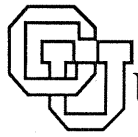
A Model and Algorithm for Concurrent Access within Groupware

Clarence A. Ellis

CU-CS-593-92 April 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Model and Algorithm for Concurrent Access within Groupware*

by

Clarence A. Ellis
MCC Software Technology Program
Austin, Texas, USA

Abstract

Groupware supports a group of users engaged in a common task. This paper introduces some groupware concurrency concepts and requirements, and presents a new model which has been useful in understanding concurrency issues, as well as many other issues of groupware architecture. The model is then applied to verify the correctness of a distributed algorithm which maintains consistency without locking, and without rollback, within a dynamic non-serializable environment.

An important aspect of the paper is that it introduces a novel verification technique that allows a designer to look at static graph theoretic properties of a groupware system to determine consistency of its dynamic distributed operation. The model, the algorithm, and the verification technique are applied to the GROVE group editor which was implemented at MCC.

1. Introduction

Groupware aims to assist groups in communicating, in collaborating, and in coordinating their activities. Groupware can be defined as computer based systems that support two or more users engaged in a common task or goal, and that provide an interface to a shared environment. The groupware group at MCC's software technology program has been researching groupware and computer supported cooperative work for the past five years. A number of prototypes have been produced, measurement and modeling of those prototypes has occurred, and lessons learned have lead to various theories and models of the resultant processes and systems.

These systems can be categorized as real-time groupware versus non-real-time groupware. Examples of real-time groupware are real time group editors, video conferencing systems, and group decision support (electronic meeting room) systems. Examples of non-real-time groupware are office coordination systems, intelligent electronic mail, and software engineering project managers. See [Elli90b] for further examples and references.

* *A preliminary version of this paper, without presentation of model and proof, appeared in the proceedings of the ACM SIGMOD'89 International Conference on Management of Data.*

It is useful to distinguish groupware from other multi-user systems. For example, both database management systems and timesharing operating systems support multiple users. However neither of these are considered groupware since they provide little notification - if one user performs some action, perhaps inserting a tuple or creating a process, other users are not normally notified of the action and may only learn of it by explicitly querying the system.

This paper is concerned with concurrent access within real-time groupware. An example that many find easy to relate to is the multi-player game. Here the movement of one player's token, perhaps a tank or spaceship, triggers updates on the displays of all players. Other examples can be found in the area of computer supported cooperative work [CSCW86, CSCW88]. For instance, in real-time computer conferencing [Sari85], the users, who are often at different locations, communicate through a software medium. This software might allow the users to view and modify a shared graph structure [Ste87] or edit a shared outline [Elli88]. There are significant challenges in implementing groupware that typically do not arise in other applications. Some of these challenges [Elli90b] reside in the areas of group interfaces, access control, social protocols, and coordination of group operations. For instance, groupware introduces new complexities to the user interface: the interface must depict group activity, and designers must weigh the need for continuous display of group context against the potential for distraction. Concurrency control also has novel aspects within groupware as will be demonstrated in this paper.

An invocation of a groupware system is called a *session*. Groupware sessions at MCC are typically an hour or two in length but may be much shorter or much longer. At any point in time, a session consists of a group of users called the *participants*. The session provides each participant with an interface to a shared context, for instance participants may see synchronized views of evolving data. The data at participants' sites is referred to as session objects. In groupware, it is useful to make a distinction between *response time*, and *notification time*. The system's response time is the time necessary for the actions of one participant to be reflected by their own interface; the notification time is the time necessary for one participant's actions to be propagated to the remaining participants' interfaces.

Real-time groupware is characterized by the following:

- highly interactive - response times must be short.
- real-time - notification times must be comparable to response times.
- distributed - in general, one cannot assume that participants are all connected to the same machine or even to the same local area network.
- volatile - participants are free to come and go during a session.
- ad hoc - generally the participants are not following a pre-planned script, it is not possible to tell a priori what information will be accessed.
- focused - during a real-time work session there is a high degree of synergistic shared data access, and many unwanted access conflicts.

- external channel - often participants are connected by an external (to the computer system) channel such as an audio or video link. We used speaker phones within our offices for many of our groupware sessions.

One example of real-time groupware is GROVE (Group Outline Viewing Editor) [Elli88], a fully distributed outline editor which we implemented in the Software Technology Program at MCC. It was specifically designed for real-time use by groups of people simultaneously editing a document during a session. GROVE supports private, shared (subgroup), and public views and windows. Figure 1 shows a typical GROVE shared group window; it appears on the workstations of the three participants shown along the bottom border of the window. Thus a participant can see at a glance who is currently sharing any group window because the iconic bitmap image of participants is always present at the bottom of each window.

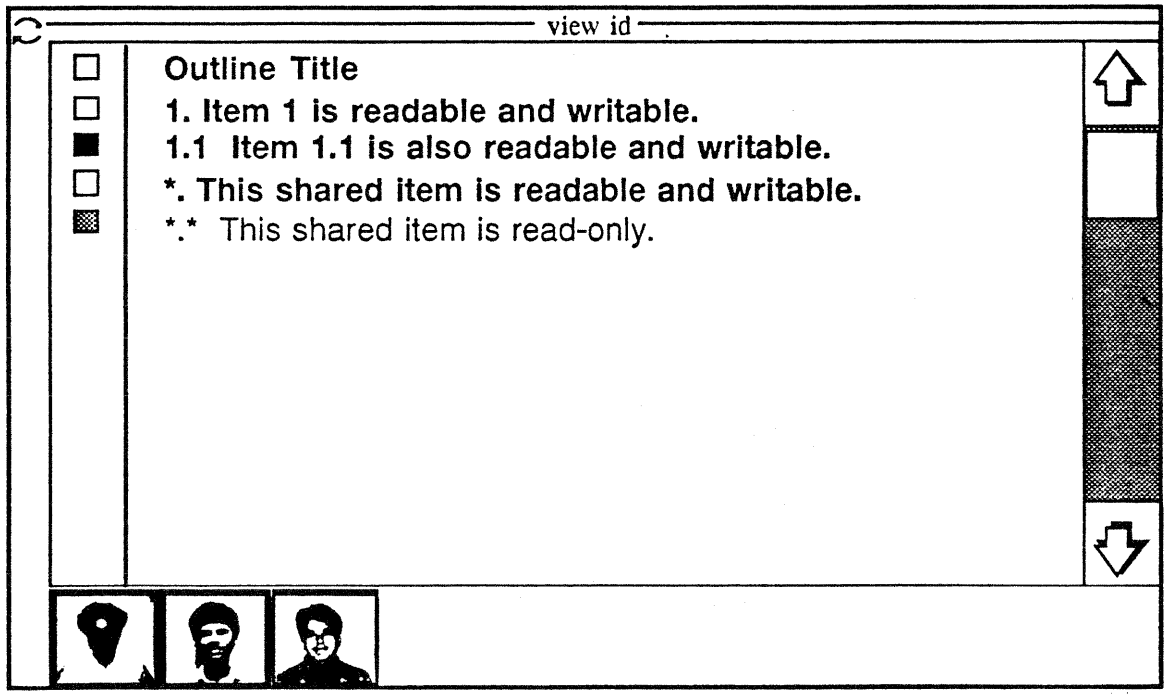


Figure 1. A GROVE group window.

Participants can modify the underlying document by performing standard editing operations (insert, delete, cut, paste, etc.) in any of the windows in which they have appropriate permissions. GROVE provides a fine-grained concurrent editing capability which allows several people to type into the same sentence at the same time and immediately see the effects of each others' edits. Participants can also change the read and write permissions of the various document items. This editor illustrates some key groupware architectural concepts such as group window (e.g. a window which appears on multiple participants screens,) telepointer (a pointer or cursor which appears on multiple participants screens,) and WYSIWIS

(acronym for "What You See Is What I See" and denotes an interface in which the shared environment is guaranteed to appear the same to all participants.)

Other real-time group editors are being explored [Kill90; Knis90]. Very closely related to real-time group editors are loosely coupled (non-real-time) group editors such as CES [Grie86], Quilt [Cohe88], or Shared Books [Lewi88]. These editors allow a group of users to work on the same document, however each user typically works on their own section at their own pace. As a result sessions are less focused and are longer in duration (days or even weeks in length); also real-time notification may not be necessary because of the asynchronous nature of participants' actions.

Considering the above, it becomes clear that group windows and telepointers add extra challenge to the usual notions of concurrency control. In this paper, a very general model (called Team Automata) and a general, non-locking, non-roll-back algorithm for concurrency control within fully distributed real-time groupware will be presented. The next section explores issues related to concurrency control in this context, briefly describes our approach, and describes some alternative approaches. Section 3 describes the Team Automaton model of groupware. Section 4 describes the category of fully distributed architectures to which our algorithm applies, and the correctness criteria. The algorithm is developed in Section 5, and some examples in section 6. Discussion of the correctness proof is presented in Section 7, and conclusions in section 8.

2. Concurrency Control Problem

Concurrency control is needed within real-time groupware to help resolve information access conflicts between participants, and to allow them to perform tightly coupled group activities. For example, with a group editor, clearly there is a conflict if one participant deletes a sentence while a second inserts a word into the sentence. In the usage observations of GROVE, we have noticed that there is a mode of operation in which a tightly coupled group will do a complex sequence of edit operations in a concurrent fashion, getting the task performed in a much more efficient manner. Many CASE tools (computer aided software engineering) discourage rather than enhance closely coupled teamwork. There is a need for mechanisms which go beyond today's typical technology. The various approaches to providing concurrency control, such as explicit locking or transaction processing, that have been developed for database applications do not appear to be suitable in groupware contexts. Interactive concurrency control techniques are much more useful in this context. The following section identifies some of the issues related to concurrency control in groupware, overviews our approach, and then discusses the drawbacks of some current approaches.

2.1 Issues

WYSIWIS. Although there has been little experience in the evaluation of interfaces to groupware [Grud88, Elli89] it appears that some form of a WYSIWIS interface [Ste87] is necessary to maintain group focus. If each user sees a slightly different or out-of-date version then the session's cohesiveness is soon lost. WYSIWIS interfaces have two implications on concurrency control. First response times are important - the time taken to access data, modify data, or notify users of changes must be as short as possible. Secondly, if the concurrency control scheme entails the use of modes where actions of one user are not immediately seen

by the others, then the effect of these modes on the group's dynamics must be carefully analyzed and only allowed if they are not disruptive.

