Conflict Control Locking in Distributed Cooperative Graphics Editors

Liyin Xue
Department of computing
Macquarie University
Sydney, NSW 2109, Australia
lyxue@ics.mq.edu.au

Kang Zhang
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083-0688, USA
kzhang@utdallas.edu

Chengzheng Sun
School of Comp & Info Tech
Griffith University
Brisbane, QLD 4111, Australia
C.Sun@cit.gu.edu.au

Abstract

The communication latency presents a major challenge to achieving high responsiveness for Internet-based cooperative editing systems. In this paper, we propose conflict control locking (post-locking) schemes for conflict resolution in real-time object-based cooperative graphics editors. With these schemes, no locking request is needed before applying an editing operation to an object, instead, a lock will automatically be generated by the system if a conflict occurs. Lock ownership assigning rules are devised for the schemes. With a specific post-locking approach, algorithms and implementation techniques, such as, lock synchronisation detection, voting, and group intention realisation, are discussed in detail.

Keywords: Concurrency control, cooperative graphics editors, multi-versioning, CSCW, distributed computing.

1 Introduction

Distributed real-time cooperative editors allow a group of users to view and edit the same document at the same time from geographically dispersed sites connected by communication networks in general, the Internet in particular. They are very useful facilities in advanced Computer Supported Cooperative Work (CSCW) applications, such as, cooperative CAD/CASE.

In a distributed environment with non-deterministic latency, a replicated architecture is usually adopted for the storage of shared documents, in order to meet the requirement for high responsiveness [3, 14]. The major challenge of supporting unconstrained cooperative editing is the management of multiple streams of concurrent activities so that document consistency can be maintained in the course of conflicts. Various concurrency control techniques and algorithms have been proposed to tackle this issue.

Locking is a widely used concurrency control technique either in database systems or cooperative applications. Conflicts can be prevented by locking, but locking is undesirable because it interrupts users in their work and

causes unnecessary overhead, that is, a locking operation has to be generated before any editing operations.

Due to the restrictiveness of locking approaches, some cooperative editing systems dispense any locking altogether [5, 16]. These systems rely upon social protocols and users' awareness of others' actions to prevent conflicts, and hope that if conflicts do occur they can be quickly and easily resolved. Their major assumption is the number of collaborators is small and each user can instantly see changes made by other users to shared objects (i.e., so called What You See Is What I See: WYSIWIS). This approach is responsive and unconstrained. However, it may fail in the following two situations [10]. First, as either the document grows in size or the number of collaborators increases, the WYSIWIS property is difficult to maintain. Second, the quality of awareness that an application can provide is highly sensitive to the delay in delivering one user's actions to other users. Thus applications that rely on awareness to prevent conflicts may not function well in an environment with high-latency communications systems, such as the Internet.

This paper examines the concurrency control problem associated with object-based cooperative graphics editors. In these systems, objects such as line, circle, etc., can be created, modified or deleted. Each object is represented by attribute type and value pairs. Attribute types consist of colour, size, position, group, etc. After an object has been created, operations, such as, move, resize, fill, can be applied to change attribute values of that object.

Conflict prevention is preferred to cure in the systems where a locking approach is adopted. In highly responsive and unconstrained cooperative editing systems, we prefer "cure" to "prevention", with an assumption that users are cooperative and conflicts are rare. We take an optimistic approach, where no locking request is needed before any editing operations. If conflicts do occur, supporting mechanisms will be provided to help users resolve them. That is, we take a risk and provide risk management techniques at the mean time, without leaving the users to suffering from the pain.

In this paper, we propose a class of conflict management schemes, called *conflict control locking*, for object-based cooperative graphics editors. Conceptually, it is a fine-grained extension of conventional version control approaches in multi-user environments [6]. Technically, it is based on a *multi-versioning approach* which has been proposed recently [1, 17].

The rest of the paper is organised as follows. Section 2 briefly introduces some background information and definitions. Then, we present lock ownership assigning rules for different conflict control locking schemes in Section 3. Next algorithms and implementation techniques for a specific conflict control locking scheme are discussed in Section 4. The conflict control locking approaches are compared to related work in Section 5. Finally, major results are summarised and further work is discussed in Section 6.

2 Previous Work

In this section, the basic concepts and definitions adopted in this paper are briefly presented. For details, the reader is referred to [1, 17, 18].

