Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems

CHENGZHENG SUN and DAVID CHEN
Griffith University

Real-time collaborative graphics editing systems 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. Consistency maintenance in the face of concurrent accesses to shared objects is one of the core issues in the design of these types of systems. In this article, we propose an object-level multi-versioning approach to consistency maintenance in real-time collaborative graphic editors. This approach is novel in achieving intention preservation and convergence, in preserving the work concurrently produced by multiple users in the face of conflict, and in minimizing the number of object versions for conflict resolution. Major technical contributions of this work include a formal specification of a unique combined effect for an arbitrary group of conflict and compatible operations, a distributed algorithm for incremental creation of multiple object versions, a consistent object identification scheme for multiple object versions, and a convergent layering scheme for overlapping objects. All algorithms and schemes presented in this article have been implemented in an Internet-based GRACE (GRAphics Collaborative Editing) system.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.2.2 [**Software Engineering**]: Tools and Techniques—*User interfaces*; H.1.2 [**Models and Principles**]: User/Machine Systems—*Human factors*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*Synchronous interaction; Theory and models*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*Interaction techniques*

General Terms: Algorithms, Design, Human Factors

Additional Key Words and Phrases: Consistence maintenance, intention preservation, convergence, multi-versioning, GRACE, collaborative graphics editors, groupware, computer-supported cooperative work.

1. INTRODUCTION

Consistency maintenance is a fundamental issue in many areas of computing systems, including databases systems [Bernstein et al. 1987], distributed systems [Birman et al. 1991], and groupware systems [Baecker 1992; Sun et al. 1998]. Real-time

This work was supported in part by an ARC (Australian Research Council) Large Grant (No: A00000711).

Author's address: C. Sun and D. Chen, School of Computing and Information Technology, Griffith University, Qld 4111, Australia; email: {C.Sun, D.Chen}@cit.gu.edu.au; URL:http://www.cit.gu.edu.au/~scz.

collaborative editing systems are groupware systems which allow a group of users to view and edit the same text/graphics/image/multimedia document at the same time from geographically dispersed sites connected via the Internet [Ellis et al. 1991; Sun et al. 1998]. They can be used in CSCW (Computer-Supported Cooperative Work) applications [Baecker 1992], such as electronic conferencing/meeting, collaborative CAD (Computer Aided Design) and CASE (Computer Aided Software Engineering), and collaborative documentation systems. The great economic potential as well as great technical challenges in real-time collaborative editing systems have attracted the attention of researchers and developers in both CSCW and distributed computing communities in the last decade [Ellis et al. 1991; Karsenty and Beaudouin-Lafon 1993; Greenberg and Marwood 1994; Nichols et al. 1995; Sun et al. 1998; Sun and Ellis 1998; Sun 2000; Li 2000; Campbell 2000; Chen 2001].

The goal of our research is to investigate and develop advanced techniques underlying the construction of collaborative editors which meet the following requirements.

(1) *High responsiveness in the Internet environment:* The response to the local user's actions must be quick, even though collaborating users may reside on different machines connected via the Internet with a long and non-deterministic communication latency. High responsiveness is important because many users subconsciously base their perception of the quality of system service more on responsiveness than on functionality, and poor responsiveness decreases system's effectiveness in supporting collaborative work. While there is no limit on the Internet bandwidth increase using fiber optic technologies, the communication latency over a transcontinental connection cannot be reduced considerably below 100 milliseconds (the threshold value for user noticeable delay [Cheshire 1996]) due to the speed limit of electronic signals. It is the communication latency, rather than the bandwidth, which presents a major challenge to achieve high responsiveness for many Internet-based collaborative systems [Cheshire 1996; Nichols et al. 1995].

(2) *High concurrency:* Multiple users are allowed to concurrently edit any part of the shared document at any time, in order to facilitate free and natural information flow among collaborating users. The major challenge here is the management of the multiple streams of concurrent activities performed by multiple users so that system consistency can be maintained in the face of conflict.

To meet the requirement of high responsiveness in the Internet environment, it seems that the only way is to adopt a *replicated architecture* for the storage of shared documents: shared documents are replicated at the local storage of each collaborating site, so editing operations can be performed at local sites immediately and then propagated to remote sites. To support concurrent editing in the replicated architecture, three inconsistency problems have to be dealt with [Sun et al. 1998]:

(1) *divergence* – operations may arrive and be executed at different sites in different orders, resulting in divergent final document states 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 the user; and

(3) *intention violation* – the actual execution effect of an operation may be different from the intention of this operation.

The *intention* of an operation is defined as the execution effect which can be achieved by applying this operation on the document state from which the operation was generated [Sun et al. 1998].

A consistency model has been proposed by Sun et al. [1998] as a framework for systematically addressing these inconsistency problems. This consistency model has three corresponding consistency properties: (1) the *convergence* property requires all copies of the same document to be identical after executing the same collection of operations; (2) the *causality preservation* property requires that causally-related (or dependent) operations to be executed in their natural causal order; and (3) the *intention preservation* property requires that the effect of executing an operation in any document state is the same as the effect of executing this operation on the document state from which the operation was generated, and the effect of an operation should not be changed by independent operations. The consistency model requires dependent operations to be executed in their causal order but imposes no execution order constraint on independent operations. Since a newly generated local operation is guaranteed to be either independent of or causally before all forthcoming remote operations, it can always be executed immediately without violating any consistency property. In other words, this consistency model has laid the theoretical foundation for achieving good responsiveness by allowing local operations to be executed immediately after their generation.

This consistency model has been successfully applied in the text editing domain [Sun et al. 1998]. To achieve causality preservation, the well-known vector logical clock timestamping [Fidge 1988; Birman et al. 1991; Raynal and Singhal 1996] has been used to capture the causal relationships among operations and to selectively delay some dependent operations if they arrive out of causal ordering. To support both intention preservation and convergence under the constraint of high responsiveness, an optimistic concurrency control technique called operational transformation, has been used [Sun et al. 1998]. The novelty of operational transformation is that it allows independent operations to be executed in any order but ensures the final effect shall be intention-preserved and identical. It has been shown that some intention-preserved results achieved by operational transformation are not achievable by any traditional serialization protocols [Sun et al. 1998]. For an integrative review of the major issues and algorithms in operational transformation, readers are referred to [Sun and Ellis 1998].

Our work in consistency maintenance has been extended from the text editing domain to the graphics editing domain. The objective of this new research is to further validate the consistency model in a new domain and to devise new consistency maintenance techniques required in the new domain. Our research has found that causality preservation techniques based on vector logical clocks are generic and applicable to both text and graphics editors, but the techniques for achieving intention preservation and convergence are different in these two different domains. The major outcome of this research is a novel object-level multi-versioning technique for achieving intention preservation and convergence in the graphics editing domain, which is the focus of this article.

The basic ideas of the multi-versioning technique have been presented in an earlier conference article [Chen and Sun 1999]. The current article is an extensive revision of the conference publication, and includes the following additional technical contributions: the rules for determining desirable combined effects of conflict and compatible operations, a formal specification of a unique combined effect for an arbitrary group of concurrent operations, a correctness verification of the distributed algorithm for incremental creation of multiple object versions, a convergent layering scheme for overlapping objects, an Internet-based implementation of the multi-versioning technique, and a comparison study on the multi-versioning technique and operational transformation.

The rest of this article is organized as follows. In Section 2, the conflict and compatibility relationships among graphics editing operations are defined and a multi-version approach to intention preservation and conflict resolution is proposed. The rules for determining desirable combined effects of conflict and compatible operations are specified in Section 3. A formal specification of a unique combined effect for an arbitrary group of concurrent operations is given in Section 4. A distributed algorithm for incremental creation of multiple object versions is described in Section 5. A consistent object identification scheme for multi-versioning systems is presented in Section 6. Strategies for dealing with special conflict relationships are discussed in Section 7. A convergent layering scheme is described in Section 8. Some issues in implementing and testing the multi-versioning technique in an Internet-based prototype system are discussed in Section 9. The work reported in this article is compared to alternative approaches and related work in Section 10. Lastly, major results of this work are summarized and future work is discussed in Section 11.

2. A MULTI-VERSIONING APPROACH TO CONFLICT RESOLUTION

2.1 Graphics editing operations and their relationships

Graphics editing systems can be object-based or bitmap-based. The work reported in this article is confined to object-based collaborative graphics editing systems only. In object-based graphics editing systems, graphic objects such as lines, rectangles, circles, and so on, can be created and updated. Each object has a set of 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, such as *Move* and *Resize*, can be applied to change the attributes of that object. For example, a *Move* operation changes the position attribute of the object to which it is applied. A *Delete* operation can be used to delete an existing object.

If all operations are generated sequentially, then simply executing the operations in their sequential order will preserve the intentions of all operations. In a group editing environment, however, operations may be generated concurrently by multiple users. If these concurrent operations are targeting different objects, then they can be executed in any order without violating their intentions. However, concurrent operations may target the same object and conflict with each other. For example, suppose User 1 generates $O_1 = Move(G, X)$ to move object

G to position X, and User 2 concurrently generates $O_2 = Move(G, Y)$ 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 site for execution. 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 object. In general, two concurrent operations are in conflict if they are targeting the same object but changing the same attribute to different values.

The challenge here is how to preserve operation intentions in the face of conflict. To address this challenge, we first define precisely the causal ordering, dependent/independent, and conflict/compatible relationships among operations. Following Lamport [1978], 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, O_1 is causally ordered before O_2, expressed as $O_1 \rightarrow O_2$, if and only if: (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 relations.* Given any two operations O_1 and O_2, (1) O_2 is *dependent* on O_1 if and only if $O_1 \rightarrow O_2$. (2) O_1 and O_2 are *independent* (or *concurrent*), expressed as $O_1 \parallel O_2$, if and only if neither $O_1 \rightarrow O_2$, nor $O_2 \rightarrow O_1$.

To define operation conflict, the following notations are introduced: (1) $Target(O)$ denotes the identifier of the object targeted by operation O; (2) $Att.Key(O)$ denotes the attribute key targeted by O; and (3) $Att.Value(O)$ denotes the attribute value targeted by O.

Definition 3. *Conflict relation "$\otimes$".* Two operations O_1 and O_2 conflict with each other, expressed as $O_1 \otimes O_2$, if and only if (1) $O_1 \parallel O_2$; (2) $Target(O_1) = Target(O_2)$; (3) $Att.Key(O_1) = Att.Key(O_2)$; and (4) $Att.Value(O_1) \neq Att.Value(O_2)$.

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

Definition 4. *Compatibility relation "$\odot$".* Two operations O_1 and O_2 are compatible, expressed as $O_1 \odot O_2$, if and only if they do not conflict with each other, i.e., $\neg(O_1 \otimes O_2)$.

According to above conflict relation definition, a *Create* operation cannot conflict with any other operation because no concurrent operation can target the same object being created by this *Create* operation. Furthermore, the *Delete* operation can be regarded as a special updating operation that sets a special *Existence* attribute of an object to the *False* value. Consequently, a *Delete* operation cannot conflict with any other operation either since no other updating operation (e.g. *Move, Color*, etc.) is able to change this *Existence* attribute and concurrent *Delete* operations targeting the same object always set the *Existence* attribute to the same *False* value. In our system, when an object's *Existence* attribute has been set to *False*, it becomes invisible to the users, but all relevant information about this object is still kept internally for supporting group *undo* [Chen and Sun 2001].

2.2 Conflict resolution by accommodating all operations effects

For compatible operations, if they are targeting the same object, they can be applied to the same object. For conflict operations, how can their conflict be resolved without violating their intentions?