Wide-area distribution. One of the main benefits of groupware is that it allows people to work together, in real-time, even though separated by great physical distances. Consequently these systems may be geographically distributed. With current communications technology, transmission times and rates for wide-area networks are significantly worse than those found in their local area counterparts; the possible impact on response time must be taken into account.

Replication. Because the response time demands of groupware are so high, the data state is usually replicated for each participant. This allows many potentially expensive operations to be done locally. For instance, consider an editing session where one participant is in Los Angeles and the other in New York. Typically each participant would be working in a windowing environment. If the objects being edited and the data state are not replicated then even simple scrolling operations require communication between the two sites. The resulting degradation in response time may be catastrophic.

Robustness. Traditionally robustness refers to recovery from unusual circumstances, typically these are component failures - the crash of a site or a communications link. While these are also concerns within groupware, there is also a second form of robustness these system must achieve, in particular, robustness to user actions. For example, the addition of a new user to the set of users issuing transactions on a database is not normally considered a major problem. However, with groupware, the addition of a participant may result in what amounts to a reconfiguration of the system. Clearly the concurrency control algorithm must adapt to such reconfigurations and in general recover from "unexpected" user actions (abruptly leaving the session, going away for coffee, etc.).

2.2 Our Approach

At MCC, we have explored notions of *soft locks [Elli87],* and *interactive concurrency control [Yeh89].* This paper describes the dOPT algorithm which, when combined with the above techniques, provides a powerful new concurrency control mechanism for groupware. dOPT abbreviates *distributed operation transformation algorithm*, and proceeds without locking or roll-back. This approach relies upon application specific semantic knowledge of the desired outcome of concurrent operations. For example, when two participants make concurrent edits to the same data structure, their local copies are updated immediately, and messages containing the edit operation and carefully selected local state are sent to all other sites. When each of these sites receives the other's message, they know if they are performing the pair of edit operations in different orders. Each first performs an application dependent transformation on the operation, and each applies the transformed operation to their local copy of the data structure. Voila! It is shown that for a significant class of applications, all is guaranteed to end up consistent.

2.3 Other Approaches

Locking. One solution to concurrency control is simply to lock data before it is written. Deadlock avoidance can be handled by the standard techniques (e.g., pre-locking all data to be

used within a transaction) or by methods more suited to interactive environments. For example, with "tickle locks" [Grie86], a request to a locked resource will be granted if the current holder is inactive. Another technique is to provide participants with visual indicators of locked resources [Stef87] and so decrease the likelihood of requests being issued for locked objects. There are three main problems with locking: First there is the overhead in requesting and obtaining the lock, this may include waiting if the data is already locked. In any case, there will be a degradation in response time. Secondly, there is a question of granularity. In the text editing example, it is not clear just what should be locked when the user moves the cursor to the middle of a line and inserts a character. Should the enclosing paragraph or sentence be locked, the file? Or just the word or character? Fine granularity locking is less constraining to the participants but entails greater overhead. The third problem is determining when locks should be requested and released. Using the previous example, should the lock be requested when the cursor is moved or when the key is struck? The system should not burden the user with these questions but it is hard to automatically embed locking into editor commands. For example, if locks are released when the cursor is moved then a user could move to one place to copy some text only to be locked out from pasting it into their original location.

Transactions. Transaction mechanisms have been used for concurrency control in interactive multi-user systems (for example, CES or Quilt), but, these are loosely-coupled systems and have less demanding response time requirements. For real-time groupware there are a number of problems. First there is the complication of distributed concurrency control algorithms based on transaction processing and the subsequent cost to response time. Secondly, if transactions are implemented using locks there are the problems mentioned above, while if some other method is used, such as timestamps, a user's actions may be aborted by the system. (Only aborts explicitly requested by the user should become visible at the user interface.) Generally, transactions are not well suited to interactive use; for instance, a user with two transactions active in separate windows on the same object would be presented with two different data states - it would be better if the windows showed the same state.

There is a basic philosophical difference between databases and groupware. The former strive to give each user the illusion of being the only user of the system, while groupware strives to make the actions of each user visible to others. There has been some work on "opening up" transactions [Banc85], however, the emphasis of this work has been on coordination of nested transactions rather than elimination of the constraints imposed by locking and transactions.

Single Active Participant. Some real-time computer-conferencing systems are intended for situations where only one participant at a time "has the floor" [Lant86]. Access to the floor may be controlled by software or through an external protocol (for example, verbal agreement by the participants). The main problem with this approach is that it is limited to just those situations where having a single active participant fits the dynamics of the session, in particular, it is not suited for sessions with much parallelism among participants. It may overly inhibit the free and natural flow of information among participants. Additionally, if change of floor is left to an external protocol, then participants' errors in following the protocol, or non-cooperation (refusal to follow the protocol), can result in two participants believing they have the floor (and potentially issuing conflicting operations).

Dependency-detection. One proposal for concurrency control in groupware is the dependency-detection model [Stef87]. Dependency detection is based on the use of timestamps to detect conflicting operations; conflicts are resolved by manual intervention. The great advantage of this method is that no synchronization is necessary, non-conflicting operations are performed immediately upon receipt, so response is very good. Mechanisms which involve the user, in general, are appropriate and valuable in groupware applications. However, any method which requires user intervention to assure data integrity is, of course, vulnerable to user error.

Reversible execution. Other schemes have been proposed, and several are under active investigation. Reversible execution is one of these proposals for concurrency control in groupware. With reversible execution [Sari85], operations are executed immediately but information is kept so that they may be undone later if necessary. Many optimistic concurrency control mechanisms fall within this category. A global time ordering in defined for the operations (this may be provided by a central sequencer or ordering on timestamps and site identifiers). When it is detected that two operations have been executed out of order, they are undone and re-executed in the correct order. As with dependency-detection, this method is very responsive. Its disadvantages are the need of a global ordering of operations and the unpleasant possibility that an operation will appear on the user's screen and later disappear because it is undone. There are groupware scenarios in which the forcing of a total ordering produces undesirable behavior, as will be shown later in this paper.

3. The Model

Groupware presents a need to integrate technical, social, and organizational concerns to produce systems which are truly beneficial. The author therefore believes that groupware modeling should encompass organizational goals and procedures, people (and groups) and their social structures, and the tools and systems which can aid in the achievement of these goals [Olson90]. The author also believes that no one model will satisfy all needs and be good for all modeling purposes. In that spirit, this paper describes one of a number of models of interest, and does not represent the spectrum of our groupware concerns. The model is novel and useful in the domain of analysis of groupware architectural structures. It models the communication network, the application domain, and the information structures of groupware, but does not explicitly model the social or organizational aspects.

There are a number of important and outstanding issues that must be faced in the design and implementation of groupware [Elli90b]. Some of these issues, such as distribution, privacy, notification, and group control, fall within the domain of groupware architecture. Our group within MCC has implemented a variety of types of groupware, and strongly believes that a groupware architecture model is needed for (at least) two reasons. First, the implementation of these systems becomes quite complex, and algorithms that have been implemented at MCC and elsewhere to solve problems have been elusively inadequate. A model is much needed to reason about the correctness and performance of these implementations. Secondly, there is a need to understand the underlying commonalities of these implementation constructs. A model such as this can suggest general constructs, elucidate the spectrum of implementation possibilities, and help lay the groundwork for a "groupware implementation language."

3.1 Team Automaton Definition

Within our model, one composes a groupware system (modeled as a team automaton), by creating instances of one or more of the four basic classes of component automata, and hooking them together in a loosely coupled or tightly coupled fashion. This aggregate can then, in turn, be used as a component in a larger team automaton. The four basic classes of automata are:
1) Application Automata (which capture the computations on application objects),
2) Information Base Automata (which capture the information structures or the DBMS),
3) Connection Automata (which capture the network protocols and transmission buffering), and
4) User Interface Automata (which capture the info presentation and user input aspects).

One then specifies the internal operations performed by each of these component automata. The component automata are all defined in the same way. They are an embellishment upon the *Input/Output Automata* conceived by Professor Nancy Lynch at MIT [Lync87]; they are nondeterministic, they interact via "shared actions," they can have an infinite number of states, they can be iteratively nested to form composite automata which satisfy the same definitions, and they can be used to specify simultaneous group operations. The Lynch model is embellished by allowing multiple automata to participate as active inputs to the "shared action" as occurs within groupware sessions.

The automata defined by Lynch represent a rather low level mathematical specification, rather than a high level language specification such as CSP [Hoar78] or Raddle [Form89]. The mechanism is simpler, and at a much different level than the multiway rendezvous [Char87] or the n-party interaction [Fran90]. They are known to be adequate to specify shared variable systems, and message passing systems, although they are neither. They have the power to specify synchronous or asynchronous, blocking or nonblocking systems. Thus their utility is derived from mathematical tractability rather than from human readability or machine executability. Indeed, for some applications, we have encountered aspects of the model which make precise description of the system behavior awkward [Malm89]. Thus, although the author is not completely contented with the model, a large body of literature, theorems, and proof techniques on automata theory is available, and applicable to this model. This has been quite useful.

The automata are rather ordinary, but their interconnection strategy is intriguing because, as we mentioned, it is neither shared variable nor message passing. Lynch classifies the actions which take an automaton from one state to another into three categories:
1) input actions (from another automaton),
2) output actions (to other automata),
3) internal actions (strictly local visibility).

Thus the automaton interconnection strategy is "shared action" in which one or more automata specify within their input action sets, the same action as one or more other automata specify within their output action sets.

A *component automaton* C consists of the following four mathematical entities:
1) a nonempty state set, S(C),

2) a nonempty initial state set, I(C) contained in S(C),
3) an action signature, A(C),
4) a transition relation, F(C) contained in S(C) X A(C) X S(C).

[I/O automata as defined in [Lync87] also include a fifth component, an equivalence relation part(C) which is used for describing fair executions. Since it is not needed in this paper, it is omitted from the current definition. Likewise it is sometimes convenient within the user interface automata to specify an output function which controls the display of information to the users. That is a different topic, and omitted from this paper.]