2.1 A Consistency Model

A cooperative editing system is said to be consistent if it always maintains the following three properties: convergence, causality-preservation and intention-preservation.

Causality-preservation can be achieved by using a distributed algorithm [18]. In object-based cooperative editors, achieving convergence is a relatively simple issue, due to the commutability of primitive operations. The intention of operation O is the execution effect that can be achieved by applying O on the document state from which O was generated. Intention-preservation is supported by the multi-versioning approach.

2.2 The Multi-versioning Approach

If two concurrent operations attempt to change the same attribute of an object to different values, then conflict occurs. Operations with this type of relationship are called conflicting operations. It is impossible for the system to accommodate conflicting intentions in the same target object. The only way to preserve the intentions of both operations is to make two replica objects from the original object, and then apply the two conflicting operations to the two replicas separately. The resulting effect is that no conflicting operations are applied to the same replica or version. If two operations are not conflicting, then they are compatible. The effects of compatible operations can be applied to the same object without causing intention violation. In a highly concurrent real-time cooperative editing environment, a group of operations may have rather arbitrary and complex relationships, their combined

replication effect is a set of versions (VS). Each version is the result of executing a set of compatible operations. All operations applied to the same version are compatible. For any two different versions, there exists at least one operation in one version that is conflicting with one or more operations in the other version. Therefore the multi-versioning approach can preserve users' conflicting intentions in a consistent way.

Any created versions are distinguished by unique identifiers assigned during replication so new operations can be generated to edit specific versions. Each operation can be uniquely identified by an identifier, and any object version is identified by a set of operation identifiers. Obviously it is not necessary to include the identifier of any operation which is compatible with all other operations applied to the object in conflicting into the versions' identifiers.

3 Conflict Control Locking Schemes

In this section we discuss the role of conflict control locking in cooperative editing systems, and present lock ownership assigning rules for the conflict control locking schemes proposed in this paper.

3.1 The Role of Conflict Control Locking

There are mainly two categories of locking schemes in the existing group editors. One is *pessimistic locking*: to ensure consistency, before an operation can be generated to edit an object, a lock on that object must be obtained. This will guarantee that only one user, the lock owner, can edit an object at a time. However, users are blocked between the time a lock is requested and the time it is granted. The other is *optimistic locking*, in which a user is permitted to edit an object while waiting for the requested lock. If the locking request is indeed successful, the user is able to continue editing the object. If the locking request finally fails, the user is not allowed to continue editing this object, and what this user has done during waiting for the lock will be undone. Due to its non-blocking and responsive nature, optimistic locking is regarded better suited to an environment where communication latency is high but conflicts are rare [4].

The optimistic locking category can still be divided into two sub-categories, i.e., *compulsory locking* and *optional locking*. Compulsory locking means that a lock must be requested (either explicitly or implicitly) before updating an object. A locking scheme is optional in the sense that a user may update any (unlocked) object without necessarily requesting a lock on it [2,19]. If a lock has been placed on an object, however, a user can update this locked object only if s/he holds a lock on it. When multiple users are editing different regions, which is the most common case in cooperative editing, editing without requesting locks

avoids locking overhead and does not cause any inconsistency problem. Only when multiple users may edit the same object of a document, should the users use locking before updating.

All existing locking schemes have two things in common. First, locking is conflict preventing. Second, a locking request (if it is required as in optional locking) is issued before an editing operation in order to get a lock, whether explicitly by a user or implicitly by the system, either compulsory or optional. They can be categorised as *conflict-prevention-locking* or *pre-editing-locking*, in short, *pre-locking*. In contrast to existing locking schemes, the locking schemes proposed in this paper are *conflict-control-locking* or *post-editing-locking*, in short, *post-locking*. Within the schemes, users need not issue a lock request before editing an object. Instead, locking will automatically be enforced by the system only when two or more concurrent operations target a same object, leading to a conflict. In the meantime, the users will be notified of the conflict. This lock may be shared by two or more users, depending on who have issued the concurrent operations and what kind of conflict control locking schemes are adopted. No further operations from other users except the owners are allowed to target the object (replicated versions).

Instead of preventing conflicts, the post-locking schemes aim at controlling conflicts only if they occur. While the schemes can be made non-blocking and highly responsive as with optional locking, they completely avoid any pre-locking overhead.

