG_{new} is created, which contains all operations in CG which are compatible with O_i, and then O_i is added to CG_{new}, which means O_i is applied to a new object version corresponding to CG_{new}. The newly created CG_{new} and the existing CG (with possibly an additional O_i) are all added to the $NCGS_i$. Since O_i is added only to groups with operations which are all compatible with O_i, the resulting groups are ensured to be compatible groups (for Conditions 1 and 2 of Definition 7). Moreover, when O_i is compatible with multiple operations in an existing group, it is always added to that group or a new group containing all these compatible operations. In this way, Condition 1 of Definition 8 is satisfied.

After checking all CGs in the $NCGS_{i-1}$, if $C = 0$, then O_i must be either the first operation (i.e., $O_i = O_1$) or conflicting with all CGs in the $NCGS_{i-1}$. In this case, a new group $CG_{new} = \{O_i\}$ is created (for Condition 2 of Definition 7).

A last but very important step in the MOVIC algorithm is to check each newly created group CG_{new} to see whether it is a subgroup of another group in the $NCGS_i$. If this is the case, CG_{new} should be removed according to Rule 2 to ensure that there shall be at least one pair of conflicting operations in each pair of CGs in the new $NCGS_i$ (for Condition 2 of Definition 8).

Since creating a new compatible group corresponds to creating a new object version, and adding O_i to an existing group or a new group corresponds to applying O_i to the object version for that group, it is straightforward to derive the method of executing O_i on the object versions corresponding to the $NCGS_{i-1}$ as follows:

1. If O_i is added to an existing CG in Step 2-(b) of Algorithm 3, then O_i is applied to the existing object version corresponding to CG.
2. If a CG_{new} is created out of an existing CG in Step 2-(d), and this CG_{new} is not removed in Step 4, then a new object version corresponding to CG_{new} is created and O_i is applied to it.
3. If a CG_{new} with only O_i is created in Step 3-(a), then a new object version corresponding to the CG_{new} is created and O_i is applied to it.

B. Order independency property

The $MOVIC$ algorithm has a very important property: no matter in which orders a group of n operations are processed, the final $NCGS_n$ constructed by the MOVIC algorithm is the same because there is only one unique $NCGS$ for any group of operations (see Theorem 1). This property is called *order-independency*, which ensures that a consistent final result can be achieved at all collaborating sites regardless of different operation execution orders. A formal verification of this property is beyond the scope of this paper. Some examples are given below to illustrate the order-independency property.

Example 1: Given four operations O_1, O_2, O_3, and O_4, with their conflict relationships expressed in Fig. 8[1].

OP	O_2	O_3	O_4
O_1	$\otimes$	$\otimes$	$\odot$
O_2		$\otimes$	$\odot$
O_3			$\odot$

Fig. 8. The CRT for Example 1

Consider the following two different execution orders:
Execution Order 1: O_1, O_2, O_3, and O_4.

1. $NCGS_1 = \{\{O_1\}\}$
2. $NCGS_2 = \{\{O_1\}, \{O_2\}\}$
3. $NCGS_3 = \{\{O_1\}, \{O_2\}, \{O_3\}\}$
4. $NCGS_4 = \{\{O_1, O_4\}, \{O_2, O_4\}, \{O_3, O_4\}\}$

Execution Order 2: O_1, O_2, O_4, and O_3.

1. $NCGS_1 = \{\{O_1\}\}$
2. $NCGS_2 = \{\{O_1\}, \{O_2\}\}$
3. $NCGS_3 = \{\{O_1, O_4\}, \{O_2, O_4\}\}$
4. $NCGS_4 = \{\{O_1, O_4\}, \{O_4, O_3\}, \{O_2, O_4\}, \{O_4, O_3\}\}$
   $\equiv \{\{O_1, O_4\}, \{O_2, O_4\}, \{O_4, O_3\}\}$ $(by\ Rule\ 2)$

It can be seen that at Step 4 of Execution Order 2, O_3 is first checked against $\{O_1, O_4\}$ and a new group $\{O_4, O_3\}$ is created since $O_4 \odot O_3$ but $O_1 \otimes O_3$; then O_3 is checked against $\{O_2, O_4\}$ and another (exactly the same) new group $\{O_4, O_3\}$ is created for the same reason. However, one of the two new groups is removeed according to Rule 2. In this way, the final $NCGS_4$ is the same for two different execution orders.