An action signature A is an ordered triple consisting of three pairwise disjoint sets of actions. We write in(A), out(A), and int(A) for the three components and refer to the actions in the three sets as input actions, output actions, and internal actions of A, respectively. In the system being modeled, the distinctions are that input actions are not under the local system's control and are caused by another non-local component, the output actions are under the system's control and are externally observable by other components, and internal actions are under the local system's control but are not externally observable.

A component automaton is considered to start in some initial state of I(C), and to execute instantaneous transitions into other states of S(C). Note that the state sets need not be finite. This means among other things, that one can model asynchronous message passing systems with unbounded buffer capacity. The transition relation relates a state and an action to another state. If the triple (s_1, a, s_2) is in the relation, then this is interpreted to mean that the automaton, when in state s_1 and presented with the action a, can transition into state s_2. We refer to (s_1, a, s_2) as a *step* of C. An *execution* of C is a finite (also can be extended to infinite) sequence $s_0, a_1, s_1, a_2, ...a_n, s_n$ of alternating states and actions of C such that s_0 is contained in I(C), and (s_i, a_{i+1}, s_{i+1}) is a step of A for every i.

Given a collection, $\{C_i\}$ of component automata, a *team automaton* T consists of the following four mathematical entities:
1) a nonempty state set, $S(T) = \Pi(S_i)$ where Π denotes the cartesian product,
2) a nonempty initial state set, $I(T) = \Pi(I_i)$ where Π denotes the cartesian product,
3) an action signature, $A(T) = \Xi(A_i)$ where Ξ denotes the action signature composition operation described below,
4) a transition relation, F(T) containing all admissible triplets (s, a, s') such that s and s' are members of S(T) and a is a member of A(T).

If there are n component automata being combined to compose a team automaton T, then the state set S(T) is composed of states which are n-tuples or n element state vectors, one element from each of the contributing C_i. On the other hand, the action signature, A(T), is composed of all single actions from any one of the component automata, not vectors of actions. An action is executed by this team automaton by finding all component automata which can execute the action from their given state at the given time, and requiring them all

to simultaneously do it. All component automata which do not recognize that action are dormant during that cycle. A requirement for a set of component automata to be composed is that their internal action sets be disjoint. Then the internal action set of T is the union of the internal action sets of the components. Also the T output action set is the union of the component automata output sets, and the T input action set is the union of all component inputs minus the set of all component outputs.

3.2 Team Automaton Example

To illustrate these concepts and definitions, a simple example is next presented of two component automata, C_1 and C_2 which each have two states and three actions (an input action, an internal action, and an output action) as shown diagrammatically in figure 2. Notice that the composed team automaton, $T = C_1 \Pi C_2$, has two output actions and only one input action. All other actions of C_1 and C_2 are members of the internal action set of T. The formal definitions of C_1, C_2 and T are as follows:

Component Automaton C_1:

$S=\{s_{1\text{-init}}, s_{1\text{-home}}\}$,

$A=\{a_{0\text{-out}}, a_{1\text{-int}}, a_{1\text{-out}}\}$ <note that $a_{0\text{-out}}$ is the only input action for C_1>

$F=\{(s_{1\text{-init}},a_{0\text{-out}},s_{1\text{-home}}), (s_{1\text{-home}},a_{1\text{-int}},s_{1\text{-init}}), (s_{1\text{-home}},a_{1\text{-out}},s_{1\text{-home}})\}$.

Component Automaton C_2:

$S=\{s_{2\text{-init}}, s_{2\text{-home}}\}$,

$A=\{a_{0\text{-out}}, a_{1\text{-out}}, a_{2\text{-int}}, a_{2\text{-out}}\}$ <note that $a_{0\text{-out}}$ and $a_{1\text{-out}}$ are input actions for C_2>

$F=\{(s_{2\text{-init}},a_{0\text{-out}},s_{2\text{-home}}), (s_{2\text{-init}},a_{1\text{-out}},s_{2\text{-home}}),$
$(s_{2\text{-home}},a_{2\text{-out}},s_{2\text{-home}}), (s_{2\text{-home}},a_{2\text{-int}},s_{2\text{-init}})\}$.

Team Automaton T:

$S=\{$ $[s_{1\text{-init}},s_{2\text{-init}}]$ <this is the initial state for T>,
$[s_{1\text{-init}},s_{2\text{-home}}]$
$[s_{1\text{-home}},s_{2\text{-init}}]$
$[s_{1\text{-home}},s_{2\text{-home}}]\}$,

$A=\{$ $a_{0\text{-out}}$, <this is input action of T>
$a_{1\text{-int}}, a_{2\text{-int}}$, <internal actions of T>
$a_{1\text{-out}}, a_{2\text{-out}}\}$ <output actions of T> $\}$

$F=\{$ $[(s_{1\text{-init}},a_{0\text{-out}},s_{1\text{-home}}), (s_{2\text{-init}},a_{0\text{-out}},s_{2\text{-home}})]$,
$[(s,a_{1\text{-out}},s_{1\text{-home}}), (s_{2\text{-init}},a_{1\text{-out}},s_{2\text{-home}})]$,
$[(s_{1\text{-home}},a_{1\text{-int}},s_{1\text{-init}}), ()]$,
$[(), (s_{2\text{-home}},a_{2\text{-int}},s_{2\text{-init}})]$,

$[(), (s_{2\text{-home}}, a_{2\text{-out}}, s_{2\text{-home}})]\}$.

Braces {} denote sets; parentheses () denotes relations, brackets [] denote vectors, and angular brackets <> denote comments.

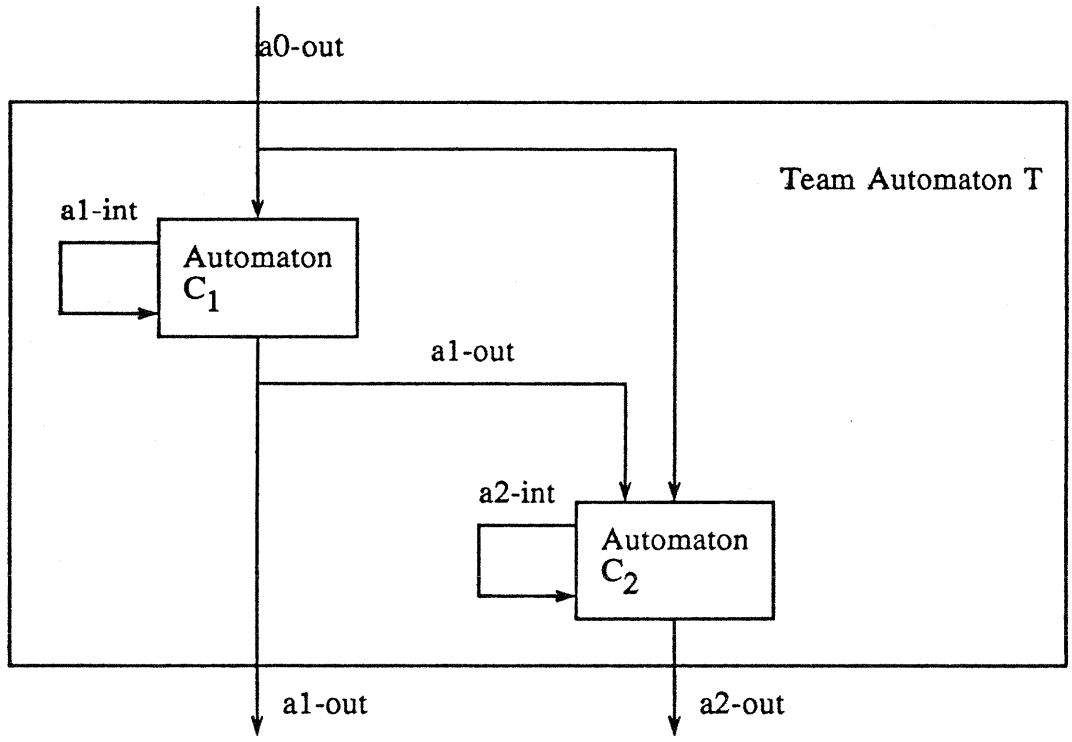


Figure 2. Example Team Automaton

Component automata deviate from Lynch's definition of I/O automata by not requiring that there must exist a transition for every combination of state and input action. Thus our automata are not forced to be input enabled, and an algorithm can choose to ignore (not be interrupted by) certain input actions while in certain states. Team automata deviate from Lynch's definition of composition of automata by not requiring that the output actions of the component automata be disjoint. This allows the modeling of *group operations* that mirror the ways that tightly coupled, well coordinated groups work.

4. The Architecture

A groupware system consists of a set of participants and other components. Frequently, participants have sub-systems to help them communicate and attain their goals. Groupware architectures can be classified according to the connectedness of the sub-systems, and according to the level of centralization or decentralization. There is a spectrum of design pos-

sibilities from fully centralized to fully distributed implementation for each of the four components of a system corresponding to the four classes of component automata which were introduced in the previous section. Data can be centralized, distributed, or hybrid (modeled by the information base automata.) Control can be centralized, distributed, or hybrid (modeled by the connection automata.). But in addition, groupware needs to pay careful attention to the group user interface. Thus, the computation of the user views as seen on participant screens (or other devices) can be done once centrally, or done separately at each site (modeled by the user interface automata.) The same argument for a spectrum of possibilities holds for the application computations (modeled by the application automata.)

It should be clear that the model presented is capable of expressing a wide range of groupware architectures. See [Elli90a] for further examples. However, in the remainder of this paper, the focus will be on using the model to describe and analyze the architecture selected for the distributed GROVE system.