One possible conflict resolution strategy is to achieve a *null-effect* in the face of conflict, which means none of the conflict 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 Figure 1. The vertical lines in this figure represent the activities performed by the corresponding users, and arrows represent the propagation of operations from the local site to a remote site. The document states observable from each user's interface are illustrated by rectangular boxes with rounded corners, with labels 1.0 to 1.2 for User 1 and 2.0 to 2.2 for User 2. Initially, both sites have the same object G, as shown in rectangular boxes 1.0 and 2.0. Then, two concurrent operations are generated in the scenario: $O_1 = Move(G, X)$ by User 1, and $O_2 = Move(G, Y)$ by User 2. At the site of User 1, G is first moved to position X, resulting in the document state shown in the rectangular box 1.1. Next, O_2 arrives and is found to be conflicting with O_1, so O_2 is rejected, and O_1 is undone to restore G to its original position, as shown in the rectangular box 1.2 (which is equal to the state shown in the rectangular box 1.0). A similar process occurs at the site of User 2, resulting in the document states shown in rectangular boxes 2.1 and 2.2 in sequence. The final results at both sites are identical. However, this null-effect does not preserve the intentions of these 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

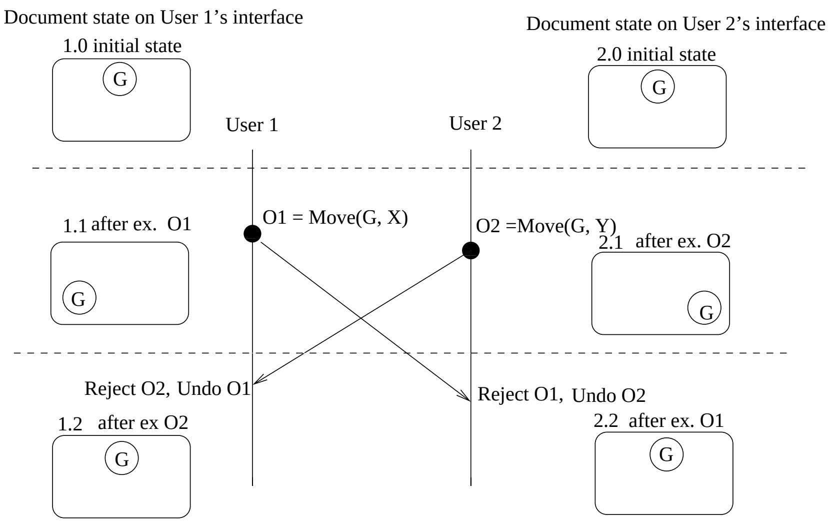

Fig. 1. Conflict resolution by a null-effect.

another independent operation. The consequence of this intention violation is that whenever there is a conflict, all work concurrently done by involved users will be destroyed, which is undesirable in the collaborative working environment. Moreover, this null-effect provides users with no explicit information about what other users intended to do in generating conflict operations, thus decreasing the users' ability to understand the nature of their conflict and to adjust their actions accordingly.

The second possible conflict resolution strategy is to achieve a *single-operation-effect*, which means the effect of only one operation, either O_1 or O_2, will be retained in the face of conflict. This can be achieved by enforcing a serialized effect among concurrent operations. As shown in Figure 2, when O_2 arrives at the site of User 1, it moves G to position Y, resulting in the document state shown in rectangular box 1.2; when O_1 arrives at the site of User 2, it is rejected, resulting in the document state shown in rectangular box 2.2 (which is the same as rectangular box 2.1). The final document states at both sites are identical. However, this conflict resolution result violates the intentions of both operations since one operation (O_1) has no effect at the site of User 2, and the other operation (O_2) has changed the effect of another independent operation (O_1) at the site of User 1. The consequence of this intention violation is that whenever there is a conflict, only one user's work can be preserved and users are not ensured to see the effects of the same set of operations. For example, 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 conflict operations, each user may see the effects of an 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 picture about what other users intended to do, and hence they may not be able to adjust their actions properly.

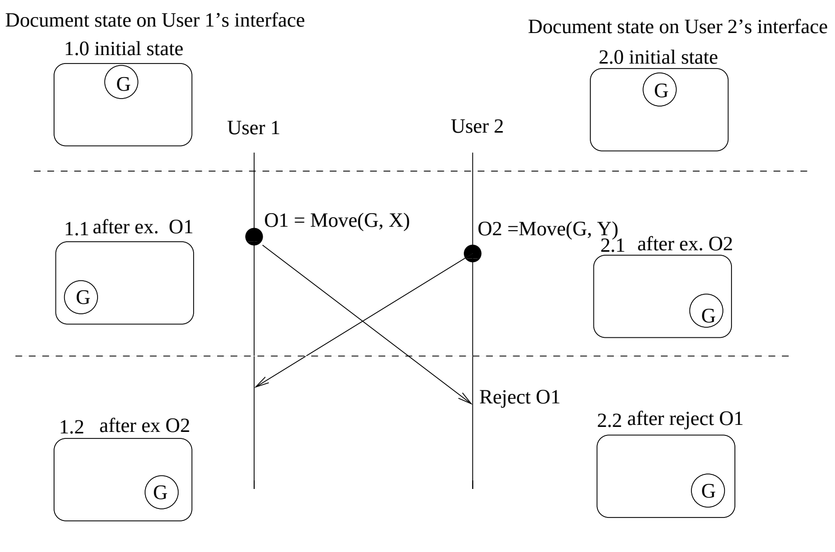

Fig. 2. Conflict resolution by a single-operation-effect.

The final conflict resolution strategy proposed in this article is to achieve an *all-operations-effect* by means of *multiple versioning*: 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 Figure 3.

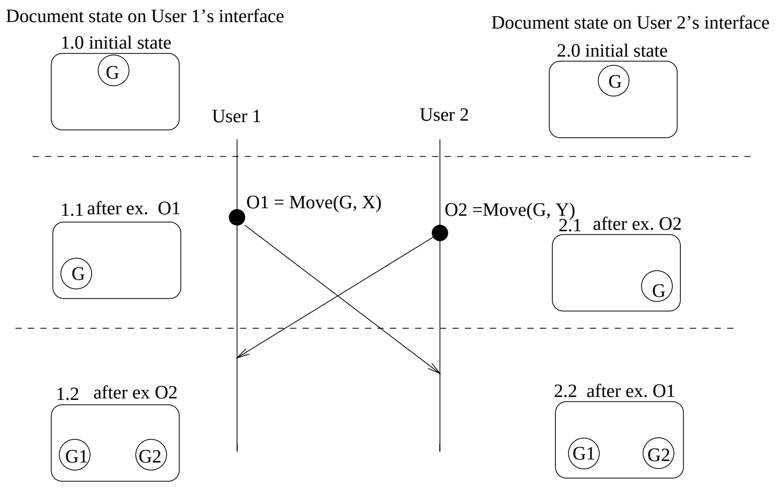

Fig. 3. Conflict resolution by all-operations-effect: the multi-versioning approach.

This all-operations-effect preserves the intentions of both operations since the effects of executing O_1 and O_2 at all sites are the same as the effects of executing them at their local sites, and these two independent operations do not change the effects of each other. Moreover, a system based on this conflict resolution strategy is able to ensure that the work produced by all users shall always be retained regardless of the existence of a conflict. Since all users are provided with a consistent and explicit picture about what other users intended to do in case of conflict, they could better assess the situation and react accordingly. Depending on application contexts, users may choose to keep all versions if that is desirable, or to remove some versions by using system-provided undo facilities [Sun 2000; Chen and Sun 2001]. One problem with this approach is that users may have trouble in differentiating the objects created due to conflict and the objects created by concurrent *Create* operations. A simple solution to this problem, adopted in our prototype system (see Section 9), is to notify the users of the conflict by highlighting the multiple versions created from the same object.

It is worth pointing out that a similar all-operations-effect strategy has also been used in the collaborative text editing domain [Sun et al. 1998]: when there are two concurrent operations inserting strings S_1 and S_2 at the same position, both strings are inserted into the document (one after the other) rather than keeping only one of them. We advocate a general groupware design principle: in case of conflict caused by concurrency, it is usually better to preserve all users' work, rather than to destroy any user's work. This system level conflict resolution result provides a better feedback to the users, helps the users to better understand the nature of their conflict and to better adjust and coordinate their actions in the face of conflict.

3. RULES FOR DETERMINING DESIRABLE COMBINED EFFECTS

Consider 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 will be created to accommodate each operation's effect in a separate version according to the multi-versioning strategy proposed in the previous section. 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 which operation to apply to which version. We need to define some basic rules for determining the desirable combined effects. 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 the initial state.

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 relationships are: $O_1 \otimes O_2$, $O_1 \otimes O_3$, and $O_2 \odot O_3$. Concrete examples of such three operations are: $O_1 = Move(G, X_1)$, $O_2 = Move(G, X_2)$, and $O_3 = Move(G, X_3)$, where $X_1 \neq X_2 \neq X_3$ and $O_1 \parallel (O_2 \rightarrow O_3)$. 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 $G\{O_2\}$. In general, we have the following combined effect rule:

Combined Effect Rule 1. (CER1) *Given two operations O_1 and O_2 targeting the same object G, if $O_1 \otimes O_2$, they must be applied to two 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 creates two versions $G\{O_2\}$ and $G\{O_3\}$ for two compatible operations O_2 and O_3. To avoid unnecessary versions, we propose combining the 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, we have the following combined effect rule 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 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 relationships are: $O_1 \otimes O_2$, $O_1 \odot O_3$, and $O_2 \odot O_3$. Concrete examples of such three operations are: $O_1 = Move(G, X_1)$, $O_2 = Move(G, X_2)$, and $O_3 = Color(G, Red)$, where $X_1 \neq X_2$ and $O_1 \parallel O_2 \parallel O_3$. Since $O_1 \otimes O_2$, two versions $G\{O_1\}$ and $G\{O_2\}$ need to be created according to CER1. Since O_3 is compatible with both O_1 and O_2, no new object version should be created for O_3 according to CER2. To which existing versions should O_3 be applied?

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 any total ordering). This approach does not produce any unnecessary version, but may exhibit an anomaly at the user interface, as shown in Figure 4.

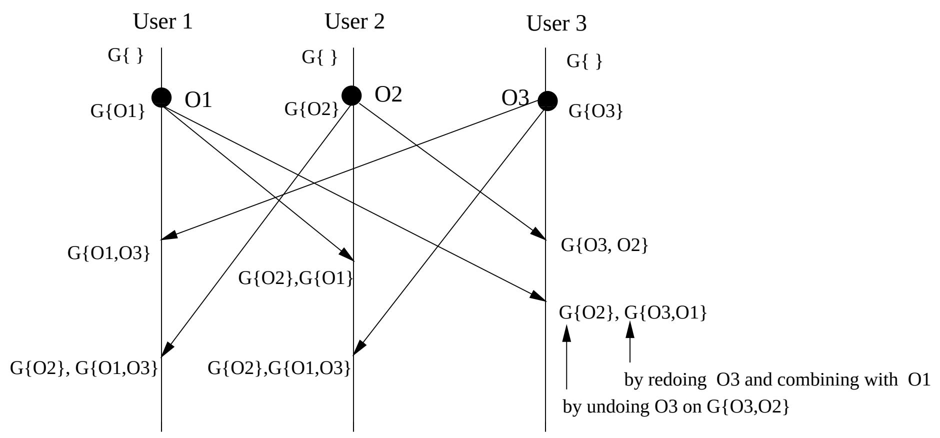

Fig. 4. A scenario for motivating CER3

Suppose the system has chosen to combine O_3 with O_1. At the site of User 3, the following anomaly occurs. First, O_3 is 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 inconsistency between its initial effect and its final effect. This anomaly is undesirable, and also violates the intentions of operations since one operation (O_1) changes (by undoing) the effect of another independent operation (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 Figure 4 will observe that the original execution effect of O_3 is undone and then redone in another object.

To avoid this anomaly, we propose combining O_3 with both compatible operations O_1 and O_2 to produce $G\{O_1, O_3\}$ and $G\{O_2, O_3\}$. In this way, no matter in which order these three operations are executed, the final combined effect will be the same at all sites, without any anomaly. Therefore, we have the following additional rule to determine the combined effects of a pair of compatible operations.

COMBINED EFFECT RULE 3. (CER3) *Given a pair of operations O_1 and O_2 targeting the same object G, if $O_1 \odot O_2$, then their effects must be combined in G or in one common object version made from G.*

In summary, CER1, CER2 and CER3 are the three criteria for judging the correctness of a combined effect for a group of operations targeting the same object. For any combined effect to comply with CER1-3, it should accommodate compatible operations in common versions and conflict operations in different versions, and should create a new in object version only when this new version contains at least one operation which cannot be accommodated in each existing object version.

4. COMBINED EFFECT FOR AN ARBITRARY GROUP OF OPERATIONS

In a highly concurrent collaborative editing environment, a group of operations may have rather arbitrary and complex conflict relationships. A major technical challenge here is: how to determine the combined effect that complies with CER1–3 and that is identical (convergent) at all sites for an arbitrary group of operations targeting the same object? Our research has found that a unique combined effect complying with CER1-3 can be derived from the inherent conflict/compatibility relationships in an arbitrary group of concurrent operations. In this section, the method of deriving the unique combined effect is presented.

4.1 Conflict relation matrix and triangle

To capture the complete picture of conflict relationships among an arbitrary group of operations, we first introduce the notation of *conflict relation matrix*.

Given a group of n operations, $O_1, O_2, ..., O_n$, targeting the same object, their conflict relationships can be fully expressed by an $n \times n$ *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 Figure 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)

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

(b)

Fig. 5. (a) CRM versus (b) CRT

It can be observed that $\otimes$ and $\odot$ relations are symmetric (i.e., $CRM[i,j] = CRM[j,i]$), and an operation is always compatible with itself (i.e., $CRM[i,i] = \odot$). Therefore, the conflict matrix can be compressed to a $(n-1) \times (n-1)$ *Conflict Relation Triangle (CRT)* by omitting these redundant and constant relationship elements. For example, the 3×3 CRM in Fig 5-(a) can be compressed into an equivalent 2×2 CRT in Figure 5-(b) by removing Column 1 and Row 3 from the 3×3 CRM and leaving $CRM[2,2]$ empty since it is always equal to $\odot$. Due to its conciseness, CRT will be used in the following discussions for expressing conflict relationships for a group of operations.