Unlike pre-locking schemes, locking is generated implicitly by the system in post-locking schemes. Locking operations are executed only on the local copies of the document in a distributed way. Each site locks the involved objects automatically. Therefore, timestamp and propagation are not needed for locking operations.

The conflict control locking schemes are based on the multi-versioning approach [17]. Without such an approach, it is not possible to preserve individual users' conflicting intentions, although conflicts can still be controlled. When a conflict occurs, interested users can adjust their actions properly, according to further negotiation or coordination within the group. For example, a voting procedure can be invoked, and group intended operations are selected and applied to the document, finally the conflict is resolved.

The multi-versioning approach was introduced to facilitate the version creation process, aiming at preserving individual users' intentions in a consistent way. The conflict control locking schemes are introduced in this paper to *restrict the number of versions that can be created, to manage the created versions, to facilitate the resolution of the conflicts, and to preserve a group intention*.

3.2 Free Editing Approach

In order to categorise conflict control locking schemes, we need to introduce lock ownership. Let $Owner(V)$ denotes a set of users who own the lock on object version V . Let $|Owner(V)|$ denotes the number of users who owns the lock on V . A lock on a version V is purely exclusive if $|Owner(V)| = 1$; a lock on a version V is shared if $|Owner(V)| > 1$.

In terms of lock ownership of various versions of an object, conflict control locking schemes can be classified into four sub-categories: *systemic (or representative) post-locking approach*, *shared exclusive post-locking approach*, *shared inclusive post-locking approach*, and *free editing approach*.

For the *free editing approach*, every user own the locks by definition, i.e. for all versions V_i ($i = 1, \dots, m$) in the current version set, $Owner(V_i) = \{U_i | i = 0, \dots, N-1\}$, where U_i denotes the user at site i , and N the number of participating sites. To put it in another way, there is no lock at all. Although the multi-versioning approach provides a mechanism to accommodate conflicting operations in a consistent way, conflict resolution is entirely left to the users. The users can further edit any versions. In an extreme, an infinite number of versions may be created. To the knowledge of authors, all current systems adopt this approach implicitly, if they allow multiple object versions. This approach places the burden of conflict resolution completely on to the users, without providing any further facility. It will be difficult for the users to reach an agreed upon result, i.e. *group intention*.

3.3 Systemic Post-locking Approach

With the *systemic post-locking approach*, no user except the system (local application) owns the lock, i.e. for all V_i ($i = 1, \dots, m$), $Owner(V_i) = \{the\ system\}$. When the first conflict occurs at a site, two versions of the targeted object will be created. Then the system locks both of them. After that, the user at that site is not allowed to generate any further operation targeting either of the object versions. Due to communication delays, operations may arrive in different orders at different sites. Newly arrived operations may be independent of both or one of the conflicting operations, they should be applied to the current versions in a proper way according to the multi-versioning approach. Any new versions created are locked by the system. It will take a period of time, which we call *lock synchronisation period*, for each site to reach a stable set of versions, which are locked and globally consistent. The system's locking operation reduces the complexity caused by multiple version creation to a minimal level. This approach is the simplest and most useful one. The major disadvantage is that all users have to wait for the synchronisation of locking before they can start to resolve the conflicts. Fortunately, this is acceptable, since the

users can still edit other objects that are not locked during this period of time.

As soon as the lock is synchronised, the system can pass the lock ownership on to a representative of the involved users, who may be the leader or supervisor of the group or sub-group (users who have generated operations targeting the object). This user now represents the group and has a purely exclusive lock on all the versions, i.e., for any V_i ($i = 1, \dots, m$), $Owner(V_i) = \{the\ representative\}$. S/he may edit or delete them to realise a group intention (presumably and implicitly), and then unlock the intended version(s). An alternative to the *lock transferring* is that the system invokes a voting procedure and the conflicts are resolved by an explicit group intention. Further discussion of this approach is presented in the next section.

3.4 Shared Post-locking Approaches

Since only when a user completes all her/his intended operations on an object, can others fully understand her/his intention and make an intelligible decision, the systemic post-locking approach seems to be too restrictive. Here come the shared inclusive and shared exclusive post-locking approaches, which make it feasible. The major issue is who should have the right to further edit one or more of the versions.