4.1 Fully Distributed Systems

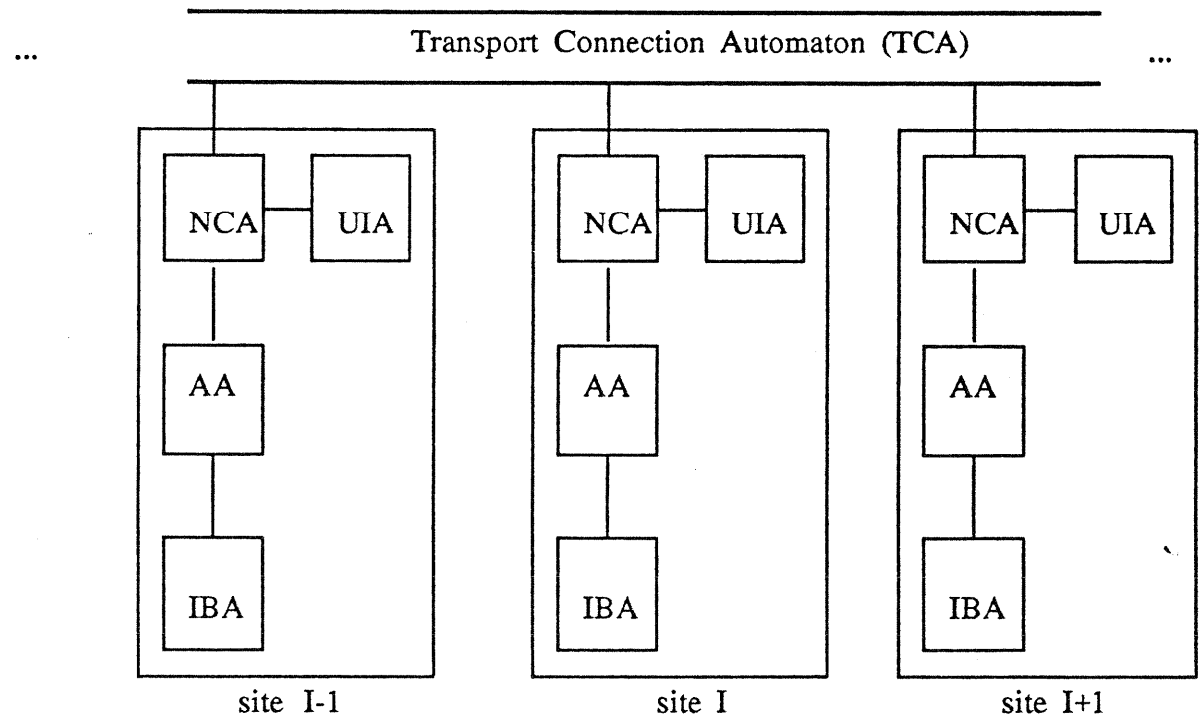
A particularly interesting category within the above classification is the fully distributed systems (distributed control, distributed data, distributed application, and distributed user interface.) These systems have very high performance potential, exhibit tantalizing distributed computing challenges to implementors, and can be sculpted to elegantly fit the real-time groupware requirements previously listed. The further discussion in this paper assumes a distributed workstation environments where the local computing facilities associated with each participant, called the participant's *site*, are capable of storing a significant subset of the operational data, performing operations on this data locally, and displaying the data via different user interfaces and different views. Given the decreasing price and size of computation power (processors,) this is a viable option, and may be a very popular option of the future. It is one of the options available within our lab, and provides the platform for GROVE (which is built upon a platform of Sun workstations, ethernet, NeWS window manager, and UNIX based tools.)

Usually, there is one site per user, and a site corresponds to one or more machines (the user's workstation and assessories); however some machines may run multiple sites. It is assumed that any site can communicate with any other site; they are typically connected via high speed networks. Within the category of fully distributed systems, there are still many possible choices of architecture. One must decide, for example, which modules are directly connected to which other modules. E.g. should the information base automata be directly interconnected?

Figure 3 shows a block diagram of the team automaton model of the distributed GROVE architecture. An arbitrary, finite number of sites are allowed; notice that the sites are shown connected by a *transport connection automaton* in figure 3. The transport connection automaton (TCA) is a subclass of the connection automaton which is frequently needed to model buffering in asynchronous systems and delay in communication lines. Thus, this automaton simply acts as a broadcast communications buffer by accepting input events from any site, and delivering them to all other sites after some finite delay. Events may be delivered with a computed or random delay, they may arrive out of order, and arrive at different times to differ-

ent sites. Our model assumes that the number of sites is fixed, and that no failures occur; these simplifying assumptions can be removed, as shown later.

Figure 3. Groupware Fully Distributed Architecture



legend:

NCA means Network Connection Automaton
AA means Application Automaton
UIA means User Interface Automaton
IBA means Information Base Automaton
TCA means Transport Connection Automaton

4.2 Internal Architecture

Internally a site consists of four components as shown in figure 3. The network connection automaton (NCA) is a site manager, the application automaton (AA) executes the application operations, the information base automaton (IBA) handles data manipulation, and the user interface automaton (UIA) handles all input from and presentation to the local participant. Each of these automaton types is next described in slightly more detail.

The network connection automaton (NCA) is a subclass of the connection automaton which implements the distributed control algorithm. It receives all relevant events from other sites via the TCA and delivers them to the local UIA and AA, Also, it is the sender of local events from the local participant via the UIA to other sites via the TCA. It maintains eventcounts

and history lists so that it can send and receive local state information to/from other NCAs. It is independent of the details of the application. The GROVE distributed control algorithm, which executes within a set of NCAs, will be a primary object of study in this paper. That algorithm will be described and verified.

The application automaton (AA) contains and executes all application specific data and algorithms. It receives requests to perform application specific operations from its local NCA. Local requests are passed from the local UIA to the local NCA to the AA. Other sites non-local requests are passed from another site to the TCA to the local NCA, and then to the AA. Modularity which separates the AA in this manner has the advantage that all application operations, local and non-local, have a single locus of execution at any site. Applications can be either *collaboration aware* as in GROVE, or *collaboration unaware* in which case the application does not know that it is servicing more than one client. Collaboration aware applications can use their knowledge of the participants and of the application domain to assist the UIA to present tailored and enhanced information to the participants, to assist the NCA, and to do more sophisticated application computation. Collaboration unaware applications represent the vast majority of currently existing applications. This option implies that there is no need to laboriously alter or recreate an existing application in order to allow it to be shared in real time by a group.

The information base automaton (IBA) is concerned with the storage structures and algorithms of the information objects being manipulated. This sub-system may range from an elaborate object oriented database to a simple string manipulation program. The information may be standard database data (e.g. employee records,) or a set of documents, or multimedia nodes of a hypermedia system, or CAD design objects. Since these structures are typically application dependent, this module is frequently directly connected to the AA (see figure 3.) Architectural issues arise in this domain as to whether the set of IBAs should transparently handle all replication of data, and be interconnected as a separate subsystem, or whether replication should be a system visible property. In the GROVE design, we have chosen the latter approach, partly because distributed object oriented databases fulfilling our requirements were not available at the time, and secondly because system visibility allows us to experiment with more options and alternatives.

The user interface automaton (UIA) implements presentation and user input functions. It receives input (e.g. keystrokes) from its local participant which it must pass to the AA. It receives data from the AA which it must present in an appropriate manner to the local participant. Frequently the UIA cannot simply spit out the next output, but needs to input the complete set of objects which form the relevant context, perform a transformation, and then regenerate a new output object set. If the participant manipulates the application by sophisticated input devices in interesting non-procedural ways, this must be mapped to standard high level input and output events that are expected and can be processed by AAs. If the participant wants to view the items of data which are output by the application as graphs or as animations, then these transformations must be handled by the UIA. The UIA describes and monitors the participant's IO environment, and is directly connected to the local NCA so that both local and non-local communication can occur without the AA necessarily being collaboration aware.

In this analysis, the UIA is not explicitly employed. Correctness is viewed in terms of the consistency of the replicated information objects, independent of the manner in which these objects, or a subset thereof, are presented to the participant. Likewise, the IBA is not explicitly employed; instead the discussion is simplified by assuming that the AA can directly access the information objects via a predefined set of actions.

4.3 Model Definitions

A *session* is defined to be a finite execution of a team automaton. Associated with a session is a *session object set* - the set of information items manipulated by the application. In the fully distributed case, this session object set is replicated at each site. Thus there is an instance of all of the working data, called a *site object set*, resident at each site, and controlled and manipulated by the local information base automaton (or the AA in simplified cases.) Note that session object sets are an abstraction, and there is no "master copy" stored anywhere.

The concurrency control team automaton presented in the next section is the specification of an NCA and an AA. Informally, it works by testing each incoming event into the NCA to see if it possibly overlaps with other events. If not, the event is passed to the AA to be executed. If so, the event is queued, passed to the AA at the correct time to be transformed, and then executed. A team automaton is defined to be *correct* if it guarantees that an initially consistent system will, at the conclusion of any admissible execution, still maintain consistency, semantic integrity, and temporal ordering.

> **Definition.** The *consistency property* states that all site object sets (replications of the session object set) must be identical at the conclusion of any admissible team automaton execution. Note that the automaton must begin in a valid start state. This is a state in which the data sets (site object sets) at each site are identical. The component NCA's and AA's must *all* execute the dOPT algorithm specified in the next section.
>
> **Definition.** The *temporal ordering property* states that if one event is completed at site s before a second event is initiated at site s, then the effect of the first event should precede the effect of the second at all sites at quiescence.
>
> **Definition.** The *semantic integrity property* states that any site object set, at quiescence, must conform to the operation semantics and to the concurrency semantics specified for the application.

Examples employing these definitions will be presented later. The first two properties are enforced by the concurrency control algorithm which executes on the NCAs. The third property, which insures that consistency is not maintained via semantically meaningless transformations, is application dependent, and must be enforced by the AA when it performs the transformations. The algorithm presented next applies to applications executing within a fully distributed architecture as depicted by figure 3. It handles non-serializable event sets, and requires no locking.

5. The Algorithm

We will now look at a distributed non-locking algorithm for solving the concurrency control problem in fully distributed groupware. This algorithm has two properties which make it suitable for such systems. First, operations are performed immediately on their originating site, thus responsiveness is good. Secondly, the algorithm is completely distributed; this can also help achieve responsiveness and robustness.

We make the following assumptions:

1) the number of sites is constant,

2) sites do not fail,

3) messages are received without error

4) messages are not lost.

Note that it is not assumed that messages arrive in the order in which they were sent. (This has a practical advantage of simplifying implementations using datagram protocols. A problem with virtual circuits is that each site must maintain a link to all other sites. With datagram protocols this is not necessary, however, messages may arrive out of order.)

It is possible to extend the algorithm to account for a varying number of sites over time and site failure. These extensions will be discussed at the end of this section. The third and fourth assumptions should be satisfied by the communications protocols so we take them as given.

