A New Approach to Collaborative Frameworks using Shared Objects

Aaron Ceglar and Paul Calder
Flinders University of South Australia
School of Informatics and Engineering
P.O.Box 2100, South Australia
{ceglar,calder}@infoeng.flinders.edu.au

Abstract

Multi-user graphical applications currently require the creation of a set of interface objects to maintain each participating display. The concept of shared objects allows a single object instance to be used in multiple contexts concurrently. This provides a novel way of reducing collaborative overheads by requiring the maintenance of only a single set of interface objects. This paper presents the concept of a shared-object collaborative framework and illustrates how the concept can be incorporated into an existing object-oriented toolkit.

1. Introduction

Multi-user applications can be used effectively across many industry and research domains to facilitate team dynamics, allowing geographically separated individuals to work concurrently in the same environment. These programs are often called groupware or collaborative applications, and support single application use by multiple concurrent users.

The inclusion of collaborative functionality is provided by integrating a generic framework upon which application specific collaborative semantics can be implemented within an application toolkit. This allows the use of a single framework to develop domain-independent collaborative applications, providing flexibility and reuse. Generic collaborative frameworks has been an area of research for the past decade, resulting in commercial collaborative products such as Microsoft's NetMeeting [7] and toolkits such as JSDT for Java (released June 1999).

So far, collaborative research has focused upon the development of these frameworks using replicated architectures and procedural languages. Although more stable and secure, centralised architectures have largely been overlooked due to problems with maintaining satisfactory interactive response. However for low latency environments with continually increasing communication bandwidth centralised architectures are an attractive alternative to replicated architectures.

Shared objects [3], provide an opportunity to investigate collaborative applications built upon a centralised framework. The use of shared objects will simplify the implementation of a collaborative object-oriented (OO) application by reducing the number of objects within the application, and by removing multiple-view dependency structures that would otherwise be required.

We present a novel approach to designing generic OO collaborative frameworks, by using the concept of shared objects. The inclusion of shared objects results in derived applications benefiting from reduced overheads as a single set of shared interface objects can be shared over the collaborative environment.

This paper is structured as follows, Sections 2 and 3 provide an overview of the shared-object concept and the relevant collaborative issues that need to be addressed within a collaborative framework. Section 4 and 5 discusses the design and implementation details. Finally sections 6 and 7 suggest directions for further work and draws some conclusions.

2. Shared Objects

Object oriented technology is widely accepted as a suitable methodology for the construction of applications. However, this approach can lead to a higher resource consumption than other methodologies. This is due to the high overheads associated with the use of fine-grained objects and the existence of replicated structures to support separate data and view objects.

Traditional data-view structural models [24, 10] allow the definition of several different views for the same application data (Figure 1). When application data is

1530-0900/01 $10.00 © 2001 IEEE

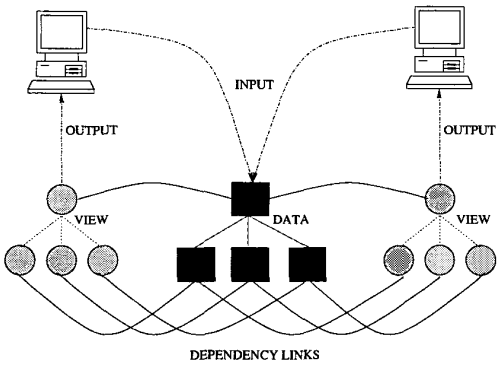

Figure 1. Data-view structural model

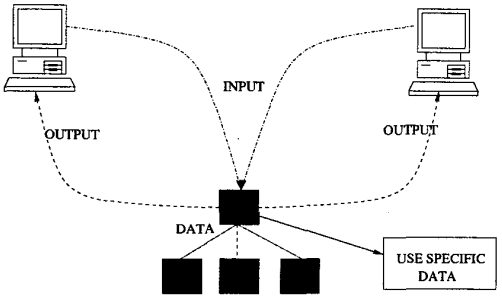

Figure 2. Shared-object structural model

modified using this model, all dependant views are notified to reflect the new state. However when large numbers of objects are instantiated within an application, the resources required to store the dependency information can be significant.