Consider the simplest case where two conflicting operations O_1 and O_2 , generated by users U_1 and U_2 respectively, target the same object A . Two versions of the object will be created, i.e., V_1 and V_2 , where O_1 and O_2 have been applied respectively. Both of them are locked by the system, and we assume $U_1 \in Owner(V_1)$, $U_2 \in Owner(V_2)$. If there are no more operations independent of them, then it seems straightforward to take an approach that U_1 and U_2 have an exclusive lock on V_1 and V_2 respectively, i.e., $Owner(V_1) = U_1$; $Owner(V_2) = U_2$. No users can further edit any one of the versions except its owner. This is an example of the lock ownership assigning scheme adopted in the shared exclusive post-locking approach. Nevertheless, some of the other users may argue that they have the same intention as U_1 or U_2 , and want to participate in the further editing. So the scheme seems too restrictive to them. In this case, all the other users who have not generated any operations targeting the same object should be added to both of the owner sets, i.e. $Owner(V_1) = U - \{U_2\}$, $Owner(V_2) = U - \{U_1\}$, where U denotes the set of participating users. This is a special case of the scheme adopted in the shared inclusive post-locking approach.

If there is a set of independent operations, denoted by $OS = \{O_i \mid i = 1, \dots, m\}$, generated by some of the participating users, $U = \{U_i \mid i = 0, \dots, N-1\}$, targeting the same object, then two or more versions may be created, $VS = \{V_i \mid i = 1, \dots, n\}$. One user's operation may be applied to one or more versions of the targeting object. It depends

on the compatible and conflicting relationships among the operations. For the shared inclusive post-locking approach, the *lock ownership assigning rules* are devised as follows:

- If U_i generated an operation O_j such that $O_j \in OS$, then for any version $V_k \in VS$ which O_j is applied to, $U_i \in Owner(V_k)$;
- If U_i did not generate any operation in OS , then U_i owns the locks for all versions in VS , i.e. $U_i \in Owner(VS)$.

A similar strategy was adopted for optional locking [2]. Here a user can further edit any version unless s/he has generated an operation that is conflicting with one or more of the operations applied to it. In this sense it is inclusive. For the shared exclusive post-locking approach, the above rules are revised as follows:

- If U_i generated an operation O_j such that $O_j \in OS$, and the identifier of O_j was added to any version $V_k \in VS$, then $U_i \in Owner(V_k)$.

It is exclusive in the sense that only those users who have contributed an operation to a version, which is conflicting with one or more other operations in the OS , own the lock on the version, others are excluded. It is more restrictive than the inclusive one.

For any one of the above mentioned approaches, the more restrictive it is, the easier for the users to resolve the conflicts. In the case of either of the shared post-locking approaches is adopted, a user may not obtain an exclusive lock on a version, and no one has an exclusive lock on all versions. Therefore, other coordination techniques are needed. For example, a procedure can be provided to allow some users to give up their lock ownerships, so others may obtain an exclusive lock on a version. We call it *lock abstaining*. An alternative to the lock abstaining technique is *lock delegating*, where a user may transfer her/his ownership to others, eventually someone (e.g. the group leader) may get an exclusive lock on every versions, and the conflicts are easily resolved. Further discussions on them are beyond the scope of this paper.

In summary, when conflicts occur, if no user is allowed to further edit any versions of the object, then the systemic post-locking approach is adopted; if every user is allowed, then the free editing approach is adopted. Between the two extremes come the shared post-locking approaches. The characterising difference between the shared exclusive and the shared inclusive post-locking approaches is whether any user who does not generate any conflicting operation targeting the object should be excluded from or included into the owner sets of any object versions. Any user who owns a lock on a particular version can further edit it without being blocked.

4 Algorithms for Implementing Systemic Post-locking

In this section, we discuss algorithms and implementation techniques for the systemic post-locking approach.

4.1 Basic Data Structures

For implementing post-locking, each site needs to maintain a number of data structures, such as *Version history list*, *History Buffer*, *State Vector*, *Locking Table*, and *State Vector Table*.

A *History Buffer (HB)* is needed to keep track of executed editing operations [18].

If an object is in conflicting, each version of it maintains a list of pointers, called *Version History List (VHL)*, which point to operations applied to the version in the history buffer. $VHL(G)$ denotes the list of compatible operations applied to version G .