5.1 Data Structures

State Vectors. Let N be the number of sites in the system (we are assuming that N is constant). Each site has a unique identifier (for example, its network address). For simplicity, assume that the sites are identified by the integers 1.....N. The state vector of site j, is a N component vector where the i'th component indicates how many operations from site i have been processed by j. Given two state vectors, s_i and s_j, we say that:

1) $s_i = s_j$ if each component of s_i is equal to the corresponding component of s_j,

2) $s_i < s_j$ if each component of s_i is less than or equal to the corresponding component in s_j and at least one component of s_i is less than the corresponding component in s_j,

3) $s_i > s_j$ if at least one component of s_i is greater than the corresponding component in s_j.

Requests. Requests are tuples of the form <*i*, *s*, *o*, *p*> where *i* is the originating site's unique identifier, *s* the originating site's state vector, *o* an operation, and *p* is the priority associated with *o*.

Request list. Requests waiting to be processed by a site manager are kept in the site's request queue. A request, <*j*, *s*, *o*, *p*>, in Q_i, the request queue for site *i*, indicates that *o* has been requested by site *j* while in state *s*. An entry is added to the request queue when: 1) the site manager receives a request from the network, or 2) the site manager receives a request from the user interface. Entries are removed from the request queue when the site manager determines that the requested operation may be executed. (Note, although the term "queue" is used, entries need not be removed in first-in-first-out order.)

Request Log. Each site manager maintains a log of requests performed on the site. A request, <*j*, *s*, *o*, *p*>, in L_i, indicates that site *i* while in state *s*, performed operation *o* (requested by site *j*). The log is ordered by insertion, so it is possible to find the first entry, the most recent entry, and to step through in insertion order.

Transformation Matrix. The transformation matrix is the key to resolving conflicting operations and is how semantic knowledge of the application is introduced into the algorithm. The transformation matrix, **T**, is an *m* x *m* matrix, where *m* is the number of actions defined over the session objects. Each component of **T** is a function which transforms events in conflicting situations into other events. These functions are executed at the appropriate times by the AA.

5.2 The Distributed Operational Transformation (dOPT) Algorithm

A specification of the distributed operational transformation (abbreviated as dOPT) algorithm is presented next using a generic algorithmic specification language augmented by *send, receive, and broadcast* operations. The following is the algorithm as executed by the network connection automaton (NCA) at the i'th site. All other NCA's execute an identical algorithm.

Initialization:

$Q_i \leftarrow \emptyset$

$L_i \leftarrow \emptyset$

$s_i \leftarrow$ <0,0,...,>

Main Loop:

do while TRUE

receive $<j, s_j, o_j, p_j>$

$Q_i \leftarrow Q_i + <j, s_j, o_j, p_j>$

for each $<j, s_j, o_j, p_j> \in Q_i$ where $s_j \leq s_i$ begin

$Q_i \leftarrow Q_i - <j, s_j, o_j, p_j>$

if $i = j$

broadcast $<i, s_i, o_j, p_j>$ *to all other sites*

fi

if $s_j < s_i$

$<k, s_k, o_k, p_k> \leftarrow$ most recent entry in L_i where $s_k \leq s_j$

do while $<k, s_k, o_k, p_k> \neq \emptyset$ and $o_j \neq \emptyset$

if the k'th component of s_j is ≤ the k'th component of s_k

let u be the index of o_j (ie, $o_j \in M_u$)

let v be the index of o_k (ie, $o_k \in M_v$)

$o_j \leftarrow \mathbf{T}_{uv}(o_j, o_k, p_j, p_k)$ *{by application automaton}*

fi

$<k, s_k, o_k, p_k> \leftarrow$ next entry in L_i (or ø if none)

od

fi

if $o_j \neq \emptyset$

perform operation o_j on i's site object set *{by application automaton}*

$L_i \leftarrow L_i + <j, s_i, o_j, p_j>$

fi

$s_i \leftarrow s_i$ with j'th component incremented by 1

end

od

The initialization section simply sets the site's log and request queue to be empty and initializes the site's state vector. The algorithm then enters its main loop. The first statement receives a request (which would come from the network (TCA) or the user interface (UIA)); the request is then added to the queue. Next the queue is examined for requests which may be executed. Let s_j be the state vector of an entry in the request queue, there are three possibilities:

Case I: $s_j > s_i$

The operation cannot be executed at this time (site *j* has performed operations which have not yet been performed by site *i)* and so is left on the queue.

Case II: $s_j = s_i$

When the two states are equal the operation is performed immediately (and, if $i = j$, the entry is also broadcast to the other sites).

Case III: $s_j < s_i$

In this case the operation can also be performed. However, since the execution order of operations on site *i* will differ from that on site *j*, the operation must first be transformed by examining the log entries site *i* has accumulated and which are more recent than s_j.

The second and third case both lead to the execution of an event by the application automaton. When this occurs, an operation is applied to the site object set by the AA, added to the log by the NCA, and the state vector is incremented.

5.3 Transform Algorithms

Since the transform algorithms are application dependent, the following discussion will focus upon the text editing application as exemplified by the GROVE editor with a text string (set of characters) as its site object set. Consider a group editor with two types of operations (actions) defined as follows:

A_1 = insert[X; P]	insert character X at a position P in the text string,
A_2 = delete[P]	delete the character at position P in the string.

The events which can be applied to the site objects are instantiations of the actions. In the previous case, this would be operation instances such as o = A_1[a, 3]. Each event changes the state of the site object, for instance o("xyz") is "xyaz".

If a string is represented as a set of (value, position number) pairs, S = {(S_i, u_i), i=1...k} then we can define the *standard textedit semantics* for insert and delete as follows:

insert(x, u) applied to a string S yields string S' whose elements are (S_i', u_i') such that:

$S_i' = S_i$, and $u_i' = u_i$ if $u_i < u$;

$S_i' = S_i$, and $u_i' = u_i+1$ if $u_i \geq u$;

and add (x, u) to the set S'.

delete(u) applied to a string S yields string S' whose elements are (S_i', u_i') such that:

omit the pair (x,u);

$S_i' = S_i$, and $u_i' = u_i$ if $u_i < u$;

$S_i' = S_i$, and $u_i' = u_i-1$ if $u_i \geq u$

Other GROVE edit operations such as *cut, paste, copy,* and *move* have similar semantics, and can be derived from insert and delete. The transformation matrix **T** which is executed by the application automaton, has four entries in the case of two actions (insert and delete). We call this the GROVE transform matrix. This transform is applied in case III of the above algorithm. This is the case when *i* is not equal to *j*. If we have two operations o_i = insert[X_i ; P_i] and o_j = insert [X_j ; P_j] originating from sites *i* and j, then $\mathbf{T}_{11}$ is defined as:

$\mathbf{T}_{11}(o_i, o_j, p_i, p_j) = o_i'$ where

if($P_i < P_j$)

o_i' = insert[X_i; P_i]

else if($P_i > P_j$)

o_i' = insert[X_i; $P_i + 1$]

else /* $P_i = P_j$ * /

if($X_i = X_j$)

$o_i' = \emptyset$

else

if($p_i > p_j$)

o_i' = insert[X_i; $P_i + 1$]

else /* $p_i \leq p_j$ */

o_i' = insert [X_i; P_i]

fi

fi

fi

There are two interesting cases in the definition of $\mathbf{T}_{11}$. First, when the arguments of both operations are equal, o_i' is set to null. The reason is that since $i \neq j$ (**T** is never applied to pairs of operations from the same site), it must be that two different sites have requested the same operation before seeing each others' request, hence it is possible to ignore one request. The second interesting case is when $P_i = P_j$ (i.e., the insertion positions are equal), such conflicts are resolved by using the priority order associated with each request. Priority is used for tie-breaking when similar operations are initiated concurrently. All nodes must order these in the same way. One would think that a unique site identifier would be sufficient, but we will see in the next section that it is not. The priority of an operation used in **T** is a composite of its predecessor's priority and its originating site identifier.

The remaining entries of **T** are simpler; they are listed here for completeness:

$\mathbf{T}_{22}$(delete[P_i], delete[P_j], p_i, p_j) = o_i' where

if ($P_i < P_j$)

o_i' = delete[P_i]

else if ($P_i > P_j$)

o_i'= delete [P_i -1]

else

o_i' = ø

fi

$\mathbf{T}_{12}$(insert[X_i; P_i], delete[P_j], p_i, p_j) = o_i' where

if($P_i < P_j$)

o_i' = insert[X_i; P_i]

else

o_i' = insert[X_i; P_i -1]

fi

T_{21}(delete[P_i], insert[X_j, P_j], p_i, p_j) = o_i' where

if($P_i < P_j$)

o_i' = delete[P_i]

else

o_i' = delete[P_i +1]

fi

5.4 Quiescence

As mentioned at the beginning of this section we have made a number of assumptions (fixed number of sites, no failures) that would in practice limit the usefulness of this algorithm. Furthermore, as it stands, the algorithm has a continuously growing data structure (the log) which must be scanned when calculating a request's priority and when transforming a request on an earlier state. These problems can be solved by regularly quiescing the system. Quiescence should be enforced periodically (e.g., once every few minutes) and when the system falls quiet for an extended period of time (e.g., ten seconds). Detection of quiescence is equivalent to the well known distributed consensus problem; algorithms for this problem may also detect site failures (dependent upon the characteristics of the underlying processing and message systems). When the system is quiescent the log can be reset (set to null) and participants can enter and leave the session. Furthermore the session object set can be checkpointed by each site.

6. Examples

Our algorithm maintains consistency without locking. In this section, motivation is given for the algorithm and its proof via several examples. For enhanced understanding, and decreased size of this paper, we provide informal proofs at a high level rather than rigorous proof presentation in the team automaton notation.

To summarize our assumptions, a set of participants are simultaneously editing a session object set which is implemented as a replicated set of objects (called the site object set) at each workstations of each of the participants (called a site). The site object sets are initially all identical, but this does not mean that the views of the objects seen by the participants are necessarily identical, since different hardware and software may reside at the different sites, and different participants may most appropriately see different views. Each site can perform operations only on their local site object set; also sites can send, broadcast, and receive messages. An event can be initiated by any participant at any time at their local site, and should eventually be applied at all sites. This operation is immediately performed locally, and simultaneously sent to all other participants. Some scenarios can aid our discussion.