4.2 Compatible groups set

Given a group of operations GO, if a subgroup of GO contains mutually compatible operations only, then this subgroup is a *Compatible Group (CG)* for this

GO. For example, for the group of operations $GO = \{O_1, O_2, O_3\}$ in Figure 5, the subgroup $\{O_1, O_3\}$ is a CG for this GO.

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

Definition 5. *Compatible Groups Set (CGS).* Given a group of operations GO, the conflict relationships among these operations can be expressed as a CGS as follows:

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

where (1) all operations in any CG_i $(1 \leq i \leq m)$ are mutually compatible; (2) for any operation $O \in GO$, there exists 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 Figure 5-(b) can also be expressed by $CGS = \{\{O_1, O_3\}, \{O_2, O_3\}\}$. In general, given a CRT, a CGS can be derived from it 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 $1 \leq i \leq N-1$, and $i < j \leq N$,
    if $CRT[i, j-1] = \odot$, then $CGS = CGS + \{\{O_i, O_j\}\}$;
(3) For $1 \leq i \leq N$,
    if $O_i \notin CG_k$ for all $k \in \{1, 2, ..., |CGS|\}$, then $CGS = CGS + \{\{O_i\}\}$.
```

For example, as shown in Figure 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 a CGS, the compatibility relationships among operations are *explicitly* expressed by the fact that they co-exist in at least one CG. However, the conflict relationships among operations are *implicitly* expressed by the fact that they do not co-exist in any CG.

4.3 Equivalent CGS

If two compatible group sets CGS_i and CGS_j capture the compatibility 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 every operation in CG_i is compatible with every operation in CG_j.

Transformation Rule 1. (TR1) *Let S be a CGS for a group of operations. For any pair $CG_i, CG_j \in S$, if $CG_i \not\subseteq CG_j$, $CG_j \not\subseteq CG_i$, and $CG_i \odot CG_j$, then $S \equiv S - \{CG_i, CG_j\} + \{CG_i \cup CG_j\}$.*

TR1 says that if two compatible groups do not embrace each 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 in these small groups.

Transformation Rule 2. (TR2) *Let S be a CGS for a group of operations. If there exist $CG_i, CG_j \in S$, $i \neq j$, such that $CG_i \subseteq CG_j$, then $S \equiv S - \{CG_i\}$.*

TR2 says that if one group is a subgroup of another group in a CGS, then the subgroup can be removed. With this rule, redundant compatible subgroups can be absorbed by bigger groups in a CGS.

An example of applying TR1 and TR2 to transform a CGS into another equivalent CGS is given in Figure 7.

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

$$\begin{aligned} CGS = & \{\{O_1,O_3\},\{O_1,O_4\},\{O_2,O_3\},\{O_2,O_4\},\{O_3,O_4\}\} \\ \equiv & \{\{O_1,O_3,O_4\},\{O_2,O_3,O_4\},\{O_3,O_4\}\} \ (by\ TR1) \\ \equiv & \{\{O_1,O_3,O_4\},\{O_2,O_3,O_4\}\} \ (by\ TR2) \end{aligned}$$

Fig. 7. An example of transforming CGS.

4.4 Maximal compatible groups set

As discussed in the previous subsection, multiple small CGs can be merged into a single big CG by applying TR1. If a compatible group cannot be extended to include additional compatible operations, then this compatible group is a *maximal compatible group*, as defined below.

Definition 6. *Maximal Compatible Group (MCG).* Let CG_x be a compatible group for a group of operations GO targeting the same object. CG_x is an MCG for GO if and only if for any $O_x \in GO$ and $O_x \notin CG_x$, there exists an operation $O_y \in CG_x$ such that $O_x \otimes O_y$.

In Figure 7, for example, the compatible group $CG_x = \{O_1, O_3, O_4\}$ for $GO = \{O_1, O_2, O_3, O_4\}$ is an MCG since O_2 is the only operation which is not included in CG_x, and there exists an operation $O_1 \in CG_x$ that conflicts with O_2.

We are particularly interested in a special form of CGS consisting of MCGs only, as defined below.

Definition 7. *Maximal Compatible Groups Set (MCGS).* Given a CGS for a group of operations GO targeting the same object, the CGS is an $MCGS$ if and only if: (1) every CG in the CGS is an MCG; and (2) all MCGs belonging to the GO are included in the CGS.

As an example, the $CGS = \{\{O_1, O_3, O_4\}, \{O_2, O_3, O_4\}\}$ in Figure 7 is an $MCGS$ since (1) both $\{O_1, O_3, O_4\}$ and $\{O_2, O_3, O_4\}$ are MCGs; and (2) the $GO = \{O_1, O_2, O_3, O_4\}$ contains only these two MCGs.

The main reason that we are interested in $MCGS$ is that $MCGS$ has a very important property – *uniqueness*, as shown below.

Property 1. *Given a group of operations GO targeting the same object, there is a unique MCGS for this GO.*

Proof. Suppose there are two $MCGS_1$ and $MCGS_2$ for the same GO. For any $CG_x \in MCGS_1$, since CG_x is an MCG in $MCGS_1$ (by Condition (1) of Definition 7), it must also be included in $MCGS_2$ (by Condition (2) of Definition 7). Therefore, $MCGS_1 \subseteq MCGS_2$. For any $CG_y \in MCGS_2$, on the other hand, it must be that $CG_y \in MCGS_1$ (by the same reasoning), so $MCGS_2 \subseteq MCGS_1$. Thus, $MCGS_1 = MCGS_2$. □

Moreover, another important property can be derived from Condition (1) of Definition 7.

Property 2. *Let M be an MCGS for a group of operations. For any pair $CG_i, CG_j \in M$, there must be at least one $O_x \in CG_i$, and one $O_y \in CG_j$, such that $O_x \otimes O_y$.*

Proof. If there exists no $O_x \in CG_i$, and no $O_y \in CG_j$, such that $O_x \otimes O_y$, then all operations in CG_i must be compatible with all operations in CG_j, which implies that CG_i and CG_j could be merged into a bigger CG (by TR1), This contradicts Condition (1) of Definition 7, which requires every CG in an $MCGS$ to be an MCG. □

4.5 Combined effect specified by MCGS

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

Definition 8. *MCGS specified combined effect.* Let M be the $MCGS$ for a group of operations GO targeting object G. The combined effect for this GO is (1) for each $CG \in M$, there is one object version made from G; and (2) for all operations in the same CG, they will be applied to the same version corresponding to the CG.

Since the $MCGS$ for a group of operations is unique, the combined effect specified by the $MCGS$ is also unique. Furthermore, the following theorem establishs that this combined effect complies with CER1-3.

Theorem 1. *The combined effects specified by the MCGS for a group of operations GO comply with CER1, CER2, and CER3.*

PROOF. Three cases are examined below.

(1) For any pair of operations O_1 and O_2 in the GO, if $O_1 \otimes O_2$, they could never coexist in the same CG in the $MCGS$ according to Condition (1) of Definition 5 (since an $MCGS$ is also a CGS). Therefore, O_1 and O_2 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 $MCGS$, there must be at least one $O_x \in CG_i$, and one $O_y \in CG_j$, such that $O_x \otimes O_y$ according to Property 2 of the $MCGS$. Since there is a one-to-one correspondence between the compatible groups in the $MCGS$ and the object versions made according to the $MCGS$ specified combined effect (Definition 8), CER2 is satisfied.

(3) For a pair of operations O_1 and O_2 in the GO, if $O_1 \odot O_2$, they must coexist in at least one CG according to Condition (3) of Definition 5. Therefore, O_1 and O_2 will be combined in at least one common object version, which complies with CER3.

□

In summary, the major result in this section is the following: given a group of operations GO targeting the same object, their combined effect can be uniquely determined by the $MCGS$ for the GO, and this combined effect complies with CER1-3. The uniqueness of this combined effect is essential to achieve convergence at all sites in constructing multiple versions of objects. The following sections will discuss how to achieve this unique and correct combined effect in a distributed and incremental way.

5. DISTRIBUTED AND INCREMENTAL CREATION OF MULTIPLE VERSIONS

In real-time collaborative editing sessions, operations can be generated concurrently and may arrive at different sites in different orders. A distributed algorithm is needed to incrementally construct the unique $MCGS$ for this group of operations at all sites. Suppose a group of n operations targeting the same object are executed at a site in the order O_1, O_2, ..., O_n. The algorithm will construct a sequence of $MCGS$s: $MCGS_1$, $MCGS_2$, ..., $MCGS_n$ in such a way that $MCGS_i$ is the $MCGS$ for the group of operations from O_1 to O_i, and the final $MCGS_n$ is the $MCGS$ for the whole group of operations. To achieve this, two technical problems need to be solved: one is how to apply operation O_i on the $MCGS_{i-1}$ to produce $MCGS_i$; and the other is how to identify all object versions corresponding to $MCGS_{i-1}$ at each step. In this section, a *Multiple Object Versions Incremental Creation (MOVIC)* algorithm will be proposed to address the first problem. The second problem will be addressed in the next section.

5.1 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) $MCGS_i$ represents the ith $MCGS$ for operations from O_1 to O_i.

(3) $O_i \odot CG_x$ means that O_i is compatible with all operations in CG_x.

(4) $O_i \otimes CG_x$ means that O_i is conflicting with all operations in CG_x.

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

Algorithm 2. $MOVIC(O_i, MCGS_{i-1}) : MCGS_i$