An alternative approach is to use a single set of objects that is shared between each view, as shown in Figure 2. The sharing of objects eliminates structural redundancy as a single object can assume all roles, for example: data and view. Sharing also reduces the number of object instances required, since reference to a single object can be made wherever an object of that type is required. The method utilised by Calder and Weston [3] in making objects shareable was to eliminate the use-specific data stored within the object. This information describes how the object is used, in comparison to instance-specific data that describes the object, distinguishing it from other objects. By externally storing an object's use-specific information, the object becomes generic, allowing it to be used in multiple places. The use-specific information is then passed as required to the object's methods, allowing it to behave differently for different instances.

3. Collaborative Issues

The inclusion of collaborative support within an environment requires the addressing of complex issues including architectural selection, coordination, global awareness, and the control of access and administration. The following subsections address these issues.

3.1. Architecture

Selection of a particular architecture can have a substantial impact on the run-time performance, functionality and scalability of derived collaborative applications.

Centralised architectures execute a single instance of the collaborative application and maintain one copy of the shared data, allowing the single application instance to process multi-I/O and support collaborative functionality.

Replicated architectures require individual nodes to manage data sharing among themselves to provide a collaborative environment. Support is provided to facilitate the coordination of the nodes to keep the state of each node consistent (eg. dOPT [16] and DistEdit [8]).

The replicated architecture's strengths lie in its user responsiveness and robustness. Robustness is increased because there are no single failure points. User responsiveness is improved because the user interacts with a local copy of the shared data. Communication between participating nodes is therefore generally more efficient, as high level I/O events rather than low level hardware I/O events are transferred [24]. The main disadvantage of this architecture is the complexity required to maintain state synchronisation between collaborative clients. This increases code complexity and processing time [17, 5, 16, 23, 11, 1, 14]. For example update latency between individual nodes can result in two users concurrently duplicating a task, without the other user being aware of the duplication. This may result in a unstable state that requires a form of rollback functionality.

The main advantage of a centralised architecture is the absence of synchronisation, as clients communicate via sequential I/O with only a single copy of the shared data. This functional simplification results in the ability to specify simpler coordination algorithms. The single copy of shared data also simplifies other replicated architectural problems (eg. concurrently duplicating tasks). The disadvantage of centralised architectures is the increased response lag, this is because of network

latency as there is no local processing and also bandwidth limitations as large quantities of low-level communication is required, transferring not only primitive data but also application interfaces PrSh97,PrSh94.

Hybrid architectures currently being researched may provide the best of both replicated and centralised architectures. Two examples are the Corona and Jupitor systems. Corona [21] involves a designated central node, allowing for the smooth integration of latecomers by providing a timely version of the collaborative application without interrupting active nodes. Jupitor [15] considers a collaborative application to consist of application objects, which provide the functionality, and interface objects, which provide the interface to the application objects. In a collaborative environment, the application objects remain centralised and the interface objects are replicated at client sites. This architecture provides the advantages of a centralised architecture, whilst minimising communication overheads by maintaining local copies of interface objects.

We suggest that although centralised architectures currently have an inferior interactive response they present a simpler collaborative model that will become more attractive in environments containing low latency and high bandwidth.

3.2. Coordination

When multiple users collaborate through jointly manipulating a shared object there is a need to synchronise actions. Concurrency control has been studied extensively in relation to database systems, where the aim is usually to provide serializable transactions [5]. This model is based on the premise that database users expect their transaction not to be interfered with by any other transactions. However collaborative participants are typically interactive and mechanisms must be provided to handle interference of an activity by another user. In fact, this may be the required behaviour and therefore the serializable model implemented by databases is too restrictive for collaborative use.

Early collaborative applications used floor control (eg. one person interacting at a time) to prevent conflict, in a model similar to the database transaction model [5]. Later models dispensed with synchronisation altogether, relying upon social protocols and global awareness to prevent conflict. However these methods are not acceptable in many situations [14].