Example 2: Given four operations O_1, O_2, O_3, and O_4, with their conflict relationships expressed in Fig. 9.

OP	O_2	O_3	O_4
O_1	$\otimes$	$\otimes$	$\odot$
O_2		$\odot$	$\odot$
O_3			$\odot$

Fig. 9. The CRT for Example 2

Consider the following two different execution orders:
Execution Order 1: O_1, O_2, O_3, and O_4.

1. $NCGS_1 = \{\{O_1\}\}$
2. $NCGS_2 = \{\{O_1\}, \{O_2\}\}$
3. $NCGS_3 = \{\{O_1\}, \{O_2, O_3\}\}$
4. $NCGS_4 = \{\{O_1, O_4\}, \{O_2, O_3, O_4\}\}$

Execution Order 2: O_1, O_2, O_4, and O_3.

1. $NCGS_1 = \{\{O_1\}\}$
2. $NCGS_2 = \{\{O_1\}, \{O_2\}\}$
3. $NCGS_3 = \{\{O_1, O_4\}, \{O_2, O_4\}\}$
4. $NCGS_4 = \{\{O_1, O_4\}, \{O_4, O_3\}, \{O_2, O_4, O_3\}\}$
   $\equiv \{\{O_1, O_4\}, \{O_2, O_4, O_3\}$ $(by\ Rule\ 2)$

[1] It should be noted that the conflict relationships between O_i and operations in $NCGS_{i-1}$ can be detected on-the-fly by examining the state-vector timestamps and other parameters of operations. The CRT is used here just to give a complete picture of the conflict relationships among all operations, which does not imply that a complete CRT for a group of operations has to be constructed before applying the MOVIC algorithm.

As shown in Step 4 of Execution Order 2, when O_3 is first checked against $\{O_1, O_4\}$, a new group $\{O_4, O_3\}$ is created since $O_4 \odot O_3$ but $O_1 \otimes O_3$; then O_3 is checked against $\{O_2, O_4\}$, and is added into this existing group (becoming $\{O_2, O_4, O_3\}$) since O_3 is compatible with all operations in this group. However, the new group $\{O_4, O_3\}$ is removeed since it is a subgroup of $\{O_2, O_4, O_3\}$ according to Rule 2. In this way, the final $NCGS_4$ is the same for two different execution orders.

VI. Consistent object identification

For the MOVIC algorithm to work, one important parameter has to be provided: the current $NCGS_{i-1}$, on which the new operation O_i is applied to produce $NCGS_i$. The technical issue here is: how to find the CGs in the $NCGS_{i-1}$ for O_i? Since a CG in the $NCGS_{i-1}$ corresponds to an object version made from the original object targeted by O_i, the above issue is converted into the question: how to find the object versions made from the original object targeted by the new operation O_i? The key to solving this problem is to devise an object identification scheme which is able to identify all object versions made from the same original object.

A. Requirements for object identification

To work in a multi-version and multi-replica (due to replicated architecture for the storage of shared documents) object-based graphics editing system, the object identification scheme must maintain the following three properties: (1) *Uniqueness:* every object at a site must have a unique identifier. (2) *Traceability:* multiple versions of the same object G must have identifiers which can be traced by using the identifier of G. (3) *Consistency:* multiple replicas of the same object at different sites must have the same identifier.

The uniqueness property ensures different objects at a site be distinguishable from each object. The traceability property ensures multiple versions of the same object be traceable by using the identifier of the original object. The consistency property ensures multiple replicas of the same object have the same identifier so that operations applied on one replica be also applied the other replicas. The three properties together ensure that an operation targeting an object be applied to all versions and all replicas of the same object at all sites.