```
(1) $MCGS_i := \{ \}$; $C := |MCGS_{i-1}|$;
(2) Repeat until $MCGS_{i-1} = \{ \}$:
    (a) Let $CG_x$ be a compatible group in $MCGS_{i-1}$.
        Remove $CG_x$ from $MCGS_{i-1}$;
    (b) If $O_i \odot CG_x$, then $CG_x := CG_x + \{O_i\}$;
    (c) Else if $O_i \otimes CG_x$, then $C := C - 1$;
    (d) Else
        —$CG_{new} := \{O | (O \in CG_x) \wedge (O \odot O_i)\}$;
        —$CG'_x := CG_{new} + \{O_i\}$;
        —$MCGS_i := MCGS_i + \{CG'_x\}$.
    (e) $MCGS_i := MCGS_i + \{CG_x\}$;
(3) If $C = 0$, then
    (a) $CG_{new} := \{O_i\}$;
    (b) $MCGS_i := MCGS_i + \{CG_{new}\}$;
(4) Let $CG_{new}$ be any new $CG$ created in Step 2-(d) or Step 3. For every $CG_{new} \in$
    $MCGS_i$, if there is another $CG_y \in MCGS_i$, such that $CG_{new} \subseteq CG_y$, then
    $MCGS_i := MCGS_i - \{CG_{new}\}$.
```

In the MOVIC algorithm, the $MCGS_i$ 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 $MCGS_{i-1}$.

Then, O_i is checked against every compatible group CG_x in the $MCGS_{i-1}$ (the order is not significant). If O_i is compatible with all operations in CG_x, then O_i is added to CG_x (to ensure this CG_x to be an MCG in the $MCGS_i$ for meeting Condition (1) of Definition 7). Else if O_i is conflicting with all operations in CG_x, then O_i is not added to CG_x. In this case, the counter C is decremented (to be used later). Otherwise, O_i must be partially compatible with some operations in CG_x. In this case, a new group CG_{new} is created, which contains the operations in CG_x which are compatible with O_i, and then O_i is added to CG_{new}, resulting in a new compatible group CG'_x. If this CG'_x is not absorbed by another CG in Step (4), it will become a new MCG in the $MCGS_i$ (for meeting Condition (2) of Definition 7).

After checking all CGs in the $MCGS_{i-1}$, 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 $MCGS_{i-1}$. In this case, a new group $CG_{new} = \{O_i\}$ is created (for meeting Condition (2) of Definition 5).

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 compatible group CG_y in the $MCGS_i$. If this is the case, CG_{new} should be removed according to TR2 to ensure that every CG in the $MCGS_i$ is an MCG (for meeting Condition (1) of Definition 7). It should be noted that TR1 is not used in the MOVIC algorithm to merge non-embracing but mutually compatible groups since such groups could not be created in execution of the MOVIC algorithm.

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 $MCGS_{i-1}$ as follows:

(1) If O_i is added to an existing compatible group CG_x in Step 2-(b) of Algorithm 2, then O_i is applied to the existing object version corresponding to CG_x.

(2) If a CG'_x is created out of an existing CG_x in Step 2-(d), and this CG'_x is not removed in Step 4, then a new object version corresponding to CG'_x is created.

(3) If a CG_{new} with only O_i is created in Step 3, then a new object version corresponding to the CG_{new} is created.

To create a new object version out of an existing one, we can first make an identical copy of the existing object and then undo the operations in its corresponding CG which are conflicting with the new operation O_i.

5.2 Correctness and order independence

The following theorem formally establishes the correctness of the MOVIC algorithm, that is, the compatible groups set constructed by the MOVIC algorithm is indeed an $MCGS$.

THEOREM 2. *Let $GO = \{O_1, O_2, ..., O_n\}$ be a group of operations targeting the same object. The $MCGS_n$ obtained by applying the MOVIC algorithm on these operations in the order of O_1, O_2, ..., O_n is the $MCGS$ for the GO.*

PROOF. We prove this theorem by applying induction on the sequence number i of $MCGS_i$, $1 \leq i \leq n$. The theorem clearly holds for $i = 1$, i.e., $MCGS_1 = \{\{O_1\}\}$ is the $MCGS$ for the operation group $GO_1 = \{O_1\}$.

Let the induction hypothesis be that the theorem holds for $i = m$, i.e., the $MCGS_m$ is the $MCGS$ for the operation group $GO_m = \{O_1, O_2, ..., O_m\}$. We need to show that the theorem holds for $i = m + 1$, i.e., the $MCGS_{m+1}$ is the $MCGS$ for the operation group $GO_{m+1} = \{O_1, O_2, ..., O_m, O_{m+1}\}$.

First, we show that every CG in the $MCGS_{m+1}$ is an MCG for the GO_{m+1} (Condition (1) of Definition 7). For any $CG_x \in MCGS_{m+1}$, it must belong to one of the following two cases:

Case 1. $O_{m+1} \notin CG_x$. According to the MOVIC algorithm, it must be that (1) $CG_x \in MCGS_m$, and (2) there is at least one operation $O_x \in CG_x$, such that $O_x \otimes O_{m+1}$ (so that O_{m+1} cannot be combined with CG_x in Step 2-(b)). By the induction hypothesis, CG_x is an MCG for the GO_m. CG_x must also be an MCG for the GO_{m+1} since CG_x cannot be extended to include the new operation O_{m+1} in the GO_{m+1}.

Case 2. $O_{m+1} \in CG_x$. According to the MOVIC algorithm, CG_x must belong to one of the following three subcases:

(1) $CG_x = \{O_{m+1}\}$. In this case, CG_x must have been created due to the fact that O_{m+1} is conflicting with all operations in the GO_m (Step 3), so CG_x must be an MCG for the GO_{m+1}.

(2) $CG_x = CG'_x + \{O_{m+1}\}$, where $CG'_x \in MCGS_m$. In this case, O_{m+1} must be compatible with all operations in an existing compatible group CG'_x (Step 2-(b)). By the induction hypothesis, CG'_x is an MCG for the GO_m. CG_x must be an MCG for the GO_{m+1}. Otherwise, suppose there was an $O_x \in GO_{m+1}$, $O_x \notin CG_x$, but $O_x \odot CG_x$. From $O_x \notin CG_x$ and $O_{m+1} \in CG_x$, we know that $O_x \neq O_{m+1}$ and $O_x \in GO_m$. Furthermore, we have $O_x \notin CG'_x$ (from $O_x \notin CG_x$ and $CG'_x \subset CG_x$) and $O_x \odot CG'_x$ (from $O_x \odot CG_x$ and $CG'_x \subset CG_x$), which implies that CG'_x is not an MCG for the GO_m, contradicting the induction hypothesis.

(3) $CG_x = CG_{new} + \{O_{m+1}\}$, where $CG_{new} \subset CG'_x$ and $CG'_x \in MCGS_m$. In this case, CG_x must have been created due to the fact that O_{m+1} is compatible with a subgroup of operations (CG_{new}) in an existing $CG'_x \in MCGS_m$ (Step 2-(d)). Since CG_x is not absorbed by a bigger group in Step 4, CG_x must be an MCG for the GO_{m+1}. Otherwise, suppose there was an operation $O_x \in GO_{m+1}$, and $O_x \odot CG_x$ but $O_x \notin CG_x$. From $O_x \notin CG_x$, we know $O_x \neq O_{m+1}$ and $O_x \in GO_m$. From $O_x \odot CG_x$, we know $O_x \odot CG_{new}$ since $CG_{new} \subset CG_x$. Then, there must be an MCG_x for the GO_m which contains O_x and all operations in CG_{new}. By the induction hypothesis, this MCG_x must be included in the $MCGS_m$. According to the MOVIC algorithm (Step 2-(d)), however, O_{m+1} must be checked against MCG_x and be combined with O_x and all operations in CG_{new} to form a compatible group $CG'' = CG_{new} + \{O_{m+1}, O_x\}$, which could have absorbed CG_x in Step 4, contradicting the fact that CG_x has not been absorbed in Step 4.

Second, we show that all MCGs for the GO_{m+1} must be included in the $MCGS_{m+1}$ (Condition (2) of Definition 7). Let MCG_x be any maximal compatible group for the GO_{m+1}. Assume MCG_x is not included in the $MCGS_{m+1}$. MCG_x must contain O_{m+1} because all MCGs for the GO_{m+1} that do not contain O_{m+1} must be included in the $MCGS_{m+1}$ by inheriting them from the $MCGS_m$ (by the induction hypothesis and the MOVIC algorithm). Suppose $MCG_x = CG_x + \{O_{m+1}\}$, where CG_x is a compatible group for the GO_m. There must be an MCG_y for the GO_m, such that $CG_x \subseteq MCG_y$. Moreover, MCG_y must be included in the $MCGS_m$ (by the induction hypothesis). According to the MOVIC algorithm (the loop in Step 2), O_{m+1} must be checked against MCG_y and be combined with all operations in CG_x (which is a subset of MCG_y) to create the $MCG_x = CG_x + \{O_{m+1}\}$ in the $MCGS_{m+1}$, which contradicts the assumption that MCG_x is not included in the $MCGS_{m+1}$. Therefore, the theorem holds for $i = m + 1$. □

In the correctness proof of the MOVIC algorithm, there is no requirement for any specific order of processing a group of operations targeting the same object, which implies a very important property of the MOVIC algorithm – *order independency.*

PROPERTY 3. *Given a group of n operations targeting the same object, the final $MCGS_n$ constructed by the MOVIC algorithm is the same no matter in which order these operations are processed.*

PROOF. It comes directly from the correctness (Theorem 2) of the MOVIC algorithm, and the uniqueness (Property 1) of the $MCGS$. □

The order independency property ensures that a consistent final result shall be achieved by executing the MOVIC algorithm at all collaborating sites regardless of operation execution orders. Two 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 Figure 8[1], consider the following two different execution orders.

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

Execution Order 1: O_1, O_2, O_3, and O_4.

(1) $MCGS_1 = \{\{O_1\}\}$

(2) $MCGS_2 = \{\{O_1\}, \{O_2\}\}$

(3) $MCGS_3 = \{\{O_1\}, \{O_2\}, \{O_3\}\}$

(4) $MCGS_4 = \{\{O_1, O_4\}, \{O_2, O_4\}, \{O_3, O_4\}\}$

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

(1) $MCGS_1 = \{\{O_1\}\}$

(2) $MCGS_2 = \{\{O_1\}, \{O_2\}\}$

(3) $MCGS_3 = \{\{O_1, O_4\}, \{O_2, O_4\}\}$

(4) $MCGS_4 = \{\{O_1, O_4\}, \{O_4, O_3\}, \{O_2, O_4\}, \{O_4, O_3\}\}$
$\equiv \{\{O_1, O_4\}, \{O_2, O_4\}, \{O_4, O_3\}\}$ $(by\, TR2)$

[1]It should be noted that the conflict relationships between O_i and operations in $MCGS_{i-1}$ 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.

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 TR2. In this way, the final $MCGS_4$ is the same for two different operation execution orders.

Example 2. Consider the four operations O_1, O_2, O_3, and O_4, with their conflict relationships expressed in Figure 9, and the following two different execution orders.

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

Execution Order 1: O_1, O_2, O_3, and O_4.

(1) $MCGS_1 = \{\{O_1\}\}$

(2) $MCGS_2 = \{\{O_1\}, \{O_2\}\}$

(3) $MCGS_3 = \{\{O_1\}, \{O_2, O_3\}\}$

(4) $MCGS_4 = \{\{O_1, O_4\}, \{O_2, O_3, O_4\}\}$

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

(1) $MCGS_1 = \{\{O_1\}\}$

(2) $MCGS_2 = \{\{O_1\}, \{O_2\}\}$

(3) $MCGS_3 = \{\{O_1, O_4\}, \{O_2, O_4\}\}$

(4) $MCGS_4 = \{\{O_1, O_4\}, \{O_4, O_3\}, \{O_2, O_4, O_3\}\}$
$\equiv \{\{O_1, O_4\}, \{O_2, O_4, O_3\}\}$ $(by\ TR2)$

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 removed since it is a subgroup of $\{O_2, O_4, O_3\}$ according to TR2. In this way, the final $MCGS_4$ is the same for two different operation execution orders.

6. OBJECT IDENTIFICATION IN MULTI-VERSIONING SYSTEMS

For the MOVIC algorithm to work, one important parameter has to be provided: the current $MCGS_{i-1}$, on which the new operation O_i is applied to produce $MCGS_i$. The technical issue here is: how to find the CGs in the $MCGS_{i-1}$ for O_i? Since a CG in the $MCGS_{i-1}$ corresponds to an object version made from the original object targeted by O_i, the above issue can be translated into the question of 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 that is able to identify all object versions made from the same original object.

6.1 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 possess the following three properties: (1) *uniqueness:* every object at a site must have a unique identifier; (2) *traceability:* multiple versions made from the same object G must have identifiers which can be traced using the identifier of G; and (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 are distinguishable from each other. The traceability property ensures multiple versions of the same object are 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 are also applied on other replicas.

6.2 Consistency issues in object identification

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 an updating 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., $Target(O) = Id(G)$). When O arrives at a remote site, its parameter $Target(O)$ 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 concurrent 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

[2] There are various ways to determine the Id for an operation O. One way is to use a pair (sid, seq) as an operation identifier, where sid is the identifier of the site at which O is generated, and seq is the sum of the elements of O's state vector timestamp [Sun et al. 1998].

identifier $Id(G)$ $(= Target(O_1))$. To execute O_2, the target object G can still be found by $Id(G)$ $(= Target(O_2))$ 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 disappears. When O_3 arrives with $Target(O_3) = Id(G)$, 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 $G\{O_1\}$ and $G\{O_2\}$ should be identified so that they can be traced by using identifier $Id(G)$.

To address this problem, the simple identification scheme can be extended (1) to let both versions inherit the identifier of the original object so that they are traceable by using $Id(G)$; and (2) to let the identifier of one version include the identifier of the operation which triggers the creation of that new version so that the two versions are distinguishable from each other. Consequently, the identifier of an object can no longer be representable by a single operation identifier but by a *set* of operation identifiers.

For 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., $Id(G\{O_2\}) = Id(G) \cup \{Id(O_2)\}$). 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 the 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 another site in a different order: O_2 followed by O_1. In this scenario, it will be O_1 that triggers the creation of a new version, so $G\{O_1\}$ will take $Id(G) \cup \{Id(O_1)\}$ 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 version $G\{O_2\}$ or $G\{O_1\}$ have been identified differently when the two conflicting operations are executed in different orders.

To solve this problem, the previous identification scheme was 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., $Id(G\{O_1\}) = Id(G) \cup \{Id(O_1)\}$), and $G\{O_2\}$ should take $Id(G)$ plus $Id(O_2)$ as its identifier (i.e., $Id(G\{O_2\}) = Id(G) \cup \{Id(O_2)\}$). 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 were not resolved. Consider three concurrent 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 $Id(G\{O_2\}) = Id(G) \cup \{Id(O_2)\}$. In the meantime, another version $G\{O_1\}$ is identified by $Id(G\{O_1\}) = Id(G) \cup \{Id(O_1)\}$ 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 $Id(G) \cup \{Id(O_1)\}$, and $G\{O_2, O_3\}$ with an identifier of $Id(G) \cup \{Id(O_2)\}$.