Coordination policy literature [17, 16, 23, 11, 14, 8, 24] focuses on replicated architectures and the inherent problems stemming from the creation of an integrated collaborative environment using loosely coupled geographically dispersed nodes. Although coordination policies must still be implemented in a centralised architecture, the central processing of information and the maintenance of a single application copy simplifies coordination implementation.

3.3. Global Awareness

Global awareness facilitates multi-user coordination within a collaborative application by providing users with information regarding other collaborators, allowing individual users to maintain a global perspective of the collaboration [22]. The implementation of awareness functionality facilitates coordination between the users, reducing conflicts and task duplication. According to Koch et al. [9] there are four common mechanisms used to provide awareness information.

Status is information regarding the collaborative participants and directly relates to the functionality specified by informal, structural and social awareness.

Events are the means by which real-time workspace awareness is maintained, enabling spontaneous interaction between users and maintaining environment context.

History refers to the maintenance of an application activity log. The inclusion of this functionality at a fine level allows for the rollback of application modifications and also for activity queries such as "What modifications have been made to the application since I last participated".

Filtering of awareness information can be required for two reasons: privacy and reducing information flow.

Global awareness is required for effective multi-user collaboration, but there is a trade-off between increased functionality and additional network latency.

3.4. Access and Administration Control

Access and administration control have been addressed extensively in non-collaborative domains (eg. operating and database systems). These domains provide access control through primitives such as access lists and capability lists in order to protect objects from unauthorised activities. The effective administration of these access-control primitives is provided through central administration and object ownership [2]. While many aspects of these mechanisms can be reused in collaborative systems, the functionality needs to be extended to allow for the sharing of user states.

This extension allows for object activities to be defined based not only upon the state of the application, but also on the state of other users [20].

4. Framework Design

This section describes the issues that need to be considered when designing a centralised collaborative framework for an OO toolkit. It includes a discussion of the requirements to support multiple IO and the use of shared objects to reduce overheads. Also presented is a design for managing the dynamic participation of interfaces that incorporate the storage of global awareness and user specific information.

A collaborative environment requires the receiving of input from and the distribution of output to multiple interfaces. This process is simplified through the use of centralised architectures and shared objects. This overcomes the need for synchronisation and reduces the number of objects required by the environment.

In order to minimise lag within a collaborative environment, verification protocols are not used in internode communication. Hence some messages sent from a node may be lost. In a replicated architecture this will cause synchronisation errors, as different nodes may have received different messages. However in a centralised architecture this problem is alleviated as all processing occurs centrally. If an input message to the central node is lost there is no effect and if an output message to a particular interface is lost, the actual collaborative environment remains synchronised and the lost message will be compensated for in subsequent messages. Therefore multi I/O is simpler and more stable using a centralised architecture.

The use of a centralised architecture results in a sequential input stream from the collaborative environment to the central node. Hence as the number of collaborators increase an input bottleneck becomes apparent, reducing the model's scalability. However the replicated architecture requires the implementation of complex synchronisation algorithms to maintain consistency at all participating nodes. Although not influencing scalability, the processing overheads of synchronisation mean that in small environments the centralised architecture should be more efficient.

The inclusion of shared objects within the framework provides an effective means of multicasting output to the environment's participating interfaces in a flexible manner, allowing messages to be sent to selected interfaces. This can be achieved through the sharing of an OO toolkits base-interface object (Figure 3).