Locking Table (LT) is introduced to keep track of all the conflicts happened in the whole document. Each entry in the *LT* represents a locked object and its replicated versions. All entries in the *LT* together represent the current locking status of the document. The fields in each entry include: identifiers of all current versions of the object in conflicting, pointers to every version history lists, voters, state vectors and originating site identifiers of the first couple of conflicting operations, and a locking status indicator (*LSI*). Voters are a group of users whose operations have been applied to the locked versions of an object.

Locking status indicator (*LSI*) represents each object's locking status:

- *Synchronising* (conflicts occurred and there may be more concurrent operations coming),
- *Synchronised* (no more concurrent operations targeting the object),
- *Voting* (in the process of conflict resolution), and
- *AgreedUpon* (a group consensus is reached).

To enforce the locking effect, any local editing operations must pass permission check before their execution. These permission checks rely on the information saved in the local *LT*. If an operation's targeting object is in the *LT*, then it is not allowed to execute, and the user will be notified. Any causally ready remote operation received is executed without validation, for it has passed the permission check at its originating site. The result of permission check is obtained instantly without delay from synchronisation with other sites. In fact, such a useless effort of users can be avoided by a properly designed user interface.

A *State Vector (SV)* is used at each site to keep track of the number of executed (global) operations from each site. It has N elements $SV[j]$ ($j = 0, \dots, N-1$), which represents

the number of operations generated at site j and executed locally.

State Vector Table (SVT) was introduced for implementing a *garbage collection scheme*, which periodically removes any operation O_x from the history buffer where all operations independent of O_x have been applied, to avoid the history buffer to become infinitely long [18].

A State Vector Table consists of N state vectors, one per site. Let SVT_k be the state vector table at site k . Initially, $SVT_k[i][j] := 0$ for all $i, j \in \{0, \dots, N-1\}$. After executing an operation O from a remote site r , timestamped by SV_O , the r th vector of SVT_k is updated as follows: $SVT_k[r][i] := SV_O[i]$, for all $i \in \{0, \dots, N-1\}$. Obviously, $SVT_k[i]$ at site k represents the view (or knowledge) of site k about the state at site i , for any $i \in \{0, \dots, N-1\}$. In particular, $SVT_k[k] = SV_k$, i.e., the k th vector of SVT_k is the local state vector.

To support lock synchronisation detection, we append a *Minimum State Vector (MSV)* at the end of the *SVT* [18]. It can be considered as a *virtual site* identified by the integer N being added to the system. The $(N+1)$ th state vector (i.e., *MSV*) of the *SVT* reflects the number of operations which have been executed at every sites. Initially $SVT_k[N][j] := 0$ for all $j \in \{0, \dots, N-1\}$. After executing an operation and updating other elements of the SVT_k , site k updates $SVT_k[N][j]$ as follows: $SVT_k[N][j] := \min(SVT_k[0][j], \dots, SVT_k[N-1][j])$, for all $j \in \{0, \dots, N-1\}$ [15].

If the value of the i th element in $SVT_k[N]$ becomes m , i.e., $SVT_k[N][i] = m$, then the first m operations generated at site i must have been executed at all sites. In other words, if an operation is executed at the virtual site, it must have been executed at all sites.

4.2 Lock Synchronisation Detection

A lock is synchronised if no more forthcoming operations are independent of the operations that have been applied to the locked object versions. In systemic post-locking approach, we assume when a conflict occurs the targeted object (or replicated versions created from it) is automatically locked. That means: if two operations, O_a and O_b , which are the first couple of conflicting operations targeting an object G , are executed at site i , then no more operations will be generated at site i to target the replicated versions created from G . It should be noted that both O_a and O_b could be generated by any sites. Although the first couple of conflicting operations at site i may not be the same as at site j ($i \neq j$), the lock is considered synchronised at site i if it has the knowledge that its first couple of conflicting operations have been executed at all sites. Therefore if both of the conflicting operations O_a and O_b have been executed at the virtual site, then the lock is considered synchronised, though some sites may not yet know that.

Based on the above discussion, a lock synchronisation detection procedure is devised as follows:

Algorithm 1: The Lock synchronisation detection procedure

- (1) Scan the *LT* from top to bottom, skip the synchronised entries (i.e. *LSI* $\neq$ *Synchronising*), for each remaining entry in the *LT*, suppose the state vectors of the first couple of conflicting operations are SV_1 and SV_2 , and their originating site identifiers are i and j .
- (2) If $SV_1[i] \leq SVT[N][i]$ and $SV_2[j] \leq SVT[N][j]$, then the lock on the object versions of the current entry is considered synchronised. Update the *LSI* to *Synchronised*, and notify the user. Otherwise the lock is not yet synchronised.
- (3) Continue scanning the *LT*, until no entry left in the *LT*.