However, if the three operations are executed at another site in 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 $Id(G) \cup \{Id(O_1)\}$, and $G\{O_3, O_2\}$ with an identifier of $Id(G) \cup \{Id(O_3)\}$ (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., $Id(G) \cup \{Id(O_3)\}$ and $Id(G) \cup \{Id(O_2)\}$)!

To solve this problem, the previous object identification scheme was further revised to let a version's identifier include the identifiers of all operations (both O_2 and O_3) that are conflicting with another operation (O_1), regardless of which operation triggers the creation of this new version. Finally, it should be stressed that the order of adding a conflicting operation identifier into the object identifier is not significant, so the object identifier must be treated as a *set*, rather than a *list*.

6.3 An object identification scheme

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

SCHEME 1. *(The COID scheme) The identifier of object G consists of a set of operation identifiers:*

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

where $Id(O_i) \in Id(G)$, $1 \leq i \leq n$, if and only if (1) O_i is the operation that 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 $Id(G) := \{Id(O)\}$.
(2) When operation O triggers the creation of a new version G' from the target object G, $Id(G')$ is constructed as $Id(G') := Id(G) \cup \{Id(O)\}$.
(3) When operation O is applied to an existing object G, $Id(G)$ is extended to include $Id(O)$ if O is conflicting with an operation in any other version made from G.
(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 $Id(G') := Id(G') \cup \{Id(O_x)\}$.

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 and hence be identified differently. Moreover, the COID scheme maintains the *consistency* property because for any object, the same set of versions will be replicated at all sites (due to the uniqueness property of MCGS), 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 $Id(G)$.

Based on the COID scheme, when an operation O_i arrives at a remote site, the collection of object versions corresponding to $MCGS_{i-1}$ can be found by checking the identifier of the target object ($Target(O_i)$) against the identifier of each object ($Id(G)$) at that site. The answer to the question raised at the beginning of this section is the following *Target Object VErsion Recognition (TOVER)* scheme.

Scheme 2. *(The TOVER scheme) Given an operation O_i, an object G is a version corresponding to a compatible group CG in the current $MCGS_{i-1}$ if and only if $Target(O_i) \subseteq Id(G)$.*

Based on the TOVER scheme, if the identifier of the target object of an operation is a subset of the identifier of an object version at a remote site, then this operation is applicable to this object version. The collection of object versions to which an operation can be applied to is called the *application scope* of this operation.

7. HANDLING SPECIAL CONFLICT RELATIONSHIPS

In the original definition of operation conflict (Definition 3), one necessary condition for two operations O_1 and O_2 to conflict with each other is $Target(O_1) = Target(O_2)$. If $Target(O_1) \neq Target(O_2)$, then $O_1 \odot O_2$. Algorithms (e.g., MOVIC) based on this conflict definition work well if whenever $Target(O_1) \neq Target(O_2)$, O_1 and O_2 are always applied to different sets of objects. Unfortunately, it could happen that $Target(O_1) \neq Target(O_2)$ but $Target(O_1) \subset Target(O_2)$, which means that O_1 and O_2 are targeting different objects but the target object of O_2 is within the application scope of O_1 (according to the TOVER scheme). Under this circumstance, O_1 and O_2 may be applied to some common objects and incorrect/inconsistent results may be produced if they are always regarded as compatible operations. In this section, we discuss how to extend the original conflict definition to cope with these special conflict relationships.

7.1 $Target(O_1) \subset Target(O_2)$ and $O_1 \otimes_d O_2$

When $Target(O_1) \neq Target(O_2)$ but $Target(O_1) \subset Target(O_2)$, O_1 and O_2 may still conflict with each other under certain conditions.

To illustrate, consider a scenario with four operations O_1, O_2, O_3, and O_4, as shown in Figure 10. The dependence relationships among these four operations can be expressed as: $((O_1 \parallel O_2) \rightarrow O_4) \parallel O_3$, which means that O_1 and O_2 are independent, O_4 is dependent on both O_1 and O_2, and O_3 is independent of O_1, O_2 and O_3. Assume that O_1, O_2, and O_3 are targeting the same object G, i.e., $Target(O_1) = Target(O_2) = Target(O_3) = Id(G)$, and their compatible relationships are: $O_1 \otimes O_2$, $O_1 \odot O_3$ and $O_2 \odot O_3$. Furthermore, assume O_4 is targeting the version $G\{O_2\}$ that is created from G due to the conflict between O_2 and O_1. Concrete examples of O_1, O_2, and O_3 are: $O_1 = Move(G, X)$, $O_2 = Move(G, Y)$, and $O_3 = Color(G, Red)$, where $X \neq Y$. Consider what happened at User 2:

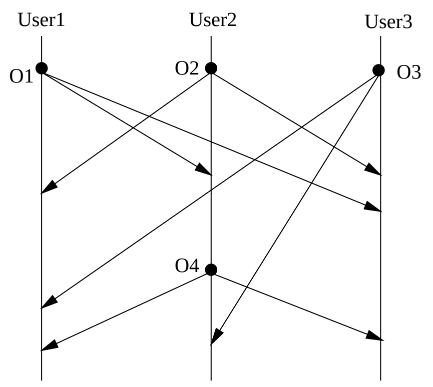

Fig. 10. A scenario for illustrating special conflict relationships between operations originally targeting different objects but applying to common objects.

(1) O_2 is first applied on the target object G to produce $G\{O_2\}$.

(2) O_1 arrives and a new version $G\{O_1\}$ of G is created since $O_1 \otimes O_2$. After executing O_1, object identifiers for the two versions are: $Id(G\{O_1\}) = Id(G) \cup \{Id(O_1)\}$, and $Id(G\{O_2\}) = Id(G) \cup \{Id(O_2)\}$.

(3) Suppose O_4 is generated to target $G\{O_2\}$, i.e. $Target(O_4) = Id(G\{O_2\})$. After executing O_4, $G\{O_2\}$ becomes $G\{O_2, O_4\}$ (but its identifier remains unchanged).

(4) O_3 arrives with $Target(O_3) = Id(G)$, so O_3 may be applied to all versions whose identifiers contain $Id(G)$, including $G\{O_1\}$ and $G\{O_2, O_4\}$ (according to the TOVER scheme). To determine whether O_3 is directly applicable to $G\{O_2, O_4\}$ (in the MOVIC algorithm), O_3 is checked against O_4 for their conflict relationship. Since $Target(O_3) \neq Target(O_4)$, we have $O_3 \odot O_4$ according to Definitions 3 and 4. So, O_3 will be applied to $G\{O_2, O_4\}$. However, since $Target(O_3) \subset Target(O_4)$, both O_3 and O_4 are applied to a common object $G\{O_2, O_4\}$. The trouble occurs when $Att.Key(O_3) = Att.Key(O_4)$, and $Att.Value(O_3) \neq Att.Value(O_4)$. For example, if $O_3 = Color(G, Red)$, and $O_4 = Color(G\{O_2\}, Blue)$, O_3 and O_4 are actually conflicting with each other and should not be applied to the same object version.

To cope with this situation, the conflict definition needs to be extended to check two independent operations for possible conflict as long as their application scopes overlap, regardless of whether they are originally targeting the same object. The extended conflict relation, called *direct conflict*, is defined below.

Definition 9. *Direct conflict relation "$\otimes_d$".* Two operations O_1 and O_2 directly conflict with each other, expressed as $O_1 \otimes_d O_2$, if and only if (1) $O_1 \parallel O_2$; (2) $Target(O_1) \subseteq Target(O_2)$, or $Target(O_2) \subseteq Target(O_1)$; (3) $Att.Key(O_1) = Att.Key(O_2)$; and (4) $Att.Value(O_1) \neq Att.Value(O_2)$.

It should be noted that Condition (2) in the above definition means that if $Target(O_2) \subseteq Target(O_1)$, then the target object of O_1 must be in the application scope of O_2, according to the COID and TOVER schemes.

7.2 $Target(O_1) \subset Target(O_2)$ and $O_1 \otimes_i O_2$

Although the direct conflict definition solves the problem of detecting special conflict relationships for operations with different target objects but with overlapping application scopes, there is still another problem associated with $Target(O_1) \neq Target(O_2)$ but $Target(O_1) \subset Target(O_2)$: inconsistency (divergence) may occur if O_1 and O_2 are executed in different orders at different sites.

To illustrate, consider the scenario in Figure 10 again. This time, O_1, O_2, and O_3 are still targeting the same object G, but their conflict relationships are different: $O_1 \otimes O_2$, $O_2 \otimes O_3$, and $O_1 \odot O_3$. Moreover, we assume that $Att.Key(O_4) \neq Att.Key(O_3)$, so O_4 and O_3 cannot be conflicting (by either Definition 3 or Definition 9). Concrete examples of such four operations are : $O_1 = Move(G, X)$, $O_2 = Move(G, Y)$, $O_3 = Move(G, X)$, and $O_4 = Color(G\{O_2\}, Blue)$, where $X \neq Y$.

At site 1, O_1 is first executed to produce $G\{O_1\}$. Then, O_2 arrives and a new version $G\{O_2\}$ is created due to $O_1 \otimes O_2$. After executing O_2, object identifiers for the two versions are: $Id(G\{O_1\}) = Id(G) \cup \{Id(O_1)\}$, and $Id(G\{O_2\}) = Id(G) \cup \{Id(O_2)\}$. When O_3 arrives with $Target(O_3) = Id(G)$, it may be applied to all versions whose identifiers contain $Id(G)$, including $G\{O_1\}$ and $G\{O_2\}$ (according to the TOVER scheme). However, O_3 is applied only to $G\{O_1\}$ to produce $G\{O_1, O_3\}$ since $O_3 \odot O_1$ and $O_3 \otimes O_2$ (according to the MOVIC algorithm). When O_4 arrives with $Target(O_4) = Id(G\{O_2\})$, it is applied only to $G\{O_2\}$ (resulting in $G\{O_2, O_4\}$) since its application scope contains $G\{O_2\}$ only. The final collection of objects at site 1 is: $\{G\{O_1, O_3\}, G\{O_2, O_4\}\}$.

At site 2, O_2 is first executed to produce $G\{O_2\}$. Then, O_1 arrives and a new version $G\{O_1\}$ is created due to $O_1 \otimes O_2$. After executing O_1, object identifiers for the two versions are: $Id(G\{O_1\}) = Id(G) \cup \{Id(O_1)\}$, and $Id(G\{O_2\}) = Id(G) \cup \{Id(O_2)\}$. Next, O_4 is generated to target $G\{O_2\}$ (i.e., $Target(O_4) = Id(G\{O_2\})$), resulting in $G\{O_2, O_4\}$. When O_3 arrives with $Target(O_3) = Id(G)$, its application scope contains both $G\{O_1\}$ and $G\{O_2, O_4\}$ (according to the TOVER scheme). O_3 is directly applied to $G\{O_1\}$ to produce $G\{O_1, O_3\}$ since $O_1 \odot O_3$, and a new version $G\{O_3, O_4\}$ is created from $G\{O_2, O_4\}$ since $O_3 \odot O_4$ and $O_3 \otimes O_2$ (according to the MOVIC algorithm). The final collection of objects at site 2 is: $\{G\{O_1, O_3\}, G\{O_2, O_4\}, G\{O_3, O_4\}\}$.

Clearly, the final result at site 2 is not identical to the final result at site 1. This divergence occurs due to the fact that O_3 and O_4 are executed in different orders at the two sites. At site 1, O_3 is executed before O_4, so O_3 can be applied to $G\{O_1\}$ only since $O_3 \odot O_1$ but $O_3 \otimes O_2$. At the time of executing O_4, O_4 can be applied to $G\{O_2\}$ only since its application scope does not contain $G\{O_1, O_3\}$. Consequently, O_3 is not able to be combined with O_4 to produce $G\{O_3, O_4\}$. At site 2, however, O_3 is executed after O_4 and is able to be applied to $G\{O_2, O_4\}$ since its application scope contains $G\{O_2, O_4\}$ and $O_4 \odot O_3$, resulting in a new version $G\{O_3, O_4\}$.

To solve this divergence problem, we must ensure all sites produce either the same result as site 1 or as site 2. To achieve the result at site 2 (O_3 is executed after O_4), we have to check O_4 against O_3 for conflict even if O_3 has been executed before O_4 and applied in an object ($G\{O_1, O_3\}$) that is not in the application scope of O_4, which is a very complicated task if at all possible. To achieve the result at site 1, on the other hand, we only need to introduce a new conflict relation, called *indirect conflict*, so that O_3 and O_4 can be regarded as indirectly conflicting with each other and will not be combined to produce a new version $G\{O_3, O_4\}$. The reason that O_3 and O_4 are regarded as indirectly conflicting because O_3 is directly conflicting with another operation O_2, which is combined with O_4 in a common version $G\{O_2, O_4\}$. We have chosen to use the indirect conflict concept to resolve this divergence problem because it is simple and can avoid the creation of additional versions (e.g., the additional version $G\{O_3, O_4\}$ in the previous example). The definition of the indirect conflict relation is given below.

Definition 10. *Indirect conflict relation "$\otimes_i$".* Two operations O_1 and O_2 indirectly conflict with each other, expressed as $O_1 \otimes_i O_2$, if and only if (1) $O_1 \parallel O_2$; (2) $Target(O_1) \subset Target(O_2)$; (3) $Target(O_2)$ contains an $Id(O_x)$, such that $O_x \otimes_d O_1$.

By applying the above definition to the previous example, we have $O_4 \otimes_i O_3$ because: (1) $O_4 \parallel O_3$, (2) $Target(O_3) \subset Target(O_4)$; (3) $Target(O_4)$ contains $Id(O_2)$, such that $O_2 \otimes_d O_3$. Furthermore, we impose the following operation execution rule: two operations are prohibited from being combined in one object version if they are indirectly conflicting. By applying this rule to the previous example, O_3 will not be combined with O_4 at site 2 since $O_4 \otimes_i O_3$. In this way, all sites will have consistent (convergent) results regardless of the execution orders of O_4 and O_3.

Finally, the direct and indirect conflict definitions are combined in the following revised conflict relation definition.

Definition 11. *Revised conflict relation "$\otimes$".* Two operations O_1 and O_2 conflict with each other, expressed as $O_1 \otimes O_2$, if and only if either $O_1 \otimes_d O_2$, or $O_1 \otimes_i O_2$.

Under the revised conflict relation definition, the original definition of a compatibility relation (Definition 4) is still valid. The MOVIC algorithm can adopt the revised conflict relation definition without any revision since direct and indirect conflict operations are handled in the same way. However, the COID scheme needs a minor revision: the identifier of an operation is added to an object's identifier only if this operation is *directly* conflicting with at least one other operation. This is because any pair of indirectly conflicting operations are always associated with a pair of directly conflicting operations. Indirectly conflicting operations can always be accommodated by the object versions created by the corresponding directly conflicting operations, and never cause the creation of additional object versions.