B. Analysis of object identification issues

We start from a simple object identification scheme which is able to uniquely identify every object. Let ***Id(G)*** denote the identifier of object G. Suppose each operation O has a unique identifier, denoted as ***Id(O)*** [2]. Then, each object can be uniquely identified by the identifier of the operation which created this object. Under this scheme, when object G is created by operation O at a local site, G is assigned a unique identifier which is equal to $Id(O)$, i.e., $Id(G) = Id(O)$. When O is propagated to a remote site, a replica of the same object will be created and assigned the same identifier. When a non-create operation O is applied to an existing object G at the local site, O will take $Id(G)$ as one of its parameters (i.e., $Tgt(O) = Id(G)$). When O arrives at a remote site, its parameter $Tgt(O)$ can be used to find the right replica of the same object to apply. This simple identification scheme works well for single version systems, but fails when multiple versions of the same object can be created due to operation conflicts.

For example, consider three operations O_1, O_2, and O_3, targeting the same object G. Suppose their conflict relationships are: $O_1 \otimes O_2$, $O_1 \odot O_3$, and $O_2 \odot O_3$. Assume these three operations are executed at a site in the order of O_1, O_2, and O_3. To execute O_1, the target object G can be found by its original identifier $Id(G)$ ($= Tgt(O_1)$). To execute O_2, the target object G can still be found by $Id(G)$ ($= Tgt(O_2)$) because the previous execution of O_1 does not change the identifier of G. However, after executing both O_1 and O_2, two versions $G\{O_1\}$ and $G\{O_2\}$ have been made from G and the original G disappeared. When O_3 arrives with $Tgt(O_3) = Id(G)$, both $G\{O_1\}$ and $G\{O_2\}$ must be found in order to combine O_3's effect with them. The question is: how should $G\{O_1\}$ and $G\{O_2\}$ be identified so that they can be traced by using $Id(G)$?

To address the multiple versions identification problem, the simple identification scheme can be extended (1) to let both versions inherent the identifier of the original object so that they are traceable by using $Id(G)$; and (2) to let one version include one additional identifier of the operation which triggers the creation of that new version so that multiple versions are distinguishable from each other.

In this example, since O_2 triggers the creation of a new version, $G\{O_1\}$ could simply take the identifier of G, i.e., $Id(G\{O_1\}) = Id(G)$, but $G\{O_2\}$ will take $Id(G)$ plus $Id(O_2)$ as its identifier, i.e., $Id(G\{O_2\}) = Id(G) + Id(O_2)$ (the precise meaning of "+" will become clear at the end of this subsection). Clearly, $Id(G\{O_1\}) \neq Id(G\{O_2\})$, and both $G\{O_1\}$ and $G\{O_2\}$ are traceable by using $Id(G)$ since $Id(G)$ is included in both $Id(G\{O_1\})$ and $Id(G\{O_2\})$.

The above extended identification scheme is able to ensure multiple versions of the same object be distinguishable from each other and traceable from the identifier of the original object. However, it is not able to ensure consistency of the identifiers of multiple replicas of the same object. To illustrate this problem, assume the two conflicting operations in the previous example are executed at a different site in a different order: O_2 followed by O_1. In this scenario, it will be O_1 which triggers the creation of a new version, so $G\{O_1\}$ will take $Id(G)$ plus $Id(O_1)$ as its identifier, but $G\{O_2\}$ will simply take the identifier of G, i.e., $Id(G\{O_2\}) = Id(G)$. Clearly, the two replicas of the same object $G\{O_2\}$ have been identified differently when the two conflicting operations are executed in different orders.

To solve this problem, the previous identification scheme is revised to let both versions include one additional identifier of the corresponding conflicting operation. For the previous example, $G\{O_1\}$ should take $Id(G)$ plus $Id(O_1)$ as its identifier, i.e., $Id(G\{O_1\}) = Id(G) + Id(O_1)$; and $G\{O_2\}$ should take $Id(G)$ plus $Id(O_2)$ as its identifier, i.e., $Id(G\{O_2\}) = Id(G) + Id(O_2)$. With this revised scheme, no matter in which order conflicting operations are executed, multiple replicas of the same object version will be identified consistently.