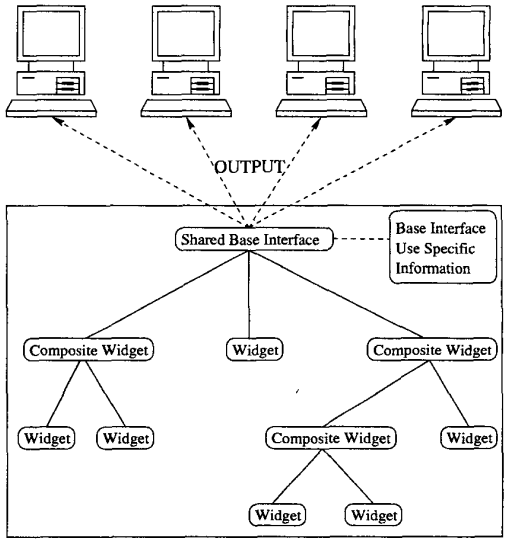

Figure 3. Shared Base-interface Object

In general, OO toolkits specify a base-interface object upon which other higher-level interface objects (eg. widgets) are defined. In a single-user application one base-interface object is required. However in a multi-user environment one instance of this object is required for each participating interface, each with its own set of higher-level interface objects (Figure 1). By creating a new shared version of the base-interface class, the replicated interface structures are removed, reducing the number of object instances and hence object overheads. The new shared base-interface object will require extensions to the original class methods and a mechanism to store the use-specific data (Figure 3).

The maintenance of a collaborative environment's current participants and the information pertaining to them is essential in the provision of a collaborative environment. This functionality is incorporated within the framework as a dynamic participation mechanism.

Dynamic participation relates to the runtime adding and removing of interfaces from the collaboration. This is simplified within a centralised architecture as the joining node will always receive the current interface state upon connection. The methodology used provides a controlled collaborative environment by requiring that the central node initiate the collaborative session and control the adding of new nodes. The alternative method is to allow remote users to request connections directly from the collaborative application including the initiation of a new collaborative session. This functionality is possible with a centralised architecture through the use of a server/thin client model

and will be discussed in further work (Section 6).

This participation mechanism is implemented within the framework as a singleton class [6] (as any derived application will only require one instance of the participation mechanism) containing a list of participants and methods to implement dynamic participation. Each list element defines a node and all relevant attributes. At a minimum this will consist of the node identity and the identity of the window created upon the node to host the collaborative application interface. Both the list and participation classes are extensible to allow for arbitrary user and global awareness functionality to be incorporated.

5. Implementation

This section details the implementation of the collaborative framework using the OO InterViews toolkit [13, 12, 19] and shared objects. A demonstration tool has also been constructed using the shared-object collaborative framework and is discussed elsewhere[4]. InterViews is a C++ GUI toolkit designed in the 1980's at Stanford University for research purposes and was selected due to source code availability and its current support for shared-object functionality.

5.1. Multiple Input & Output

The InterViews *Glyph* class represents the visible structure or interface of the application. It provides application functionality to control the geometry of the interface and its appearance. Defining compositions of sub-classed *Glyphs* creates an application's interface providing visible data of various types, for example characters, shapes and borders. The distinguishing characteristic of the *Glyph* class over similar components in other toolkits [10] is that they are shareable and provide the basis for the collaborative sharing of objects.

As previously discussed, object sharing will be implemented at the base interface level; by sharing this single *Glyph* the entire interface will in effect be shared.

This is achieved by externally storing a list (*SharedCanvas*) in which each element contains the attributes required to store and process information pertaining to individual interface allocation, for example information unique to each collaborative node's base interface *Glyph*. To send updates to the collaborating nodes, the relevant functions have been extended to process operations for each element in the *SharedCanvas*. For example, the redraw method that incurs damage upon the canvas area occupied by the *Glyph* is extended to incur damage upon each *Canvas* specified by the list. When the program redraws the *Glyph*, it does so for each *Canvas* in the list, updating all particitpating interfaces.

5.2. Dynamic Interfaces

The *DynamicAttendance* singleton class was created to provide support for the runtime creation and deletion of application interfaces. This class is required to maintain a list (*DynamicDisplay*) of the participating node's characteristics and to provide methods for the node's dynamic attendance. Each *DynamicDisplay* element uniquely identifies a node by its network address and provides pointers to the *Display* and *Window* objects, which are allocated to the node when it is appended to the collaboration environment. When a node is to be removed from the environment, it is located within the list and disconnected from the session.