8. CONVERGENT LAYERING FOR OVERLAPPING OBJECTS

Graphics objects may overlap with each other: concurrently created objects may overlap at the time of creation, non-overlapping objects may become overlapped when they are modified (e.g., resized, or moved), and multiple versions of an object may overlap with each other or with other existing objects. In the previous sections, we have devised algorithms like MOVIC to ensure that after executing the same collection of operations, all collaborating sites maintain the same collection of graphics objects. In this section, we propose one additional convergent layering scheme to ensure that, in the presence of concurrency and conflict, all overlapping objects are layered in the same order at all sites.

8.1 Graphic objects list

The key data structure for supporting convergent layering is a *Graphic Objects List (GOL)* at each site. The GOL contains the list of graphic objects created so far. The system paints the objects in the GOL in the order from left to right whenever there is a need to re-paint the graphic interface. Therefore, if an object at $GOL[i]$ is overlapping with an object at $GOL[j]$ and $i < j$, $GOL[i]$ will be layered under $GOL[j]$. The objective of the convergent layering scheme is to ensure the $GOLs$ at all sites are identical, i.e., they contain the same list of objects in the same order.

8.2 Total ordering among objects

The basic idea to achieve this objective is to define a total ordering relationship among all objects so that they can be stored in the GOL according to their total ordering relationship. Since every object is associated with a collection of operations applied to it and all operations have a total ordering relationship "$\Rightarrow$" determined by their state vector timestamps and their site identifiers [Sun et al. 1998], we can use the total ordering relationships among operations to derive the total ordering relationships among objects.

We differentiate two kinds of objects in the system: one is single-version objects which are directly created by the *Create* operation, the other is multi-version objects which are created due to operation conflict. Since there is one-to-one correspondence between a single-version object and its *Create* operation, the total ordering relationship among single-version objects can be defined by the total ordering relationship among their corresponding *Create* operations.

Definition 12. *Total ordering among single-version objects.* Given two single-version objects G_1 and G_2 created by O_1 and O_2, respectively, G_1 is totally ordered before G_2, denoted by $G_1 \Rightarrow G_2$, if and only if $O_1 \Rightarrow O_2$.

Based on the above definition, for any pair of single-version objects G_1 and G_2, if $G_1 \Rightarrow G_2$, then G_1 and G_2 will be stored in $GOL[i]$ and $GOL[j]$, respectively, where $i < j$. Since the total ordering relationship among all single-version objects are the same at all sites, they must be stored in the same order in $GOLs$ at all sites.

A single-version object may spawn several versions when conflicting operations are applied to it. So, we need to extend the total ordering relationship from single-version objects to multi-version objects.

Suppose two concurrent operations O_1 and O_2 which move an existing object G to two different but overlapping positions, resulting in the creation of two object versions $G\{O_1\}$ and $G\{O_2\}$. To order these two object versions created due to conflict, certain questions need to be answered: (1) *global ordering:* how should the two versions be ordered with respect to other existing objects in the GOL?

(2) *relative ordering:* how should the two versions be ordered with respect with each other?

Since the two new versions are replacing the original object, the answer to the first question is simply to assign the global ordering of the original object to the new versions. That means, $G\{O_1\}$ and $G\{O_2\}$ will be inserted into the GOL at the position originally occupied by G. The answer to the second question is to determine the relative ordering of the two new versions according to the total ordering relationship among the operations applied to them. Therefore, if $O_1 \Rightarrow O_2$, then $G\{O_1\} \Rightarrow G\{O_2\}$.

One complication arises when there are multiple compatible operations applied to each object version. To illustrate, consider four concurrent operations O_1, O_2, O_3, and O_4, targeting the same object G. Suppose they have the conflict relationships expressed by the CRT in Figure 11:

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

Fig. 11. A CRT for illustrating problems in total ordering among multi-version objects.

According to the MOVIC algorithm, after executing these four operations, the final result would be two object versions: $G\{O_1, O_2, O_3\}$ and $G\{O_1, O_4\}$. Suppose these two versions are overlapping in space, and the total ordering relationships among the four operations are: $O_1 \Rightarrow O_2 \Rightarrow O_4 \Rightarrow O_3$. To determine the ordering relationship between these two object versions, if O_4 from $G\{O_1, O_4\}$ is compared with O_2 from $G\{O_1, O_2, O_3\}$, we have $G\{O_1, O_2, O_3\} \Rightarrow G\{O_1, O_4\}$ since $O_2 \Rightarrow O_4$. However, if O_4 is compared with O_3 from $G\{O_1, O_2, O_3\}$, we have $G\{O_1, O_4\} \Rightarrow G\{O_1, O_2, O_3\}$ since $O_4 \Rightarrow O_3$. Which relative ordering relationship should the two versions have?

To unambiguously determine the relative ordering, the following scheme is proposed: (1) sort all operations in each compatible group in their total ordering; then (2) compare the corresponding pair of operations one by one (from the oldest to the youngest) in two compatible groups to determine their relative ordering. For $G\{O_1, O_2, O_3\}$, we have $O_1 \Rightarrow O_2 \Rightarrow O_3$; and for $G\{O_1, O_4\}$ we have $O_1 \Rightarrow O_4$. The relative layering order among the two versions should be $G\{O_1, O_2, O_3\} \Rightarrow G\{O_1, O_4\}$ since $O_1 = O_1$ and $O_2 \Rightarrow O_4$. In general, to capture both the global and the relative ordering relationships among multi-version objects created from an existing object, we have the definition of total ordering among multi-version objects below.

Definition 13. *Total ordering among multi-version objects.* Given two object versions G_1 and G_2 created due to conflicting operations on the same original object G, their ordering is determined as follows:

(1) *Global ordering:* For any existing G_x in the GOL, $G_x \Rightarrow G_1$ and $G_x \Rightarrow G_2$ if and only if $G_x \Rightarrow G$.

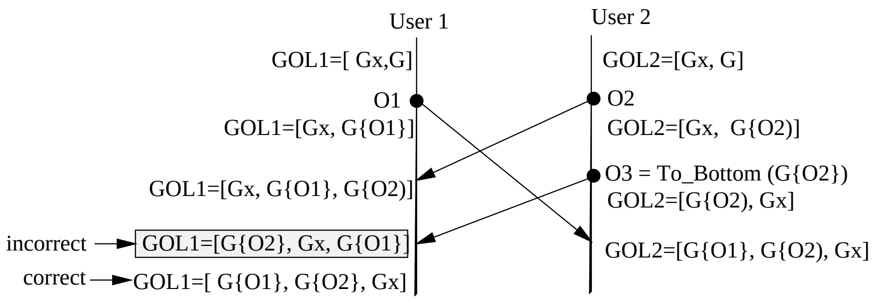

Fig. 12. Dealing with concurrent *To_Bottom* and updating operations.

(2) *Relative ordering:* Suppose OL_1 is the list of operations applied to G_1 and OL_2 is the list of operations applied to G_2. Suppose operations in OL_1 and OL_2 are sorted according to their total ordering relationships. $G_1 \Rightarrow G_2$ if and only if there is a k $(1 \leq k \leq |OL_1|)$, such that $OL_1[k] \Rightarrow OL_2[k]$ and $OL_1[i] = OL_2[i]$ for all $i < k$.

When a new object is created by a *Create* operation, this object is placed at a proper position in the GOL according to Definition 12. When a new object is created due to conflict, it will be placed at the place determined by Definition 13. When the same group of operations have been executed at all sites, the GOLs at these sites must be identical since (1) GOLs at all sites contain the same collection of objects according to the MOVIC algorithm, and (2) all objects are totally ordered in the same way according to Definitions 12 and 13.

8.3 Changing objects layering

The layering of an object G can be changed by a special operation $To\_Bottom(G)$, which moves object G to the bottom of all objects in the GOL i.e., the left-most position of the GOL. For example, suppose the current objects layering at a site is: $GOL = [G_1, G_2, G_3]$. After executing $To\_Bottom(G_2)$ at this site, the new objects layering becomes: $GOL = [G_2, G_1, G_3]$.

The use of $To\_Bottom(G)$ may cause layering divergence when a *To_Bottom* operation is targeting an object which spawns multiple versions due to the execution of other concurrent operations. For example, consider the scenario in Figure 12. There are three operations O_1, O_2, and O_3, and their concurrency relationship can be expressed as: $O_1 \parallel (O_2 \rightarrow O_3)$. Suppose the initial graphic object lists at the two sites are: $GOL_1 = GOL_2 = [G_x, G]$; O_1 and O_2 are targeting the same original object G in a conflicting way (i.e., $O_1 \otimes O_2$); and O_3 is a *To_Bottom* operation targeting object $G\{O_2\}$, which has the same identifier as object G. At site 2, after executing O_2, $GOL_2 = [G_x, G\{O_2\}]$. Then, $O_3 = To\_Bottom(G\{O_2\})$ is generated to move $G\{O_2\}$ to the bottom and GOL_2 becomes $[G\{O_2\}, G_x]$. Finally, O_1 arrives to create a new version $G\{O_1\}$ and GOL_2 becomes $[G\{O_1\}, G\{O_2\}, G_x]$ (assume $O_1 \Rightarrow O_2$). At site 1, after executing O_1, $GOL_1 = [G_x, G\{O_1\}]$. Then, O_2 arrives to create a new version $G\{O_2\}$, and $GOL_1 = [G_x, G\{O_1\}, G\{O_2\}]$. Finally, O_3 arrives to move $G\{O_2\}$ to the bottom and GOL_1 becomes $[G\{O_2\}, G_x, G\{O_1\}]$ (as shown in a rectangular box in Figure 12). Clearly, $GOL_1 \neq GOL_2$.

This layering divergence problem can be solved by defining a To_Bottom operation as being compatible with all concurrent operations that are targeting the same object. Based on this definition, the MOVIC algorithm will apply the To_Bottom operation to all object versions created by these concurrent operations. In the previous example, since $Target(O_3) = Id(G\{O_2\}) = Id(G)$ at the time when O_3 was generated at site 2 (according to the COID scheme), the application scope of O_3 should be $\{G\{O_1\}, G\{O_2\}\}$ at the time when O_3 arrived at site 1 (according to the TOVER scheme). Since the To_Bottom operation O_3 is compatible with the concurrent operation O_1 (i.e., $O_3 \odot O_1$), the MOVIC algorithm will apply O_3 to both $G\{O_1\}$ and $G\{O_2\}$ to move both versions to the bottom of the object list, i.e., $GOL_1 = [G\{O_1\}, G\{O_2\}, G_x]$, as shown in Figure 12.

When multiple users issue concurrent To_Bottom operations to move different objects to the bottom, all these objects will be moved to the left of the GOL and the relative layering among these objects will be determined by the total ordering relationships among these concurrent To_Bottom operations.

Similar to the $To\_Bottom(G)$ operation, another operation $To\_Top(G)$ is also available to move object G to the top of all objects in the GOL, i.e., the right-most position of the GOL. For the same reason, the $To\_Top(G)$ operation is defined as being compatible with all concurrent operations that are targeting the same object G.

9. THE GRACE PROTOTYPE SYSTEM

All algorithms and schemes presented in this article have been implemented in an Internet-based GRACE (GRAphics Collaborative Editing) prototype system in the programming language Java. The prototype GRACE system architecture resembles the architecture of the (text-oriented) REDUCE (REal-time Distributed Unconstrained Cooperating Editing) system [Sun et al. 1998], where multiple collaborating sites, typically a PC machine or a workstation, are directly connected via TCP connections over the Internet. Each collaborating site runs a GRACE process which takes care of operation generation, processing and propagation and document management. The major difference between GRACE and REDUCE is that GRACE has a graphics editing interface (as shown in Figure 13) and uses the multi-versioning technique for consistency maintenance, whereas REDUCE has a text editing interface and uses operational transformation for consistency maintenance.

A GRACE process consists of multiple threads, with one special thread for handling local operations generated from the user interface, and one thread dedicated to each remote site for handling operations propagated from that site. At the core of a GRACE process is the GRACE Engine (GE) object which maintains the key data structures (e.g., the state vector, the operation *History Buffer* (HB), and the *Graphics Object List* (GOL)) and implements consistency maintenance algorithms. GE is implemented as a Java *monitor* object to provide the multiple threads with a synchronized point of access to the shared internal resources and the external user interface.

The implementation of the multi-versioning algorithms and schemes follows the definitions and descriptions in this article in a straight forward way. When a

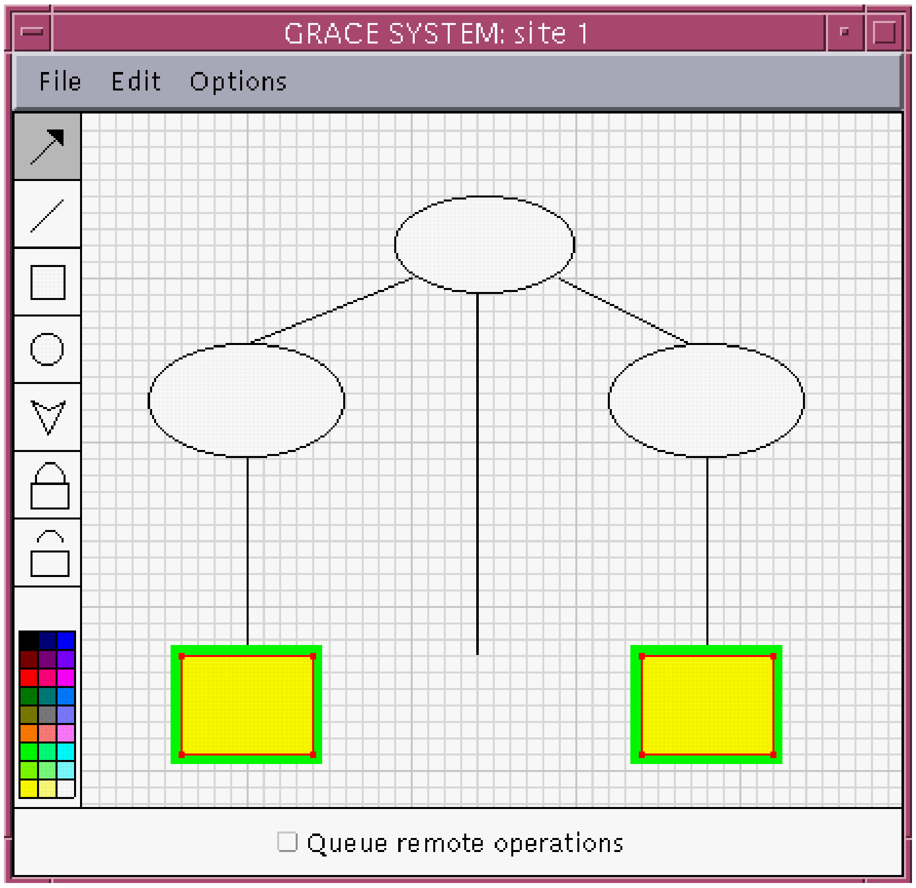

Fig. 13. The GRACE graphics editing interface.

local operation is generated, it is executed immediately on the local interface, timestamped by the current value of the local state vector, saved in the local HB, and propagated to all remote sites. All graphics objects are assigned distinctive identifiers according to the COID scheme and saved in the GOL. Each object in the GOL refers to the collection of operations in the HB which have been applied to it.

When a remote operation arrives and becomes causally ready for execution, the collection of objects in the application scope of this operation are selected from the GOL by executing the TOVER scheme. Then, the MOVIC algorithm is executed to check this operation against each object in its application scope to determine the combined effects. In the execution of the MOVIC algorithm, the conflict relationships between this remote operation and all concurrent operations referred by each object in the application scope are examined according to Definition 11. The execution of a remote operation may create new object versions due to conflict, and the new object versions (if any) are placed in the GOL according to the total ordering relationship (Definition 13). Whenever there is a need to re-paint the user interface, objects in the GOL are painted in their order in the GOL from the left to the right to ensure convergent layering (see Section 8).