The object identification scheme would not be completely correct if the following more subtle inconsistency scenario was not discovered and resolved. Given three operations: O_1, O_2, and O_3 targeting the same object G. Suppose their conflict relationships are: $O_1 \otimes O_2$, $O_1 \otimes O_3$, and $O_2 \odot O_3$. First, consider the outcome of executing these operations in the order of O_1, O_2 and O_3. After executing O_1, G becomes $G\{O_1\}$, but $Id(G\{O_1\}) = Id(G)$. After executing O_2, a new version $G\{O_2\}$ is created and is identified by $Id(G\{O_2\}) = Id(G) + Id(O_2)$. In the meanwhile, another version $G\{O_1\}$ is identified by $Id(G\{O_1\}) = Id(G) + Id(O_1)$ according to the revised identification scheme. Finally, when O_3 arrives, it will be applied to the existing versions $G\{O_2\}$ directly since $O_3 \odot O_2$. The final outcome of executing the three operations will be two versions: $G\{O_1\}$ with an identifier of $Id(G) + Id(O_1)$, and $G\{O_2, O_3\}$ with an identifier of $Id(G) + Id(O_2)$.

However, if the three operations are executed at a different site in

[2] One way of making the $Id(O)$ is to use a pair (sid, lc), where sid is the identifier of the site at which O is generated, and lc is the sum of the state vector value associated with O

a different order: O_1, O_3 and O_2, the final outcome of executing the three operations will also be two versions: $G\{O_1\}$ with an identifier of $Id(G) + Id(O_1)$, and $G\{O_3, O_2\}$ with an identifier of $Id(G) + Id(O_3)$ (because O_3 triggers the creation of $G\{O_3\}$). Clearly, the two replicas of the same object version $G\{O_2, O_3\}$ are identified by two different identifiers (i.e., $Id(G) + Id(O_3)$, and $Id(G) + Id(O_2)$)!

In recognizing this problem, the previous object identification scheme is further revised to let a version's identifier include the identifiers of all operations (e.g., both O_2 and O_3) which are conflicting with another operation (e.g., O_1), regardless which operation triggers the creation of this new version. Furthermore, it becomes clear that the collection of operation identifiers in the object identifier should be treated as a *set*, rather than a *list*. In a set representation, the order of adding a conflicting operation identifier into the object identifier is not significant.

C. The COID scheme

Based on the above analysis, a *Consistent Object IDentification (COID)* scheme is defined below.

Definition 10: The COID scheme
The identifier of object G consists of a set of operations identifiers:

$$Id(G) = \{Id(O_1), Id(O_2), ..., Id(O_n)\},$$

where $Id(O_i) \in Id(G)$, $1 \leq i \leq n$, *iff*: (1) O_i is the operation which created G, or (2) O_i has been applied to G, and O is conflicting with an operation O_x, which has been applied to another version made from G. □

In the context of the MOVIC algorithm, the COID scheme can be realized as follows:

1. When operation O creates an original object G, $Id(G)$ is constructed as follows: $Id(G) := \{Id(O)\}$.
2. When operation O triggers the creation of a new version G' from the target object G, $Id(G')$ is constructed as follows: $Id(G') := Id(G) + \{Id(O)\}$.
3. When operation O is applied to an existing object G, $Id(G)$ is extended to include $Id(O)$ if O is conflicting with any other operation (in G or in any other version).
4. When operation O is applied to one version of object G, every other version of G, denoted as G', is checked to see whether G' has the effect of an operation O_x, such that $O_x \otimes O$. If there exists such an O_x and $Id(O_x)$ has not been included in $Id(G')$, then $Id(G')$ is extended as follows: $Id(G') := Id(G') + \{Id(O_x)\}$.

The COID scheme maintains the *uniqueness* property because the *Create* operation is unique, and any two versions of the same object must have at least one pair of conflicting operations. Moreover, the COID scheme maintains the *consistency* property because for an object, the same set of versions will be replicated at all sites (due to the uniqueness property of the NCGS), and conflict relationships among all operations are the same at all sites. Finally, the COID scheme maintains the *traceability* property because the identifiers of all versions of the same object G are supersets of $Id(G)$. To answer the question raised at the beginning of this section, the following *Target Object VErsion Recognition (TOVER)* scheme is defined.