6.1 Two Party Scenarios

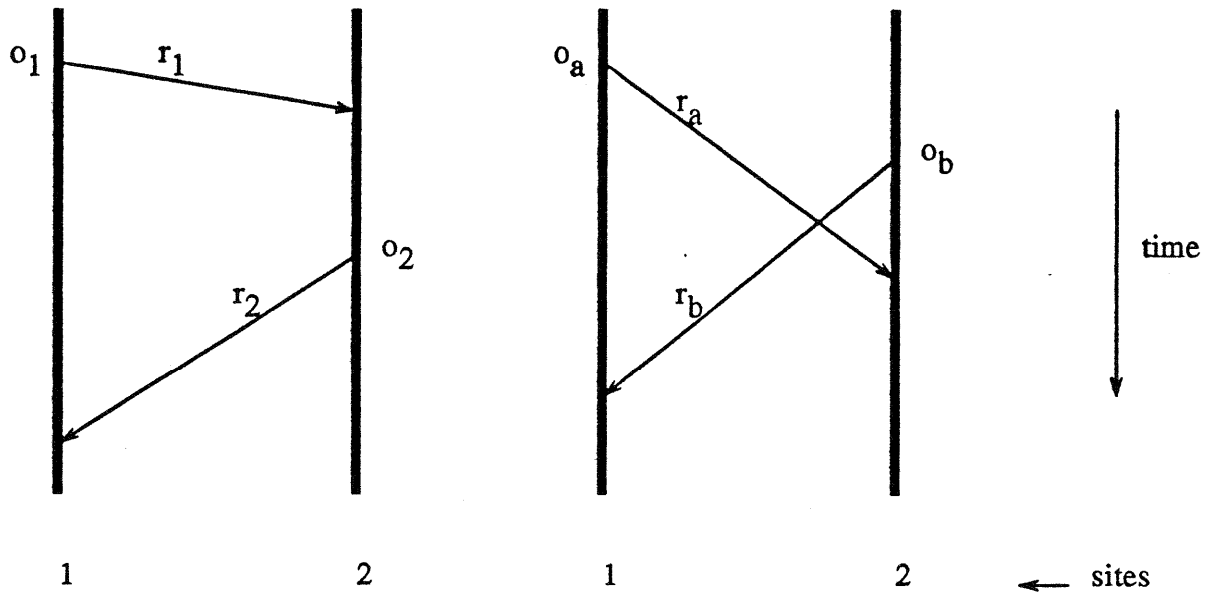


Figure 4. Non-overlapping (a) and Concurrent (b) Event Sets

A session can be depicted graphically by a time line diagram. Figure 4a shows a time line for two sites as two thick vertical lines with time progressing downward. The point o_1 on the first vertical line denotes a point in time at which the event o_1 is executed at site 1, and a message is sent to site 2. The thin directed line labelled r_1 denotes the request message from site 1 to site 2, requesting that o_1 be executed at site 2. Likewise, o_2 denotes an event at site two that is passed to site 1 via request r_2. This is an example of two events, o_1 and o_2, which are not concurrent because o_1 arrives at site 2, and is executed there, before o_2 is initiated. Thus there is no potential for conflict. We say that o_2 has temporal precedence over o_1 written $o_2 >_t o_1$.

Figure 4b shows a session where there are concurrent events o_a and o_b. Messages are sent by both sites at approximately the same time, before the first of them is received. Each site executes its event without knowledge of the other's, so the messages cross in transmission, shown by crossing lines labelled r_a and r_b in the figure. In this example, this results in the operations o_a and o_b being executed in opposite orders at the two sites.

Suppose o_a = insert[a; 2] and o_b = insert[b; 3]. Then o_a("xyz") yields xayz, and o_a followed by o_b yields xabyz. At the second site, o_b is first executed which yields xybz, and o_b followed by o_a yields xaybz. The two site object sets are unequal, so this is an inconsistent execution.

In this example, the participant who inserted the b did not see the insertion of a, and thus intended the b to be placed after the y within the string xyz. This suggests a transformation which adds one to the insertion point if an insert event occurs concurrently with another insert at an earlier position in the string. Note that concurrent events can be detected by comparing a site's state vector to the state vector attached to the incoming message. Applying this transform to the current example, the final string at both sites would be xaybz. We state the generalization of this rule as follows.

Concurrent Edit Semantics: If an operation o_i at string position P_i is concurrent with j inserts at earlier positions, and k deletes at earlier positions, then this should be transformed to o_i' = [the event o_i applied at position P+j-k], where j-k > 0.

Since there is no locking, it is possible that several participants invoke actions to operate on the same position in the string at the same time. Consider the session of figure 4b in which the operations being executed are both delete operations with the same parameter. Clearly, both participants were intending to delete the same character, and clearly, neither participant saw the other's delete operation before performing their own. Thus, the intent of these actions was to get rid of a single character. If no transformation was performed, then the end result would be the erroneous deletion of two characters. In GROVE, this crossing is detected and only one character is deleted (see T_{22} above). Redundant deletes must be marked as such within the log, and the concurrent edit semantics modified appropriately. This has been incorporated within the delete-delete component of the transform matrix T. This is an example of a DWWM facility (Do What We Mean). It appears to be quite useful to implement DWWM facilities within groupware.

Notice that this is one of many examples of non-serializable behavior. Much of the literature concerning database consistency and replicated systems defines correctness in terms of serializability [Bern87]. There is no serial ordering of the pair of delete operations which produces the desired result. Thus groupware implementation considerations suggest new frontiers where non-serialized correctness criteria are needed.

Another interesting case is the one in which the scenario of figure 4b is carried out with two insert operations at the same insertion point. Suppose o_a = insert[a; 2] and o_b = insert[b; 2]. These two events are inserting different characters at the same position. At site 1, o_a("xyz") yields xayz, and o_a followed by o_b yields xbayz. At the second site, o_b is first executed which yields xbyz, and o_b followed by o_a yields xabyz. The two site object sets are unequal, so this is an inconsistent execution. One attempt to correct this problem is to again add one to the event if a concurrent event at the same position is detected by comparing

state vectors. Thus, in the above example, when the second operation, o_b is executed at site 1, it is inserted at position 3 rather than position 2 which yields xabyz rather than xbayz. At site 2 the second operation is transformed to insert[a; 3] so that the final string is xbayz. Opps; the two strings are still unequal and inconsistency again occurs. If both events are transformed, then the same problem arises. We must construct a method to choose only one of the two events to transform.

A solution to this inconsistency might assign priorities to each event, and when two events collide as above, then the one with a lower priority would be incremented, but the higher priority one would not. Considering this as a race condition, the priority comparison is a tie breaking mechanism. Since sites have unique integers as site identifiers, the site identifier can be used as the priority since this conflict only arises with events from different sites. Note that priority comparison is not needed in the scenario of figure 4a even if both events are insertions at position 2 because of temporal precedence. The explicit intent at site 2 in this case, was to place the second insertion (b) in front of the a, and the result of the insertions would naturally be xbayz at both sites. Since there are no concurrent events, there is no need to apply a transformation in this case.

The transformation rule which we have discussed says that whenever concurrent insertions at the same position are detected, we compare priorities, and if the operation to be performed has a lower priority, then we increment its insertion position by 1. Using this rule for the specified two insertions within figure 4b, when the insert[b; 2] event arrives at site 1, it would not be altered by the transform T and would result in the insertion of b at the second position in the string in front of the a character yielding xbayz. When the insert[a; 2] event arrives at site 2, its priority of 1 would be compared to the priority of the receiving site which is 2. Since its priority is lower, we transform the event by incrementing its insertion position yielding insert[a; 3]. The application of this transformed event places the 2 in the third position after the b, also yielding the string xbayz. Using this transformation rule, the final string at each site is the same. Thus consistency is maintained in this case by the transformation. If there were multiple crossings then an event's insertion position might be incremented numerous times. It seems that this solves our consistency problem; however, intuitions can be deceptive as previously mentioned.

6.2 Three Party Scenarios

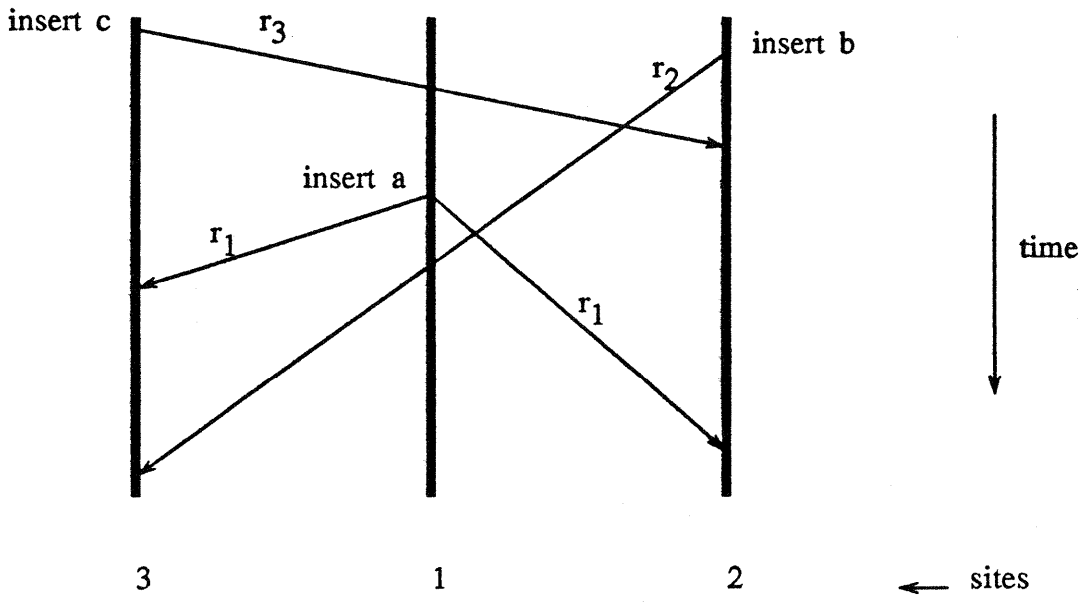


Figure 5. Mixed Priority Example

Lets examine the workings of our algorithm for the session shown in figure 5 which has three sites and two crossings of messages. Assume that site 1 initiates operation insert[a; 1]; site 2 initiates insert[b; 1]; and site 3 insert[c; 1]. At site 3, the c is first prepended to the initial string, then the insert a operation arrives. Although its priority is lower, it is prepended in front of the c (due to temporal precedence). Finally the insert b operation arrives and its priority is compared to both of the previous two operations. Since its priority (2) is lower than that of site 3, it is changed by the transform T to insert[b; 2] and is inserted after the a and before the c. The final string begins with abc.