On the user interface, object versions created due to conflict are specially highlighted to allow users to differentiate the objects created by concurrent *Create* operations from objects created due to conflict resolution. As shown in Figure 13, the two rectangles are highlighted since they were created by two conflicting operations which moved a single rectangle (in the middle of the two versions, not shown in this figure) to two different positions.

After the execution, the remote operation is saved in the HB. When an operation in the HB is no longer needed for future use, it is removed from the HB as garbage. An operation in the HB becomes garbage when it is causally before all future operations. The same garbage collection algorithm designed for REDUCE [Sun et al. 1998] has been directly used in GRACE.

The GRACE system was designed for use in the Internet environment, in which operations with arbitrarily complex concurrency and conflict relationships could be naturally generated because of the long and non-deterministic communication delay in such an environment. Before using the system in the Internet environment, we have to test it in a Local Area Network (LAN) environment, where communication latency is low and messages are automatically serializable by the underlying broadcasting transmission media. Without special measures, conflicts due to concurrency rarely occur in the LAN environment and many scenarios commonly observable in the Internet environment (e.g., the scenario in Figure 10) are hardly detectable in the LAN environment. To facilitate the testing of multi-versioning algorithms in the LAN environment, we have built into the system a communication delay mechanism to delay any remote operation for an arbitrary period of time (controllable from the user interface). By means of this mechanism, we are able to generate a group of operations with arbitrary concurrency and conflict relationships and to observe the system's behavior under any particular scenario.

The current GRACE prototype system has been developed mainly to test the feasibility of the multi-versioning 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.

10. DISCUSSIONS

In this section, we compare the work reported in this article with alternative approaches in the graphics editing domain and related work in the text editing domain.

10.1 Comparison within the graphics editing domain

Most existing collaborative graphics editing systems have adopted a conflict prevention approach based on locking. Example systems based on locking include: Aspects [Biel 1991], Ensemble [Newman-Wolfe et al. 1992], GroupDraw [Greenberg et al. 1992], and GroupGraphics [Pendergast 1995]. In these systems, the 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, a coordinating process is used in these systems to keep track of which objects have been locked so that permissions for locking requests can be granted/denied accordingly. The problem with this kind of 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 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. In GroupDraw, locally generated operations are executed right away and a lock request message is sent to the coordinating process. If the coordinating process does not approve the lock request, then the effect of that operation is undone, which may cause anomaly at the user interface.

In contrast to conflict prevention approaches like locking, the multi-versioning strategy proposed in this article allows conflict to occur and provides a conflict resolution mechanism (i.e., the multi-versioning technique) to accommodate all operations effects in a consistent way. Major advantages of this approach include helping to achieve high responsiveness and to preserve the work concurrently produced by multiple users in the face of conflict. However, locking does have the merit of preventing conflicts from happening. Locking is actually complementary with optimistical concurrency control strategies like multi-versioning. In fact, we have proposed a novel *optional* locking scheme (in contrasting to existing *compulsory* locking schemes) to enhance the consistency maintenance capability of the system in the text editing domain [Sun 2002]. The design and integration of an optional locking scheme with the multi-versioning approach in the graphics editing domain is one of our ongoing research tasks [Chen 2001].

Another alternative conflict resolution approach is *serialization*, which ensures that the effect of executing a group of concurrent operations is the same as if these operations were executed in the same total order at all sites. With this approach, if there is any conflict among concurrent operations, only the effect of the last operation (in the total ordering) will be maintained. Examples of such systems are: GroupDesign [Karsenty et al. 1993] and LICRA [Kanawati 1997]. This approach is essentially the *single-operation-effect* approach discussed in Section 2, a major problem of which is that the system randomly decides which operation's effect to keep, which may not be what the collaborating users really want. In our all-operations-effect approach, users can choose to keep all versions (if that is desirable), to keep one version but remove other versions (i.e., a single-operation-effect), or to remove all versions (i.e., an null-effect). It is conceivable to support multiple conflict resolution policies on top of the all-operations-effect approach and to allow users to select the suitable conflict resolution policy according to the application context. For example, if the single-operation-effect or null-effect approach is what the users really want in a particular application context, then the multi-version-based system can automatically remove unwanted versions according to the user-selected conflict resolution policy. How to support multiple conflict resolution policies on top of the multi-version-based system is an interesting topic of future research.

The Tivoli whiteboard meeting-support tool developed at Xerox PARC [Moran et al. 1995] is most closely related to our work in the graphics editing domain. 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 the GRACE approach is that in Tivoli, a conflict occurs whenever two concurrent operations target the same object, whereas in GRACE, 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 (by our 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. Another limitation with Tivoli is that it was able to support only two collaborating sites in a session [Moran et al. 1995]. In contrast, the GRACE system is able to support an arbitrary number of sites in a collaborative editing session and to minimize the number of object versions by combining the effects of compatible operations. The technical issues and solutions reported in this article have never been addressed by Tivoli or any other collaborative graphics editing system.

There are still a number of areas in the proposed multi-versioning technique that deserve further investigation, optimization, and extension. First, in the current GRACE system, an object identifier is represented by a set of operation identifiers (the COID scheme), and all objects at a site are stored in a linear graphic object list (GOL). When an operation is propagated to a remote site, its target object identifier is propagated as well. For each remote operation, the system has to search the GOL from left to right and uses the operation's target object identifier to select the objects in this operation's application scope (the TOVER scheme). For each object in the remote operation's application scope, the MOVIC algorithm has to check the compatibility relationships between the remote operation and all concurrent operations applied on this object in order to determine the correct combined effects. These general and simple implementation strategies for object identification, storage, and searching, and for checking operation compatibility work well for systems like GRACE, where the number of users in a session is small (typically 2 to 5 people), the number of objects in the GOL is small, and conflicts are rare. However, these strategies may not scale well to systems with a very large number (e.g. millions) of objects concurrently manipulated by a large number of users. Further research is needed for a formal evaluation of the time and space complexity of the MOVIC algorithm and the COID and TOVER schemes and for optimizing these algorithms.

Second, the conflict definition in this article implicitly assumes that one operation targets only one object, attributes of an object are independent, and objects in a system are independent. These assumptions impose limitations on the operation and object types supportable by the current multi-versioning technique. Future research will extend the conflict definition and the multi-versioning technique to support interrelated object attributes, interrelated objects, and operations capable of targeting multiple objects [Bharat and Hudson 1995; Campbell 2000].

Last but not least, future research will be directed toward user interface issues related to the multi-versioning technique. Novel interface techniques, such as *hint* *highlighting* in the Gamut system [McDaniel and Myers 1999], will be investigated for better displaying object versions on the interface to help users recognize and manage object versions. In addition, novel *version-merging* facilities will be investigated to help users to merge multiple versions according to their collaboration needs.

10.2 Comparison between the graphics and the text editing domains

In this subsection, we compare the work on collaborative graphics editing systems reported in this article with our previous work on collaborative text editing systems reported in [Sun et al. 1998] from consistency maintenance point of view.

The same consistency model with three properties – causality preservation, intention preservation, and convergence – has been applied to REDUCE in the text editing domain and GRACE in the graphics editing domain. Moreover, the same causality preservation technique based on vector clock timestamping has been used in both REDUCE and GRACE because causality preservation relies only on the dependency relationship among operations, which is independent of whether these operations are text or graphics editing operations. However, two different techniques – operational transformation and multi-versioning – have been devised in REDUCE and GRACE, respectively, for achieving intention preservation and convergence. Why are two different techniques needed? What is the relationship between them? The answers to these questions are related to the differences between REDUCE and GRACE in their ways of identifying and updating objects.

For REDUCE in the text editing domain, all data objects (i.e., characters) have a linear ordering relationship. Therefore, a text document can be naturally modeled as an one-dimensional array of characters, and each object can be uniquely identified by its position or its index in this one-dimensional array. Text editing operations use the index identifier to refer each character in the document. Inserting/deleting characters into/from a text document may shift the positions (and change the index identifiers) of the characters in the one-dimensional array. To achieve intention preservation and convergence, the index identifier of an operation has to be adjusted (or transformed) according to the impact of previously executed concurrent operations. This is why operational transformation was necessary for REDUCE. Text documents in REDUCE are plain text documents. Apart from *insert* and *delete*, no attributes updating operations are supported in the current REDUCE system.

In the graphics editing domain, on the other hand, the relationship among graphic data objects is application-dependent: objects may be independent of each other as in GRACE, or related to each other in an Entity-Relation diagram [Campbell 2000]. Therefore, it is not natural to organize graphic objects into an one-dimensional array and to identify them by their index positions in this array. Graphic objects are normally identified by specially designed identifiers, which are independent of each other. As shown in Section 6, a sophisticated object identification scheme is needed in a multi-versioning system like GRACE. Consequently, creating/deleting objects into/from the graphics document has no impact on the identifiers of other objects in the document, and there is no need to update object identifiers due to concurrent execution of create/delete operations. This is why operational transformation is not necessary for GRACE. In fact, if characters in a text document were uniquely identified by independent identifiers (rather than dependent indices) in REDUCE, operational transformation would have been unnecessary for REDUCE either. However, there are two major reasons against this way of identifying characters in text documents. Unlike the coarser-grain graphic objects, characters are too fine-grain objects (representable by one byte), and the overhead to associate each character with an independent identifier (requiring at least one integer) is prohibitive. In addition, since a text document may easily contain a large number of characters, searching the whole document for the right character targeted by an editing operation is significantly slower than simply indexing into a linear array. This is why REDUCE has used a simple object representation and identification scheme based on the linear relationship among characters but has used a sophisticated operational transformation technique to dynamically adjust index-based identifiers for characters targeted by operations.

Why is it necessary to have the multi-versioning technique for collaborative graphics editing systems like GRACE? If GRACE supported only *create* and *delete* operations, corresponding to the *insert* and *delete* operations in REDUCE, consistency maintenance in GRACE would have been trivial – concurrent operations can be executed in any order without the need for either operational transformation or any other technique in achieving intention preservation and convergence (but a convergent layering scheme is needed for overlapping objects). However, GRACE allows users to change the attributes of existing graphic objects and concurrent updating operations may conflict with each other in an arbitrary way. The multi-versioning technique is needed to achieve intention preservation and convergence in the face of conflict. Although the multi-versioning technique was devised in the context of GRACE, it is generally suitable for systems which support concurrently updating object attributes. In fact, if characters in REDUCE documents had attributes, such as font, size, style, color, and the like, then concurrent attributes updating operations might also conflict with each other and the multi-versioning technique might be used in REDUCE for conflict resolution as well.

