GROUPKIT
A Groupware Toolkit for Building Real-Time Conferencing Applications

Mark Roseman
Saul Greenberg

Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4
(403) 220-6015 roseman@cpsc.ucalgary.ca

ABSTRACT
This paper presents our approach to the design of groupware toolkits for real-time work, and how the design is instantiated in our toolkit, GROUPKIT. The design is based on both the technical underpinnings necessary for real-time groupware, and on user-centered features identified by existing CSCW human factors work. We also present three strategies for building GROUPKIT's components. First, an extendible, object-oriented *run-time architecture* supports managing distributed processes and the communication between them. Second, *transparent overlays* offer a convenient method for adding general components to various groupware applications, for example supporting gestures via multiple cursors and annotation via sketching. Third, *open protocols* allow the groupware designer to create a wide range of interface and interaction policies, accommodating group differences in areas such as conference registration and floor control.

KEYWORDS
real-time groupware, toolkit, development tools

INTRODUCTION
Real-time computer conferencing applications have been a major focus of groupware development efforts. Often motivated by CSCW research, we have recently seen the creation of numerous systems, such as shared text editors [23], freehand sketching systems [13,22], structured drawing programs [15], and group support systems [24]. Yet construction of these applications is fraught with difficulties. Besides all the normal problems of building single-user applications, the groupware developer must be concerned with technical issues such as synchronization, concurrency, communications, registration and more. The developer must also understand and incorporate the fundamental CSCW human factors issues identified for effective group work. *Groupware toolkits* are now emerging that address some of these issues. By providing the key components for common groupware needs, the toolkits can reduce development effort, enable rapid prototyping, and increase product quality of multi-user applications.

Of course, there are many potential groupware applications, and no one toolkit could cover all possibilities. In our case, we are concerned with toolkits that assist in constructing real-time *work surfaces* — shared visual environments where one user's actions are made immediately visible to other users [11]. These work surfaces are useful in both face to face and geographically distributed meetings. At its simplest, a work surface is a "What You See Is What I See" (WYSIWIS) space [27] that conference participants use for creating, importing, displaying, and interacting with their artifacts — usually text and graphics of some sort. Thus work surfaces include generic applications such as shared windows, whiteboards, structured drawing systems, and shared editors. But work surfaces can be far more powerful and specialized than that. For example, WYSIWIS could be relaxed so that people could view the work surface from different perspectives; an example would be a competitive card game [26]. Specialized applications could expand the notion of work surface to include tools for voting, brainstorming, organizing ideas, and so on.

This paper discusses our toolkit design philosophy, as embodied in GROUPKIT. We begin by listing a preliminary set of requirements for groupware toolkits, derived from both human factors work and technical innovations in CSCW. The following section presents an overview of GROUPKIT, with a detailed description of its technical architecture. Examples of three strategies for designing toolkit component are then given.

TOOLKIT DESIGN REQUIREMENTS

In this section we propose and defend some of the underlying design requirements we feel are important for a real-time conferencing toolkit. We do not believe this list is complete, nor will every item be necessary for every groupware application. However, we do feel the requirements listed will apply to a wide range of real time conferencing applications, and that they provide a reasonable starting point for discussion. Requirements listed in the first section are human-centered, and those in the second are programmer-centered.

Human-centered design requirements

Supporting multi-user actions over a visual work surface.
We believe there are several general but critical activities that people do over a shared work surface, regardless of its contents. Two of these, gesturing and annotation, are described below.

a) Provide support for gesturing. Researchers at Xerox PARC studied the use of conventional drawing surfaces by small groups [28]. A critical finding was that participants frequently gesture over the drawing surface: to enact ideas, to signal turn-taking, to focus the attention of the group, and to reference objects on the surface. Several recent computer systems emulating group drawing surfaces support gesturing with multiple cursors appearing on all displays [13,2,15], and their usability studies confirm the ubiquity of gesturing. We believe that gesturing can enhance communication in many diverse types of conferences, and should be supported at the toolkit level.

b) Provide support for graphical annotation. The Xerox studies also noticed many instances of annotations made to existing drawings, serving both as gestures (eg underlining text while saying "this one here") and as meta-level notes. Several systems now incorporate graphical annotations of their objects. FREESTYLE users, for example, can verbally and graphically annotate bitmap snapshots; the results can be mailed to others who can then play back the transcript [9]. Both PROOF-MARKS in *vmacs* [18], and the commercial MARKUP application allow comments and markup symbols to be added to written documents. As with gesturing, we believe that real-time group graphical annotation over a work surface is useful in many situations and should be supported by the toolkit.