When a node is added to the collaborative environment an application interface window will appear on the node's display with the application's current state. In order to receive subsequent interface updates the *Window* object's associated *Canvas* object and the base-interface object's allocation attributes must be appended to the *SharedCanvas* list. Similarly when a node is disconnected from the environment and removed from the *DynamicDisplay* list, its related element must be removed from the *SharedCanvas* list.

The incorporation of global awareness functionality is application dependent. This functionality should be implemented at the same level as the *DynamicAttendance* class, as global-awareness widgets are created and managed at the application's Window level. To support this the *DynamicAttendance* class and *DynamicDisplay* list are extensible, allowing application developers to derive sub-classes and specify the implementation of their own global awareness widgets within these structures.

In order to demonstrate the capability of the framework to allow the effective incorporation of application level global awareness widgets, tele-pointers have been incorporated at a framework level. Tele-pointers are the visible representation of the current positions of the other node's pointers within the collaborative environment.

5.3. Interaction

InterViews defines an *InputHandler* class that manages the handling of input upon an associated *Glyph*. This class was extended to provide support for the following functionality as required by the design:

- The concurrent interaction by multiple users upon an object.
- The provision for an object to have different states or behaviours.
- An awareness of the state of an object's environment.

A collaborative environment requires the management of an interface object's activity state because an arbitrary number of users can currently interact with an object, constrained only by the application's semantics. This is not required in a single-interface environment as the single interactor can only interact with a single object at a time and hence state information at an object level is not required, although it may be necessary at the application level. Multi-user interaction with an object requires the maintenance of a state attribute and an object-level list structure (*MultiControl*) identifying the objects current interactors.

The *MultiControl* elements contain information relating to the users currently accessing the object. It is expected that users will only be appended to the list if their interaction with the object exists for a period of time, for example dragging, however this may not be true for all applications. Therefore methods have been implemented giving application developers control over the maintenance of this list including adding and removing users, counting the number of current object interactors and querying if a specific user currently has control. The object's state attribute is required to allow an object to exhibit dynamic behaviour, depending upon the state of its environment and its interactor's actions.

A collaborative object requires knowledge of its environment to accurately determine its current state. In general an object's state will be determined by its own current activity. However its state can also depend upon the current state of its environment in particular its hierarchical descendants. For example an application's semantics may specify that an object can only be deleted if neither it nor any of its descendants are currently interacting. Hence whether the object may be deleted or not is determined not only by its own activity, but also by its descendant's activities.

In order to provide environmental awareness a hierarchical structure was embedded within the *MultiInputHandler* class. This structure maintains pointers to the object's parent and a list of the object's descendants; each element is of type *MultiInputHandler*. When a *MultiInputHandler* instance is created it is passed a pointer to its parent object, the object's constructor stores this pointer and appends the new instance to its parent's child–list. Additionally when a *MultiInputHandler* is to be deleted, it automatically removes itself from its parent's child-list.

The functionality relating to an object's child–list has been encapsulated within a new class called *ChildList* that maintains the listing of an object's children. In addition, the *ChildList* class also implements a method that transverses the *MultiInputHandler* hierarchy and initiates input events. This provides the basis for the multi-level interaction model which is described elsewhere [4].

5.4. Collaborative Glyph

The above implementation resulted in the extension of two core InterViews classes with the creation of additional supporting classes to manage the sharing of base-interface *Glyphs* and the embedding of an interaction hierarchy. However, it was realised that these two classes could be combined as they were both derived from the same shareable base class (*Glyph*).

This optimisation means that collaborative functionality can be implemented within derived applications through the sub-classing of a single class *CollaborativeGlyph*, that incorporates both input interaction and the multiple updating of multiple base-interface objects.

5.5. Demonstration Tool

A proof-of-concept tool was developed in conjunction with the development of the InterViews collaboration framework. It was used to provide feedback as to the current state of development. The tool comprises a simple graphical interface that allows for the collaborative manipulation of graphical shapes by multiple users. It incorporates both the *DynamicAttendance* class for the implementation of tele-pointers and the runtime maintinence participating nodes and also a derivation of the *CollaborativeGlyph* class that provides the required collaborative functionality and defines the graphic's objects interface and behaviour.