The lock synchronisation detection procedure is invoked after processing each remote operation or message. We can see that each site perceives the lock synchronisation time based on its own knowledge about the states of other sites in a completely distributed fashion.

4.3 Conflict Resolution by Voting

As soon as a lock is synchronised, the system may transfer the lock to a representative of the sub-group (i.e. voters). S/he can then resolve the conflicts by exclusive locking. Here we assume that a more democratic technique is adopted, i.e., a *voting procedure* will be invoked to get a group intention. Generally it is triggered when the lock is synchronised. It is worth pointing out that a remote voting message may come before the local application is aware of that the lock is synchronised. In this case the locking status indicator is updated to *Synchronised* by the remote voting message.

Algorithm 2: The voting procedure

- (1) Update the *LSI* to *Voting*.
- (2) Wait for the voting messages from the voters. If more than half of them, for example, choose the same version, then a group intention is reached. Update the *LSI* to *AgreedUpon*, and invoke the *group intention realisation procedure*.
- (3) Otherwise, voting fails, the local user (if s/he is a voter) is notified and a new voting process is initialised. This process repeats until a consensus is reached.

It should be noted that voting is a global operation, thus it is time-stamped. As soon as the user (voter) chooses one of the versions, this decision message is broadcasted to all the other sites.

4.4 Group Intention Realisation

If an agreement is reached among the group, only one version of the object in conflicting is kept, all unwanted

versions are deleted, and the corresponding operations in the history buffer can be collected as garbage.

Algorithm 3: The group intention realisation procedure

- (1) Delete all versions of the object in conflicting except the agreed upon one from the local document.
- (2) Remove the operations in the *HB*, to which the pointers of all *VHLs* of the object point; then delete the *VHLs*.
- (3) Finally, remove the corresponding lock entry from the locking table.

4.5 Garbage Collection

For any executed operation *EO* in *HB*, generated at site i and time-stamped by SV_{EO} , if $SV_{EO}[i] \leq m$ ($= SVT_k[N][i]$), then *EO* must be one of the first m operations generated at site i and hence will certainly causally precede (thus compatible with) all forthcoming operations to site k . Therefore, *EO* can be removed from *HB*. Based on this, a garbage collection procedure has been devised [18].

However, if an operation has been detected conflicting with other operations, then it should not be collected as garbage, even if all operations that are independent of it have been executed at the local site. It will be collected only after the conflict is resolved. Therefore the garbage collection procedure should be revised to reflect the introduction of a multi-versioning approach.

Algorithm 4: The garbage collection procedure

Scan *HB* from left to right. Let *EO* be the current operation under inspection. Suppose that *EO* was generated at site i and time-stamped by SV_{EO} ,

- (1) If $SV_{EO}[i] \leq SVT_k[N][i]$ and $Tgt(EO)$ is not in the *LT*, then remove *EO* from *HB*, and continue scanning.
- (2) Otherwise stop scanning and return.

Where $Tgt(EO)$ denotes the target object of *EO*.

5 Comparison to Related Work

Various locking schemes have been proposed to maintain consistency in cooperative graphics editors. Locking in all existing systems is of conflict prevention. Example systems based on locking include: Aspects[20], Ensemble[12], GroupDraw[5], and GroupGraphics[13]. In these systems, a user has to acquire a lock on an object before editing it, thus introducing unnecessary locking overheads, and preventing other users from expressing their intentions on the same object.

In contrast to conflict prevention locking, the conflict control locking approaches proposed in this paper allow conflicts to occur. They provide mechanisms to manage and resolve the conflicts in order to realise a group intention. Therefore the pre-locking overheads are eliminated completely and consequently the

responsiveness characteristic of single user editors is preserved.

Optional locking has been introduced to avoid the pre-editing locking in the most common case cooperative editing, where locking is only occasionally needed [2, 19]. Nevertheless, it is still on the track of conflict prevention locking. A major problem for it is that there is no reasonable criterion for a user to decide when to request such a lock. An intelligent agent may be introduced to generate optional and implicit locks and eliminate the burden placed on the users, but it adds an extra complexity to the system. In fact, fine-grained optional locking is unnecessary in the sense that most of its functionality can be realised by conflict control locking schemes. However, optional locking may be useful in implementing user defined flexible access control policies.