At site 2, it first prefixes b to its initial string, and then prefixes c from site 3. When the insert[a; 1] arrives from site 1, its priority is compared to that of the previously executed insert b operation, and this causes the incoming operation to be changed to insert[a; 2]. Note that due to temporal precedence, the operation's priority is not compared to the insert c from site 3. The final string in the view buffer at this node begins with cab which is inconsistent with the abc at node 3.

We can gain some perspective on why this solution fails by further examining our notion of correctness. Recall that the temporal precedence property implies that if operation o_2 is initi-

ated after its site has executed some other operation o_1, then the final site object set must reflect o_2's knowledge of o_1. This means that the transformation T must not under any conditions change o_2 to appear to have occurred before o_1. Thus, in figure 5, the a generated by the operation instigated by site 1 must occur to the left of the c instigated by site 3 because the participant at site 1 saw the insertion of the c, and then explicitly requested an operation which would place an a in the position in front of (to the left of) the c. Transformations and other insert operations could cause other characters to be between the a and the c, but they should not cause the a to occur after (to the right of) the c in the final string.

Temporal precedence suggests a problem with one of the transforms in the session depicted by figure 5. At site 2, the position parameter of the insert a operation is incremented before it is executed. The intent is to move the a past the b which has higher precedence, but the actual effect is to move the a past the c which violates the temporal precedence constraint. At this point we might try changing the rule so that if the priority of X is less than the priority of Y, then we put X to the left of Y (rather than to the right as we proposed above.) Although this patch works in this case, it fails in other rather similar cases; the problem is deeper than this and requires more rethinking as we illustrate next.

We can obtain assistance in our quest for a solution by using theorems in the literature of automata theory. One useful theorem specifies that there is an equivalence between any composite automaton such as our team automaton and a nondeterministic single stream automaton [Lync87]. Different execution sequences of this single stream automaton implement the behavior of the various sites, and any other permutations of inputs that are possible. Consider the session depicted by figure 5; figure 6a is a temporal precedence graph which shows the temporal ordering t of the three events which will drive the equivalent single stream automaton. The three nodes represent the events; the thick solid line denotes a temporal precedence relation - it says that, in our example, the *insert a* event comes after (and therefore knows about) the *insert c* event. Notice that there are no arcs touching the *insert b* event. This means that it has no knowledge of others when it is initiated, and others know nothing about it at their initiation, so *insert b* is executing concurrently.

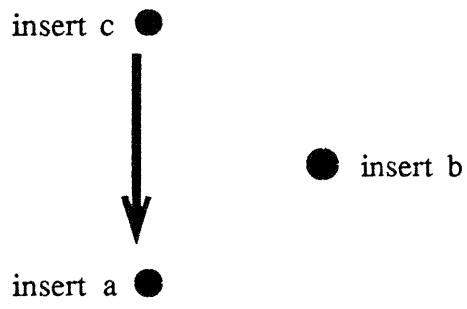


Figure 6a. Temporal Precedence Graph

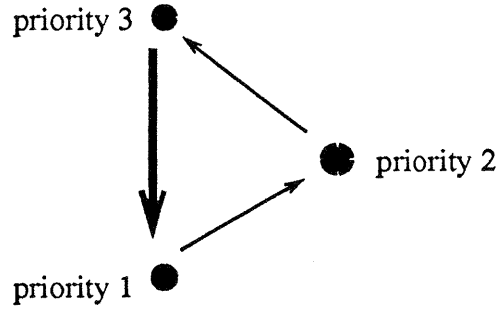


Figure 6b. Combined Precedence Graph

Figure 6b is a combined precedence graph which takes into account the temporal relations as well as priorities between the events. This figure explicitly shows via thin lines that c should appear in the final string before b, and b should appear before a (due to priorities). Notice that we do not draw a thin line between any two nodes connected by a thick line because if there is temporal precedence, this over-rides the priority associated with an event. The thick line shows that a should appear before c (due to temporal precedence). These three constraints are mutually unsatisfiable. One cannot have c before b and b before a and a before c. This dilemma arises because node 2 has a precedence that separates a predecessor priority (3) from its successor's priority (1).

Our suggested solution, which we call the *list priority* scheme, is to define the priority of an event dynamically as a concatenation of its predecessor's priority and its own site identifier. Thus, an unrelated event's priority can never be less than some event, but more than its predecessor.

Definition: The list priority scheme defines the priority p of an event to be a list constructed according to the following priority calculation.

Priority calculation:
The *list priority* of an event o is a list of the site identifiers of the relevant predecessors of o in order, along its maximal chain. To compute this we use the site identifier of o as the final element of the list. We find the maximal immediate predecessor of o, and use its site identifier as the list element preceding the final element. Maximal means largest list priority, where the list priority is specified below. We continue building up the list from last element to first. The first element means an operation in this session which has no predecessor. Note that this priority calculation only needs to be applied to events with the same position parameter. The site identifier of this element is the first item on the list which composes the priority of o. This list can also be easily computed on the fly if messages all contain the priority of their senders. An event simply appends its site identifier to the priority of its maximal immediate predecessor.

Priority value comparison:
Two list priorities are compared element by element, from beginning of the list to the first element in which they differ. Whichever is larger at the element where the lists differ has the higher priority value. If one list is a sublist of the other, then the longer list has a higher priority value.

In the example depicted by figure 5, the event *insert a* would have a priority of (3,1) which is a list containing its site identifier preceded by its predecessor's priority. The priority of (3,1) as well as its predecessor (3) is defined to be larger than (2), so this operation will have a very similar priority to its predecessor, avoiding inconsistency. In the next section we will state some theorems and proofs showing that, among other properties, this list priority scheme always guarantees consistency.

7. Theorems and Proofs

We first prove a general theorem (the concurrent consistency theorem) equating the acyclic property of a graph to consistency of a concurrent execution. This theorem can be used for

many different types of priority schemes. Next we informally prove the correctness of any team automaton executing the dOPT algorithm by verifying that the properties of temporal ordering, semantic integrity, and consistency are all maintained. This latter property is proven by showing that the list priority scheme always results in an acyclic graph, and then applying the concurrent consistency theorem. This section presents the proof steps at a high level without using the precise but voluminous notation of team automata.

Definition: A *temporal precedence relation, t,* over a set E of events is a set of pairs of events such that (x,x) is not in t, and if (x,y) is in t, then (y,x) is not in t for all x and y. The interpretation of this is that (x,y) in t means that x occurs before y in time. We say that y has temporal precedence over x, $x <_t y$. Thus, in any valid execution of E, the event y is executed at its initiating site with full knowledge of the prior execution of x at the same site. Clearly, t viewed as a graph over E must be acyclic. This relation helps us to prove the temporal ordering property.

Definition: A *priority precedence relation, p,* over a set E of events is a set of pairs of events such that (x,x) is not in p, and if (x,y) is in p, then (y,x) is not in p for all x and y. The interpretation of this is that (x,y) in p means that for tie breaking purposes, x has a lower priority than y. We say that y has priority precedence over x, $x <_p y$. Thus, when need arises to decide which of two concurrent events should be chosen to appear first, p can be used, and x will be chosen to execute before y. Clearly, p viewed as a graph over E must be acyclic. There are many possibilities for the choice of p; judicious choice of this relation helps us to prove the consistency property.

Definition: Given a team automaton A executing the dOPT algorithm, we define a set E of events of A to be an *interaction set* if the elements of E are all non-commutative events operating upon the same session object (e.g. same position parameter).

Definition: Given an interaction set E, a temporal precedence relation t over E, and a priority precedence relation p over E, we define the *interaction graph* G(E, t, p) whose nodes are the elements of E, and whose edges are the elements of (t +* p). This is the set of all elements (u,v) of t augmented by all elements (x,y) of p such that (x,y) is not a member of t, and (y,x) is not a member of t.

Theorem 1 (The Concurrent Consistency Theorem): Every valid execution of an interaction set E maintains consistency iff the interaction graph G is acyclic.

Proof via contradiction:

(only if case) Given that every valid execution of E maintains consistency, suppose the interaction graph, G(E, t, p) is cyclic. Since there is a cycle, there are two chains of events of E which can be described as $x < x_1 < x_2 < ... < x_n < y$; and $y < y_1 < y_2 ... < y_m < x$. It is then possible to construct a valid execution in which one passive site has knowledge of the x to y chain inferring (x < y), and another passive site has knowledge of the y to x chain inferring (y < x). This construction can be done by judiciously arranging the order in which messages arrive at these two sites. Thus the events x and y will be executed in opposite orders at the two sites. Since events are non-commutative, the resulting site object sets will be non-iden-

tical. This contradicts our working hypothesis, so it must be the case that the interaction graph G is not cyclic.

(if case) Given that the interaction graph is acyclic, suppose that there is a valid execution of E which results in inconsistency. This means that at two or more sites, the site object sets are different at quiescence. This in turn means that at least one pair of events, x and y, were executed in different orders. One of the sites had local knowledge of a chain of events inferring $x < y$. Another site had local knowledge of a chain of events inferring $y < x$. From these two we can construct an argument that $x < y < x$. Since the interaction graph is acyclic, this is clearly a contradiction. Q.E.D..

Theorem 2 (The dOPT Correctness Theorem): If the transform matrix T maintains semantic integrity, then every admissible execution of a team automaton using the dOPT algorithm maintains correctness.

Proof:

(Serial Execution case) Since execution is not concurrent, we know that the incoming state vector, s_j is never less than the current state vector at the current site, s_i. If $s_j = s_i$, then execution at the receiving site is performed immediately, and the order of execution at the receiving site is the same as at the sending site, so temporal ordering is maintained. If $s_j > s_i$, then the execution of the event is delayed by putting the incoming message in a queue until the messages that preceded s_j have all arrived and been executed. In both of these casès, no transformations are performed, and the order of execution at the receiver is identical to the order of execution at the sender. Thus, the temporal ordering property is maintained.