Definition 11: The TOVER scheme
Given any operation O_i, any object G is a version corresponding to a compatible group CG in the current $NCGS_{i-1}$ *iff* $Tgt(O_i) \subseteq Id(G)$. □

VII. Comparison to related work

Most existing collaborative graphics editing systems have adopted a conflict-prevention approach based on locking. Example systems based on locking include: Aspects [17], Ensemble [11], GroupDraw [3], and GroupGraphics [12]. In these systems, a user has to place a lock on an object before editing it, thus preventing other users from generating conflicting operations on the same object. For locking to work, however, there has to be a coordinating process in the system which keeps track of which object(s) has been locked so it can grant/deny permissions for locking requests. The problem with locking is that when an editing operation is generated, it has to wait for at least a round trip time of sending a request message to the coordinating process and receiving a grant message back, before it can be executed (if it is allowed) at the local site. This round trip delay in the Internet environment may significantly degrade the system's responsiveness. Various techniques have been proposed to overcome this problem. For example, Ensemble allows conflict-free operations to execute immediately without waiting for approval. While in GroupDraw, locally generated operations are executed right away and a message is sent to the coordinating process. If the coordinating process does not approve the operation, then the effect of that operation is undone, which may cause abnormal phenomena at the user interface.

In contrast to conflict-prevention approaches like locking, the multi-version strategy proposed in this paper allows conflict to occur. It provides a mechanism (i.e., multiple object versions) to accommodate conflicting operations in a consistent way. Conflict resolution is left to the users. Major advantages of this approach are: it helps achieve high responsiveness of the system and preserve the work concurrently produced by multiple users in the face of conflicts. However, locking does have its merit of reducing conflicts by enforcing mutual exclusion. In fact, we have found locking is actually complementary and compatible with the optimistical concurrency control strategies and proposed a novel *optional* locking scheme (in contrasting to existing *compulsory* locking schemes) to enhance the consistency maintenance capability of the system [15]. Our investigation into the integration of this optional locking scheme with the multi-version approach in the graphics editing domain will be reported in a forthcoming paper.

Another alternative conflict-resolution approach is *serialization*. With this approach, operations can be executed as soon as they are generated to give a quick response. Before an operation is executed, it must be checked against executed operations for possible conflict. If a conflict is detected, a total ordering (i.e., serialization) between operations is used to determine which operation's effect will appear. Examples of such systems are: GroupDesign [7] and LICRA [5]. This approach is essentially the *single-operation-effect* (determined by a total ordering) approach discussed in Section III. For problems with this approach and its major differences with our *all-operations-effect* approach, please refer to the analysis in Section III.

The most closely related work is the Tivoli whiteboard meeting-supporting tool developed at Xerox PARC [9]. Tivoli also used multiple object versions (called *replicas* in Tivoli) to accommodate the effects of conflicting operations. The major difference between the Tivoli approach and our GRACE approach is that in Tivoli conflict is defined at the object level, i.e., a conflict occurs whenever two concurrent operations target the same object; whereas in GRACE, conflict is defined at object attribute level, i.e., a conflict occurs only

when two concurrent operations target the same object *and* change the same attribute to different values. Consequently, Tivoli does not allow compatible operations (according to GRACE conflict definition) to be applied to the same object (e.g. concurrent *Move* and *Fill* operations cannot be applied to the same object), resulting in unnecessary object versions. To our knowledge, the GRACE system is the only one in which operation conflict is defined at the object attribute level to minimize the number of object versions. Consequently, the technical issues and solutions reported in this paper are unique and have never been addressed by any other work.

VIII. Conclusions and future work

In this paper, we have proposed a novel multi-version approach to conflict resolution in real-time collaborative graphics editing systems. This approach is able to preserve the work concurrently produced by multiple users in the face of conflicts, and to minimize the number of object versions for accommodating combined effects of conflicting and compatible operations. Major technical contributions of this work include a formal specification of a unique combined effect for any group of conflicting and compatible operations, a distributed algorithm for incremental creation of multiple object versions, and a consistent object identification scheme for multi-version and multi-replica graphics editing systems.