Locking is a global operation in all pre-locking approaches, whereas locking operations are executed only on the local copies of the document in a distributed fashion in the post-locking approaches, thus time-stamping and propagation are not needed.

Due to the assumption that conflict is rare because people mediate themselves, there are a number of systems that have adopted optimistic approaches without any pre-editing locking. The major difference among these systems is the way and degree a computer can support the human users in their resolution of rare conflicts. They can be categorised into two approaches, i.e., *manual conflict resolution* and *automatic conflict resolution*.

With the automatic conflict resolution approach, local operations can be executed as soon as they are generated to give a quick response. Before an operation from a remote site is executed, it must be checked against executed operations for possible conflicts. If a conflict is detected, either *serialisation* or *merge matrix* mechanisms is used to determine which operation's effect will appear without human users' intervention. The serialisation mechanism is adopted by GroupDesign [8] and LICRA [7], where a total ordering between concurrent operations (or users) is used to determine the final effect, i.e. if operation O_a totally precedes operation O_b , then O_b 's effect will overwrite O_a 's. The merge matrix mechanism is implemented in SYNC [11]. A merge matrix consists of rows and columns labeled by the operations for an object. The merge matrix entry identified by a particular row operation and column operation determines the action the system will take – typically accepting one operation or the other. SYNC provides default merge matrix settings, which can be configured, even be replaced at runtime. SYNC can support any number of users, but only in an asynchronous mode of merging.

A consequence of automatic conflict resolution approach is that whenever there is a conflict, only one user's intention can be preserved. The representative post-

locking approach proposed in this paper preserves the representative's intention, but this intention may not necessarily be her/his own work. In fact it is a group intention resulting from an informed assessment of the conflicting situation with multiple versions of an object presented. Automatic resolution approach has the advantage of being efficient. Nevertheless, it is generally infeasible for the system to have the knowledge to properly resolve conflicts among concurrent users. Conflicts are best resolved by cooperative users, with the system providing explicit information about users' actions.

The multi-versioning approach has been adopted in systems, such as Tivoli [9] and GRACE [1, 17]. Both of them rely on the users manually resolving the conflicts without providing any further support except a conflict awareness mechanism, i.e. highlighting the multiple versions of the same object. They belong to the free editing approach mentioned earlier in this paper. The users don't know how long a (lock) synchronisation process will last in a non-deterministic communication network, even if they abide by a social protocol not to further edit the versions. When the version creation process ends, there may be more than one user trying to resolve the conflicts, thus causing further conflicts.

The approaches proposed in this paper can be categorised as *system assisted conflict resolution*. The system provides facilities, such as *locking*, *locking status indicating*, *lock abstaining*, *lock delegating*, and *voting*, to support the users to reach a consistent group intention. To our knowledge, the technical issues and solutions reported in this paper have never been addressed by any other work.

6 Conclusions and Future Work

In this paper, we have proposed conflict control locking (post-locking) schemes for conflict resolution and group intention preservation in real-time object-based cooperative graphics editors. With these schemes, no locking request is needed before applying an editing operation to an object, instead, a lock will automatically be generated by the system if a conflict occurs. Thus locking overhead resulting from any conflict prevention locking (pre-locking) approaches is completely avoided. In order to preserve users' conflicting intentions, a multi-versioning approach is adopted. The post-locking schemes are introduced to restrict the number of versions can be created, to manage the created versions, to facilitate the resolution of the conflicts, and to preserve a group intention. We classify post-locking schemes into three sub-categories, i.e. systemic (representative) post-locking, shared exclusive post-locking, and shared inclusive post-locking (here free editing approach is not counted). All of them are fully distributed and highly responsive. Lock ownership assigning rules have been devised for each of

them. With the systemic post-locking approach, algorithms and implementation techniques, such as, lock synchronisation detection, voting, and group intention realisation, are discussed in detail.

Although conflicts can be resolved with the post-locking approaches, group awareness and conflict awareness mechanisms are needed to help minimise the chance of conflict. The user interface issues related to post-locking, such as conflict presentation, locking status and lock ownership indication, should be further investigated.