Since no transformations are performed, there is no need to use the (tie breaking) priority precedence relation p. Therefore the precedence, (t +* p) reduces to simply t. Recall that the temporal precedence relation t can have no cycles, since event a cannot occur before b, with b also before a. The graph is acyclic, and no transforms are applied in this case. Thus, the consistency property is maintained.

Since it is given that the transform maintains semantic integrity, all three properties necessary for correctness have been shown to hold. This proves correctness in the non-concurrent case.

(Concurrent Execution case) By definition (list priority), the priority of an event is a list composed of the originating site identifier of the event concatenated onto the end of the priority list of its maximal predecessor. The priority denoted by list l_2 is greater than list l_1 iff the value of the first entry in which the two lists differ is larger in l_2; or the values of all entries which they share in common are equal, but l_2 is a longer list than l_1. This implies that if event 2 occurs temporally after event 1, then it will always have a priority which is greater than its predecessor. Therefore, for all events x and y, the case can never arise of $x <_t y$, and $y <_p x$. Therefore, no loops can occur in the interaction graph (t +* p). We can then apply theorem 1 to conclude that the consistency property is maintained.

Consider the construction of a passive automaton which only receives all of the events generated by the other sites. For any pair of events, there is an ordering of the arrival of those events such that no transformation is necessary. For non-concurrent events, it is the order of their initiation. For concurrent events, they should arrive in priority order (lowest p first; highest p last). We can thus arrange the order of arrival of events to this passive site such that no transforms are done on this site. Therefore, this site is guaranteed to maintain the temporal ordering property. Since we have shown that the site object sets will be identical at all sites (consistency,) the temporal ordering property is maintained at all sites.

Since it is given that the transform maintains semantic integrity, all three properties necessary for correctness have been shown to hold. This proves correctness in the concurrent case. Q.E.D..

Theorem 3 (The GROVE Correctness Theorem): Assume the standard textedit semantics for insert and delete. Assume the automaton is using the GROVE transform matrix. Then, every admissible execution of a team automaton using the dOPT algorithm maintains correctness.

Proof:

(Serial Execution case) We must verify that the dOPT algorithm fulfills the standard textedit semantics of insert and delete. In the case of no overlap of operations at different sites, the algorithm performs no transformations. Thus the generated events of the form insert[X; P] and delete[P] are directly executed as previously defined with the appropriate semantics.

(Concurrent Execution case) We must verify that the transform T yields the concurrent edit semantics as previously defined: o_i' = [the event o_i applied at position P+j-k]. This semantic statement is unconcerned with which of several inserts to the same position is placed first. That is a concern of consistency, not semantic integrity. Notice that the algorithms specified for T have +1 operations within the insert, and -1 operations within the delete. Thus, explicitly via this transform, or implicitly via other insertions or deletions, j additions, and k subtractions are performed in some order; potentially different orders at different sites. Since these operations are commutative, the distributed implementation exactly realizes the desired summation effect. Thus semantic integrity is maintained.

We can simply apply theorem 2 to prove the temporal ordering property and the consistency property. These, together with the semantic integrity property proved above, imply correctness. Q.E.D..

8. Conclusions and Future Developments

This paper has introduced the notion of groupware, and presented a novel algorithm and proof for concurrency control within real-time groupware. Groupware reflects a change in emphasis from using the computer to solve problems to using the computer to facilitate human interaction. For these systems to be accepted requires fine-granularity sharing of data, rapid response time, and rapid notification time. Therefore, the algorithm presented does not use locking, performs non-serializable sets of operations, and works in a workstation environment with replicated data and distributed control.

The concept of Team Automaton was introduced to characterize and prove the correctness of the algorithm. In the latter part of the paper, we developed a rather general theorem which allowed us to prove consistency of the replicated object set by examining properties of the combined precedence graph.

The model, the algorithm and the proof were presented in the context of GROVE, a group editor which we have implemented within the Software Technology Program at MCC. They also are applicable to other groupware such as the rIBIS group hypertext system [Rein90]. Notice that in these systems, users frequently apply associative access techniques rather than accessing objects by name. Thus, within a text editing application, users browse and point at the items they want to update; the individual characters are not given separate immutable names or unique addresses.

A primary difference between these systems, and the majority of database systems, is the visibility criterion. DBMS's are constructed with an intent to hide the presence of other users (transactions, locking, etc.); groupware is constructed with the intent of making visible the presence and state of other participants. Future work of our research team includes embellishing the group user interface to better make this happen. There is also work in progress to generalize the characterization of our transform matrix, and extending our proof technique to encompass other systems. This has been done to some extent with rIBIS. We will also continue to incorporate these ideas within other groupware which we are constructing, and will be constructing in the future. One challenging issue is the implementation of the UNDO function because there is only a partial ordering on previous operations. We also have some ideas for the implementation of further DWWM features. We have been developing and applying the notions of *team automata* to model and prove various other properties of groupware. We are also exploring alternative models. We hope that this will be useful in further correctness proof extensions. In conclusion, we believe that the new emphasis on groupware suggests a number of interesting and challenging frontiers. Perhaps this paper can help to stimulate work at these frontiers.

Acknowledgments

Special thanks to Simon Gibbs, co-author of our previous SIGMOD paper on this subject, and chief designer / programmer of GROVE. His creative ideas and programming have been critical to the success of this project. Thanks to Gail Rein, the other member of the groupware team who put in much effort to help implement and evaluate GROVE and its user interface.

References

[Back88] Back, R., and Kurki-Suonio, R. Distributed Cooperation with Action Systems, *ACM TOPLAS,* 10(4), pp. 513-554, 1988.

[Banc85] Bancilhon, G., Kim, W., and Korth, H.F. A Model of CAD Transactions, *Proceedings of the Intl. Conference. on Very Large Data Bases*, pp. 25-33, 1985.

[Bern87] Bernstein, P., Goodman, N., and Hadzilacos, V. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Chan88] Chandy, M., and Misra, J. *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[Char87] Charlesworth, A. The Multiway Rendezvous, *ACM TOPLAS*, 9(3), pp. 350-366, 1987.

[Cohe88] Cohen, M., Fish, R., Kraut, R., and Leland, M. Quilt: A Collaborative Tool for cooperative writing, *Proc. ACM SIGOIS Conference*, pp. 30-37, 1988.

[CSCW86, CSCW88, and CSCW90] *Conferences on Computer-Supported Cooperative Work*; Proceedings editor: ACM, New York, N.Y.

[Elli87] Ellis, C.A., and Ege, A. Design and Implementation of Gordion, an Object Base Management System, *Proceedings of the International Conference on Data Engineering*, 1987.

[Elli88] Ellis, C.A., Gibbs, S. J., and Rein, G. Design and Use of a Group Editor, *Report STP-263-88, MCC Software Technology Program*, Austin, Texas, 1988.

[Elli89] Ellis, C.A., Rein, G.L., and Jarvenpaa, S.L. Nick Experimentation: Some Selected Results, *Hawaii International Conference on System Sciences*, 1989.

[Elli90a] Ellis, C.A., and Malmquist, J., Team Automata and Teamsheets, *Report STP-131-90, MCC Software Technology Program*, Austin, Texas, 1988.

[Elli90b] Ellis, C.A., Gibbs, S.J., and Rein, G. Groupware: The Research and Development Issues, To appear in *Communications of the ACM*, December 1990, also available as MCC Software Technology Program technical report.

[Evan89] Evangelist, M., Nissim Francez, and Shmuel Katz. Multi-Party Interactions for Interprocess Communication and Synchronization. *IEEE-TSE 15(11)*, pp. 1417-1426, 1989.

[Fran89] Francez, N., Cooperative Proofs for Distributed Programs with Multi-Party Interactions. *IPL 32*, pp. 235-242, 1989.

[Fran90] Francez, N., and Forman, I., *Interaction Processes: A Multiparty Approach to Coordinated Distributed Programming*, Forthcoming book,1990.

[Form89] Forman, I. Design by Decomposition of Multiparty Interactions in RADDLE87, *Proceedings of the Fifth International Workshop on Software Specification and Design*, 1989.

[Grie86] Grief, I., Seliger, R., and Weihl, W. Atomic Data Abstractions in Distributed Collaborative Editing System, *Proceedings of the 13th Annual Symposium on Principles of Programming Languages*, pp. 160-172, 1986.

[Grud88] Grudin, J. Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces, *Proceedings of the CSCW'88*, 1988.

[Hoar78] Hoare, C.A.R. Communicating Sequential Processes, *Communications of the ACM*, 21(8), pp.666-677, August 1978.

[Kill90] Killen, L., et.al. ShrEdit: A Shared Text Editor - A User's Manual, *University of Michigan Internal Report* 1990.

[Knis90] Knister, M., and Prakash, A., DistEdit: A Distributed Toolkit for Supporting Multiple Editors, *Proceedings of the CSCW'90*, 1990.

[Lamp78] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM* 21(7), pp. 558-565, 1987.

[Lant86] Lantz, K.A. An Experiment in Integrated Multimedia Conferencing, *Proceedings of the CSCW'86*, pp.267-275, 1986.

[Lewi88] Lewis, B. and Hodges, J. Shared Books: Collaborative Publication Management for an Office Information System, *Proceedings of the Fourth ACM SIGOIS Conference on Office Information Systems*, pp. 197-204, 1988.

[Lync87] Lynch, N. and Tuttle, M. Hierarchical Correctness Proofs for Distributed Algorithms, *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987.

[Lync88] Lynch, N., et.al. A Theory of Atomic Transactions, *MIT Laboratory fof Computer Science Report LCS/TM-362*, 1988.

[Malm89] Malmquist, J. Private conversations.

[Lamp78] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM* 21(7), pp. 558-565, 1987.

[Olson90] Olson, G., and Olson, J., User-Centered Design of Collaboration Technology, *University of Michigan Report*, to appear in *Organizational Computing 1,1* January, 1991.

[Sari85] Sarin, S. and Grief, I. Computer-Based Real-Time Conferences, *Computer*, 18(10), pp. 33-45, 1985.

[Stef87] Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S., and Suchman, L. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings, *CACM 30(1)*, pp. 32-47,1987.

[Yeh89] Yeh, S., Ellis, C., Ege, A., and Korth, H. Performance Analysis of Two Concurrency Control Schemes for Design Environments, *Information Sciences Journal 49(1)*, 1989.