The collaborative use of this application provided powerful semantics by providing both global awareness and dynamic behaviour. The inclusion of multiple nodes instigates tele-pointers, showing each node's pointer position within the other participating node's screens. The collaborative functionality allowed for the addition, removal and dragging of objects, but these events are now constrained depending upon the targeted object's current environment. Also, dynamic behaviour was implemented by extending a object's drag function to incorporate the dynamic stretching of a object when there are two concurrent interactors.

6. Further Work

We have provided proof of concept through the extension of the InterViews toolkit, however additional testing and development is required to quantify the performance of the framework. Although the framework has been generically designed, there is no proof that it can be applied in all application domains. However the development of a range of applications using the existing framework, from simple to complex and incorporating different levels of interaction, would highlight any framework limitations.

An extension to this work is the ability to share different levels of a collaborative environment with different nodes. This would allow for the development of more complex and effective environments. Consider the following scenario:

A company's collaborative environment comprises of three separate applications. All of these applications required collaborative use. Depending upon a user's privileges, when they instigate a collaborative session they will only receive their authorised collaborative applications.

This illustrates the concept of constraining the sharing of glyphs to specific nodes. This functionality will provide further transmission optimisation, as only that node's authorised glyphs are sent. This method also has advantages in relation to system security by providing an effective method of governing glyph allocation, through an abstract Collaborative Environment Manager. Although the above example illustrates glyph-sharing constraints at a course level it could be applied to any Collaborative Glyph and hence can be implemented at a very fine level. The framework already enables the underlying structure to support this functionality, by allowing each glyph to specify its own set of shared canvases. This functionality requires the specification of methods to control the allocation and management of the multi-level glyphs.

Mobile objects refer to a concept by which objects are able to relocate to various nodes within the collaborative environment. This concept is taken from the Jupitor integrated architectural collaborative framework [15], provides centralised processing with replicated interface objects at each collaborative node. Such a process reduces communication overheads by allowing the transference of primitive data and not their visual representations over the network. Further research could extend the framework discussed in this paper to incorporate this concept. The theory is that by intelligently moving the shared interface objects to the vicinity of their utilisation instead of remaining on the central node, a reduction in communication overheads will result.

With respect to remote session instantiation, the framework discussed specifies that for any derived application, a collaborative session must first be instigated at the central node, creating the application process. Subsequently, participating nodes must connect a requesting node to the collaborative environment. Therefore the new node must make its request known to a current collaborative participant by an external means (eg. e-mail). In order to provide a fully flexible environment, remote instantiation is required. This should allow for not only new nodes to independently connect themselves to an existing collaborative session, but also the remote instantiation of a new session.

The inclusion of this functionality would require a form of client/server architectural model, whereby the server represents the central node and the client represents a participant node. However in order to maintain a centralised architecture the client would be thin, containing only the functionality required to enable the node to instigate or connect to an existing session. This inclusion would provide the mobility currently available within replicated architectures. However this architecture's problems in obtaining remote access (eg. obtaining current application state and knowledge of current participants) would be overcome.

7. Conclusion

This research has shown that by introducing the concept of shared objects to collaborative systems the required framework is simplified, through the elimination of redundant structures and the reduction of object instances. This work culminated in the integration of a shared collaborative framework within the InterViews toolkit. The core collaborative functionality has been encapsulated in the *CollaborativeGlyph* shareable class. This provides flexible and effective collaborative functionality and through inheritance and extensibility allows for the development of different types of application-specific shareable interface objects.

References

[1] G. Banavar, S. Doddapaneni, K. Miller, and B. Mukherjee. Rapidly Building Synchronous Collaborative Applications By Direct Manipulation. In *Computer Supported Cooperative Work*, pages 139–148, Seattle, Washington,USA, 1998. ACM.