In summary, multi-versioning and operational transformation, though motivated and invented in two different editing domains, are two complementary techniques for achieving intention preservation and convergence. Their suitability to a particular system depends on the data representation and identification method and the nature of operations in the system. Operational transformation is suitable for systems in which data objects are organized into an one-dimensional array and identified by their indexes in the array, and users are allowed to generate operations to insert/delete objects into this array concurrently. Multi-versioning is suitable for systems in which data objects may be identified in any way, and users are allowed to generate operations to update object attributes concurrently. Integrating these two techniques in one collaborative editing system, such as a word processor, is one of our future research tasks.

11. CONCLUSIONS AND FUTURE WORK

In this article, we have proposed and discussed a novel multi-versioning approach to consistency maintenance for real-time collaborative graphics editing systems. A generic consistency model with three consistency properties – causality preservation, convergence, and intention preservation – has been used as a framework for addressing consistency maintenance issues in the graphics editing domain. Since causality preservation is independent of application domains, this article has focused on the issues and solutions related to achieving convergence and intention preservation in collaborative object-based graphics editing systems.

We started by analyzing and defining conflict and compatibility relationships among graphics editing operations. To achieve intention prevention in the face of conflict, a multi-versioning approach was proposed to resolve conflict by accommodating all operations effects in multiple versions of the same object. Furthermore, three Combined Effect Rules (CER1–3) for determining the correct combined operation effects have been derived: CER1 requires conflicting operations to be applied in different object versions, CER2 requires each pair of versions made from the same object to have at least one pair of operations (one in each version) conflicting with each other, and CER3 requires compatible operations to be applied in common object versions. This multi-versioning approach is novel since it achieves intention preservation and preserves the work concurrently produced by multiple users in the face of conflict, and minimizes the number of object versions created for conflict resolution.

A major technical challenge in supporting the multi-versioning approach is how to determine the correct and convergent combined effects for an arbitrary group of operations at all sites. Our research has found that a unique combined effect complying with CER1–3 can be derived from the inherent conflict relationships among any group of operations targeting the same object. To capture the conflict relationships among any group of operations, we have defined the conflict relation matrix and triangle, the compatible groups set, the rules for transforming one compatible groups set to another equivalent set, and the Maximal Compatible Groups Set ($MCGS$). Moreover, we have proven the uniqueness of the $MCGS$, specified the combined effect based on the $MCGS$, and verified that this combined effects comply with CER1–3. The formal specification of a unique combined effect for an arbitrary group of operations by their $MCGS$ is one of the major contributions of this work.

Another major technical contribution of this work is the design of the MOVIC (Multiple Object Versions Incremental Creation) algorithm, which is able to incrementally construct the $MCGS$ for any group of operations, regardless of the order in which these operations are executed. We have proven the correctness of the MOVIC algorithm and its order independence property, which guarantees consistent construction of multiple versions of the same object across all collaborating sites. To support the MOVIC algorithm, we have devised the COID (Consistent Object IDentification) scheme that is able to trace all object versions made from the same original object. In addition, the TOVER (Target Object VErsion Recognition) scheme has been devised to determine the correct object versions belonging to the application scope of any operation based on the identifier of the target object of this operation and the identifiers of existing objects at a remote site.

We have studied the conflict relationships among operations targeting different original objects but with overlapping application scopes. The concepts of direct conflict and indirect conflict have been introduced to solve inconsistency problems caused by these special conflict relationships. Moreover, a convergent layering scheme has been devised to ensure that, in the presence of concurrency and conflict, all objects appear in the same layering order at all sites. The successful implementation of all algorithms and schemes in this article in the Internet-based GRACE prototype system has proven the feasibility of the multi-versioning technique and allowed us to explore some system design and implementation issues.

Comparison between this work and related work in both graphics editing domain and text editing domain has shown that multi-versioning is a unique and novel technique for consistency maintenance in collaborative graphics editing systems and is generally suitable for any system which supports concurrent update of object attributes. Our comparison study has also found that multi-versioning and operation transformation are two complementary techniques for achieving intention preservation and convergence. We are currently working on the integration of the multi-versioning and operational transformation technologies into one collaborative editing system which supports both graphics and text editing of XML-based documents.

Conflict resolution, based on either multi-versioning or other techniques, is not the sole method for dealing with conflict in collaborative editing systems. Conflict avoidance based on mechanisms which provide group-awareness information on other users' editing locations, activities, and intentions can be useful to avoid conflict. Conflict prevention based on locking can also be useful to reduce conflict [Sun 2002]. Furthermore, flexible *undo* facilities [Sun 2000; Chen and Sun 2001] capable of rolling the document back to any previous status and novel *version-merging* facilities capable of combining the valuable work done on different versions into integrated versions can be very useful for users to better resolve conflict according to their collaboration needs.

Most existing work on consistency maintenance in collaborative editing systems, including the work reported in this article, has focused on issues and techniques related to *syntactic consistency*, which is mainly concerned with whether operations are executed in the right order (causality preservation) and whether the same (convergent and intention preserved) view of the shared document is maintained at all sites. Future research should put more efforts on exploring the issues and techniques related to *semantic* consistency [Dourish 1996; Edwards 1997], which is concerned with whether the same view of the shared document is maintained at all sites as well as whether the common view makes sense or not in the application context. Semantic consistency maintenance relies on syntactic consistency maintenance as well as on the availability of application-dependent semantic information. A major challenge here is that application-dependent semantic information is very hard (if not impossible) to extract from primitive operations. To maintain semantic consistency, new mechanisms are needed to support the interaction and collaboration between the user and the system in providing and utilizing application-dependent semantic information in collaborative environments.

Consistency maintenance research in groupware has drawn inspiration from traditional distributed computing techniques, such as causal/total ordering of events, and vector logical clocks timestamping, and has also invented non-traditional techniques, such as multi-versioning and operational transformation. The generalization and application of these novel groupware techniques to other areas of distributed computing and CSCW is an exciting direction for future exploration. We are currently working on applying the consistency maintenance techniques in GRACE and REDUCE to collaborative CAD and CASE systems, and expanding our research on consistency maintenance from discrete interactive media to continuous interactive media, particularly Internet-based multi-player gaming applications.

ACKNOWLEDGMENTS

We wish to thank the anonymous referees and the Associated Editor, Brad A. Myers, for their very insightful and constructive comments and suggestions which have helped improve the presentation of the article.

REFERENCES

Baecker, R. 1992. *Readings in groupware and computer-supported cooperative work.* Lawrence Erlbaum Associates, Morgan Kaufmann Publishers Inc.

Bernstein, P., Goodman, N., and Hadzilacos, V. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, Reading, Massachusetts.

Bharat, K. and Hudson, S. E. 1995. Supporting distributed, concurrent, one-way constraints in user interface applications. In *ACM Symposium on User Interface Software and Technology.* ACM, New York, 121–132.

Biel, V. 1991. Groupware grows up. *MacUser*, 207–211.

Birman, K., Schiper, A., and Stephenson, P. 1991. Lightweight causal and atomic group multicast. *ACM Trans. on Comp. Sys. 9,* 3 (August), 272–314.

Campbell, J. D. 2000. Consistency maintenance for real-time collaborative diagram development. Ph.D. thesis, Department of Information Science, University of Pittsburgh, Pittsburgh, PA, USA.

Chen, D. 2001. Consistency maintenance in collaborative graphics editing systems. Ph.D. thesis, School of Computing and Information Technology, Griffith University, Brisbane, Qld 4111, Australia.

Chen, D. and Sun, C. 1999. A distributed algorithm for graphic object replication in real-time group editors. In *Proceedings of ACM Conference on Supporting Group Work.* ACM, New York, 121–130.

Chen, D. and Sun, C. 2001. Undoing any operation in collaborative graphics editing systems. In *Proceedings of ACM Conference on Supporting Group Work.* ACM, New York, 197–206.

Cheshire, S. 1996. Latency and the quest for interactivity. In *White paper commissioned by Volpe Welty Asset Management, L.L.C., for the Synchronous Person-to-Person Interactive Computing Environments Meeting.*

Dourish, P. 1996. Consistency guarantees: exploiting application semantics for consistency management in a collaborative tookit. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work.* ACM, New York, 268–277.

Edwards, W. K. 1997. Flexible conflict detection and management in collaborative applications. In *ACM Symposium on User Interface Software and Technology.* ACM, New York, 139–148.

Ellis, C., Gibbs, S., and Rein, G. 1991. Groupware: some issues and experiences. *CACM 34,* 1 (January), 39–58.

Fidge, C. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference.* 56–66.

Greenberg, S. and Marwood, D. 1994. Real time groupware as a distributed system: concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work.* ACM, New York, 207–217.

Greenberg, S., Roseman, R., and Webster, D. 1992. Issues and experiences designing and implementing two group drawing tools. In *Proceedings of the the 25th Annual Hawaii International Conference on the System Science*. 139–250.

Kanawati, R. 1997. LICRA: a replicated-data management algorithm for distributed synchronous groupware application. *Parallel Computing 22*, 1733–1746.

Karsenty, A. and Beaudouin-Lafon, M. 1993. An algorithm for distributed groupware applications. In *Proceedings of the 13th International Conference on Distributed Computing Systems*. 195–202.

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

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *CACM 21,* 7, 558–565.

Li, D. 2000. Coca: A framework for modeling and supporting flexible and adaptable synchronous collaborations. Ph.D. thesis, Department of Computer Science, University of California, Los Angeles, USA.

McDaniel, R. G. and Myers, B. A. 1999. Getting more out of programming-by-demonstration. In *Proceedings of ACM Conference on Human Factors in Computing Systems*. 442–449.

Moran, T., McCall, K., van Melle, B., Pedersen, E., and Halasz, F. 1995. Some design principles for sharing in tivoli, a whiteboard meeting-support tool. In *Groupware for Real-time Drawings: A Designer's Guide*, S. Greenberg, Ed. McGraw-Hill International(UK), 24–36.

Newman-Wolfe, R., Webb, M., and Montes, M. 1992. Implicit locking in the ensemble concurrent object-oriented graphics editor. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*. ACM, New York, 265–272.

Nichols, D., Curtis, P., Dixon, M., and Lamping, J. 1995. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of ACM Symposium on User Interface Software and Technologies*. ACM, New York, 111–120.

Pendergast, M. 1995. Groupgraphics: prototype to product. In *Groupware for Real-time Drawings: A Designer's Guide*, S. Greenberg, Ed. McGraw-Hill International(UK), 209–227.

Raynal, M. and Singhal, M. 1996. Logical time: capturing causality in distributed systems. *IEEE Computer 29,* 2 (Feb.), 49–56.

Sun, C. 2000. Undo any operation at any time in group editors. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*. ACM, New York, 191–200.

Sun, C. 2002. Optional and responsive fine-grain locking in Internet-based collaborative systems. *To appear in IEEE Trans. on Parallel and Distributed Systems*.

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

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

Received May 2001; revised October 2001; accepted December 2001