All algorithms and schemes presented in this paper have been implemented in the Internet-based GRACE prototype system in Java. The current GRACE prototype system has been developed mainly to test the feasibility of our approach and to explore system design and implementation issues. Efforts are being directed towards building a more robust and useful system, which will be used by external users in real application contexts to evaluate the research results from end-users' perspective.

The multi-version approach alone is not a complete solution to resolving conflicts in collaborative systems. Other complementary techniques should be intergrated to work in conjunction with the multi-version technique. We are in the process of devising a group awareness mechanism and an optional locking scheme to help minimize the chance of conflict. Work is underway to apply GRACE techniques to other advanced object-based graphics editing systems as well.

Acknowledgement

The work reported in this paper has been partially supported by ARC (Australia Research Council) Large Grants A49601841 and A00000711 and an ARC Small Grant.

References

[1] R.M. Baecker, *Readings in groupware and computer-supported cooperative work*, Morgan Kaufmann Publishers Inc., 1992.

[2] C. A. Ellis, et al: "Groupware: some issues and experiences," *CACM 34(1)*, pp.39-58, Jan. 1991.

[3] S. Greenberg, et al: "Issues and experiences designing and implementing two group drawing tools," In *Proc. of the 25th Annual Hawaii International Conference on the System Science*, pp. 139-250, Jan. 1992.

[4] S. Greenberg and D. Marwood:"Real time groupware as a distributed system: concurrency control and its effect on the interface," In *Proc. of ACM Conference on Computer Supported Cooperative Work*, pp. 207-217, Nov. 1994.

[5] R. Kanawati: "Licra: a replicated-data management algorithm for distributed synchronous groupware application," *Parallel Computing*, Vol. 22, pp.1733-1746, 1997.

[6] A. Karsenty and M. Beaudouin-Lafon: "An algorithm for distributed groupware applications," In *Proc. of 13th International Conference on Distributed Computing Systems*, pp. 195-202, May 1993.

[7] A. Karsenty, et al: "Groupdesign: shared editing in a heterogeneous environment," *Usenix Journal of Computing Systems*, 6(2), pp. 167-195, 1993.

[8] L. Lamport: "Time, clocks, and the ordering of events in a distributed system," *CACM 21(7)*, pp.558-565, July 1978.

[9] T.P. Moran, et al: "Some design principles for sharing in Tivoli, a whiteboard meeting-support tool," In *Groupware for Real-time Drawing: A Designer's Guide*, ed. by S. Geernberg, et al, pp. 24-36. McGraw-Hill, 1995.

[10] D. Nichols, et al: "High-latency, low-bandwidth windowing in the Jupiter collaboration system," In *Proc. of ACM Symposium on User Interface Software and Technologies*, pp. 111-120, Nov. 1995.

[11] R.E. Newman-Wolfe, et al: "Implicit locking in the Ensemble concurrent object-oriented graphics editor," In *Proc. of ACM Conference on Computer Supported Cooperative Work*, pp. 265-272, Nov. 1992.

[12] M.O. Pendergast:" Groupgraphics: prototype to product," In *Groupware for Real-time Drawing: A Designer's Guide*, Ed. by S. Geernberg, et al, pp. 209-227, McGraw-Hill, 1995.

[13] 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.

[14] C. Sun and C. A. Ellis: "Operational transformation in real-time group editors: issues, algorithms, and achievements," In *Proc. of ACM Conference on Computer-Supported Cooperative Work*, pp.59-68, Seattle, USA, Nov.14-18, 1998.

[15] C. Sun and R. Sosič "Optional locking integrated with operational transformation in distributed real-time group editors," In *Proc. of The 18th ACM Symposium on Principles of Distributed Computing*, pp.43-52, Atlanta, USA, May 4-6, 1999.

[16] C. Sun and R. Sosič: "Consistency maintenance in Web-based real-time group editors," *Proceedings of 19th IEEE International Conference on Distributed Computing Systems (workshop)*, pp. 15-22, Austin, TX, USA, May 31- June 4, 1999.

[17] Von Biel: "Groupware grows up," In *MacUser*, pp.207-211, June, 1991.

[18] Y. Yang, C. Sun, Y. Zhang, and X. Jia: "Real-time cooperative editing on the Internet," *IEEE Internet Computing*, May/June, 2000.