[2] E. Bertino, S. Jajodia, and P. Samarati. Supporting multiple access control policies in database systems. In *IEEE Symposium on security and privacy*. IEEE, 1996.

[3] P. Calder and U. Weston. Building User Interfaces with Shared Objects. In *17th Australasian Computer Science Conference*, volume 16(1), pages 312–330, Christchurch, NZ, 1994. Australian Computer Science Communications.

[4] A. Ceglar. *Analysis and Implementation of Shared Objects in Collaborative Systems.* Honours thesis, Flinders University, 1999.

[5] C. Ellis and S. Gibbs. Concurrency Control in Groupware Systems. In *Management of Data*, pages 399–407. ACM SIGMOD, 1989.

[6] E. Gamma, R.Helm, R.Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Orientated Software.* Addison and Wesley Publishing Company, Reading, 1994.

[7] D. Garfinkel, B. Welti, and T. Yip. HP SharedX: a tool for real time collaboration. *HP*, 45(4):26–33, 1994.

[8] M. Knister and A. Prakash. DistEdit: A Distributed ToolKit for supporting Multiple Group Editors. In *Computer supported Cooperative Work*, pages 343 – 355. ACM Press, 1990.

[9] M. Koch, D. Kohler, and M. Burger. Awareness Information in WideArea Network. Technical report, Applied [Informatics and Distributed Systems Group, Department of Informatics, Technische Universitat Munchen, 1996.

[10] T. Lewis, editor. *Object Oriented Application Frameworks.* Manning Publication Co., Greenwich, 1995.

[11] D. Li and R. Muntz. COCA: Collaborative Objects Coordination Architecture. In *Supporting Group Work*, pages 179 – 181. ACM Press, 1998.

[12] M. Linton. Programming with InterViews. Technical report, Silicon Graphics, Mountain View, 1994.

[13] M. Linton, P. Calder, J. Interrante, S. Tang, and J. Vlissides. InterViews Reference Manual, Version 3.1. Technical report, Department of Computer Science, Stamford University, 1994 1994.

[14] J. Munson and P. Dewan. A Concurrency Control Framework for collaborative systems. Technical report, Department of Computer Science, University of North Carolina, 1996.

[15] D. Nicholos, P. Curtis, M. Dixon, and J.Lamping. High Latency, Low bandwidth Windowing in the Jupitor Collaboration System. In *UIST95*, Pittsburgh,PA, 1995. ACM Press.

[16] C. Palmer and G. Cormack. Operation Transformations for a distributed Shared Spreadsheet. Technical report, Department of Computer Science, University of Waterloo, 1998.

[17] A. Prakash and H. Shim. DistView: Support for Building Efficient Collaborative Applications Using Replicated Objects. In *Computer Supported Cooperative Work*, pages 153–164. ACM, 1994.

[18] A. Prakash and H. Shim. Data Management Issues and Tradeoffs in CSCW systems. Technical report, Software Systems Research Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, June 1997.

[19] M. Roseman. A Not-entirely Gentle Introduction to InterViews. Technical report, Department of Computer Science, University of Calgary, 1992.

[20] H. Shen and P. Dewan. Access Control for Collaborative Environments. Technical report, Department of Computer Sciences, Purdue University, 1992.

[21] H. Shim, R. Hall, A. Prakash, and F. Jahanian. Providing flexible Services for Managing Shared State in Collaborative Systems. In *European conference on Computer-Supported Cooperative Work*, pages 175–186. ACM Press, 1997.

[22] J. Stewart. Single Display Groupware. Technical report, Department of Computer Science, University of New Mexico, 1997.

[23] C. Sun and C. Ellis. Operational Transformations in Real-Time Group Editors: Issues, Algorithms and Achievements. In *Computer Supported Cooperative Work*, pages 59–68. ACM Press, 1998.

[24] C. Sun, X. Jia, Y. Zhang, and Y. Yang. A Generic Operation Transformation Scheme for Consistency Maintinence in Real-time Cooperative Editing Systems. In *Supporting Group Work*, pages 425 – 434. ACM, 1997.