Although our discussions are based on a graphics editing scenario, apparently the schemes presented in this paper are general in the sense they can be applied to any object-based cooperative editors. The generalisation and application of them to other areas of distributed computing and CSCW is a direction for future exploration.

We are in the process of devising more elaborate and flexible post-locking techniques for conflict resolution in cooperative graphics editors. Work is underway to apply post-locking approaches to cooperative text editors.

References

- [1] D. Chen and C. Sun. A distributed algorithm for graphic objects replication in real-time group editors. In *Proc. of the International ACM SIGGROUP Conference on Supporting Group Work*, Nov. 1999, pp.121--130.
- [2] D. Chen and C. Sun. Optional and responsive locking for object-based collaborative graphic editors. In *ACM SIGGROUP Bulletin*, 20(3), December 1999.
- [3] P. Dewan. Architectures for collaborative applications. In M. Beaudouin-Lafon (ed.), *Computer supported Cooperative Work*, John Wiley & Sons, 1999, pp.169--193.
- [4] S. Greenberg and D. Marwood. Real time groupware as a distributed system: concurrency control and its effect on the interface. In *Proc. ACM Conference on Computer Supported Cooperative Work*, November 1994, pp.207--217.
- [5] S. Greenberg, M. Roseman, and D. Webster. Issues and experiences designing and implementing two group drawing tools. In *Proc. of the 25th Annual Hawaii International Conference on the System Sciences*, January 1992, pp.139--250.
- [6] G. E. Kaiser. Cooperative transactions for multi-user environments. In W. Kim (ed.), *Modern Database Systems: the object model, interoperability, and beyond*, ACM Press, 1994, pp.409--433.
- [7] R. Kanawati. LICRA: A replicated-data management algorithm for distributed synchronous groupware application. In *Parallel computing*, vol. 22, 1997, pp.1733--1746.
- [8] A. Karsenty, C. Tronche, and M. Beaudouin-Lafon. Groupdesign: shared editing in a heterogeneous environment. In *Usenix Journal of Computing Systems*, 6(2), 1993, pp.167--195.
- [9] T. P. Moran, K. McCall, B. van Melle, E. Ronby Pedersen, and F.G.H. alasz. Some design principles of sharing in Tivoli, a white-board meeting support tool. In S. Greenberg, S. Hayne, and R. Rada (eds.), *Groupware for Real-time Drawing: A Designer's guide*, McGraw-Hill, 1995, pp.24--36.
- [10] J. P. Munson and P. Dewan. A concurrency control framework for collaborative systems. In *Proceedings of ACM CSCW'1996*, pp. 278--287.
- [11] J. P. Munson and P. Dewan. Sync: a Java framework for mobile collaborative applications, In *IEEE Computer*, June 1997.
- [12] R. E. Newman-Wolfe, M. L. Webb, and M. Montes. Implicit locking in the Ensemble concurrent object-oriented graphics editor. In *Proc. ACM Conference on Computer Supported Cooperative Work*, November 1992, pp. 265--272.
- [13] M. O. Pendergast. GroupGraphics: prototype to product. In S. Greenberg, S. Hayne, and R. Rada (eds.), *Groupware for Real-time Drawing: A Designer's guide*, McGraw-Hill, 1995, pp.209--227.
- [14] A. Prakash. Group editors. In M. Beaudouin-Lafon (ed.), *Computer Supported Cooperative Work*, John Wiley & Sons, 1999, pp.103--133.
- [15] M. Raynal and M. Singhal. Logical time: capturing causality in distributed systems. In *IEEE Computer Magazine*, 29(2), February 1996, pp.49--56.
- [16] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS revisited: early experiences with multi-user interfaces. In *ACM Transactions on Office Information Systems*, 5(2), April 1987, pp.147--167.
- [17] C. Sun and D. Chen. A multi-versioning approach to conflict resolution in distribute groupware systems. In *Proceedings of International Conference on Distributed Computing Systems*, April 2000.
- [18] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. In *ACM Transactions on Computer-Human Interaction*, 5(1), March 1998, pp.63--108.
- [19] C. Sun and R. Sosič. Optional locking integrated with operational transformation in distributed real-time group editors. In *Proceedings of ACM 18th Symposium on Principles of Distributed Computing*, Atlanta, USA, May 1999, pages 43--52.
- [20] von Biel. Groupware grows up. In *MacUser*, June 1991, pp.207--211.