Structuring group processes during a meeting
Some researchers believe that groupware should impose a social model of interaction on the group. This is an explicit attempt, based on management theory, to provide methods for keeping the group on task, enforcing roles and commitments, and making the group more efficient and productive. There is certainly controversy between those who believe that social protocol should be determined only by the group members (eg [7]), by the software (eg [24]), and somewhere in between (eg [16,14]). We believe that some group process primitives should be provided by the toolkit, accommodating groupware that wishes to control meeting structure. The list below discusses only a few group process requirements.

c) Provide various floor control policies. Floor control or turn-taking mechanisms provide a way to mediate access to shared work items. Lauwers [19] and Greenberg [14] recommend that systems should "support a broad range of [floor control] policies" to suit the users' needs. Systems such as SHAREDX [10] and ASPECTS (from Group Technologies) support a few different policies, while SHARE [14] strives to provide complete flexibility. Floor control can be important in many situations, such as shared screens allowing only serial interaction, or systems following strict interaction models, such as a teacher controlling which students can access the work surface.

d) Support different registration methods. Another part of group process controls who is allowed to join the meeting. For some meetings, anyone may be allowed to join. For others only a select group can participate, or perhaps new users must be "sponsored" by an existing user. Sometimes more spontaneous creation of conferences is desired [17] while other situations require a central facilitator to handle registration [24]. Toolkits should provide the flexibility to support any reasonable registration process.

e) Support latecomers to the conference. A consequence of spontaneous conferences is that all users will not join the conference at the start. Provisions should exist allowing newcomers to join at any time, as well as allowing existing members to leave. Strategies must also be supported to assist the newcomers in "getting up to speed." This may involve simply sending the current conference state to the new user [15] or providing summary information on how the conference has progressed over its lifetime.

Integration with conventional ways of doing work.
Groupware should not impose a barrier between "individual" and "group" ways of working. For example, the system should provide group members with ready access to their individual work, and allow them to import it to a conference. Additionally, all normal communication channels (eg telephone, email) should be readily available.

f) Integrate other forms of communication. Voice communication is an important factor in most conferences [4], and given the ubiquity of telephones, we should assume that a voice channel is available. While many real-time conferencing systems assume a voice channel is present, they do not explicitly support creating voice links. Ideally, there should be a mechanism in the conferencing system to establish voice conferencing — perhaps automatically when a data conference is started. Similar arguments hold for other channels that may be available such as video links.

g) Allow use of single-user applications. Soon, most computer applications may be designed to support multiple users. Unfortunately, most of today's applications support single users only. There are several reasons why single-user

programs should be available in multi-user conference settings. Groupware counterparts to single-user programs may not exist; a person's work may be accessible only through a particular application; people are skilled on particular applications. Conference users should be able to view and interact with single-user systems through shared screen or shared windows eg SHARE [12] and SHAREDX [10]. The toolkit should provide shared windows or the means to incorporate other shared window systems.

Programmer-centered design requirements

Technical support of multiple and distributed processes. Most groupware, especially for geographically distributed conferences, will require an architecture where multiple (perhaps distributed) processes can communicate with each other. While most operating systems provide process control and inter-process communication, the programmer's job of initiating, maintaining and tearing down processes and their communication channels is a tedious one. As well, state information about sessions may need to survive beyond the lifetime of a single process or meeting.

h) Provide processes for basic conference management. Groupware applications must oversee all conference management, which include activities such as participant registration, initiation and teardown of meeting processes, communications, and so on. Groupware toolkits have placed much emphasis in providing capabilities for conference management eg LIZA [11], CONFERENCE TOOLKIT [3], and MMCONF [5]. We too believe that the basic run-time infrastructure for conference management must be supplied by the toolkit.

i) Provide a robust communications infrastructure. Any groupware toolkit must provide the communications facilities on which to build conferencing components. At the very least, it must be possible for any process to send messages to specific processes owned by conference users, and it is preferable if a multi-cast facility is available to broadcast a single message to all. Of course, the communications demands will depend heavily on the way the process architecture is determined. The trade-offs between centralized and replicated architectures are well-documented [20], with centralized architectures simplifying concurrency control and replicated architectures being more efficient and robust to machine failure.

j) Provide support for persistent sessions. Often computer conferences will span more than a single session, for example decision support meetings [24]. It is desirable to maintain session state information over the full duration of the conference. There should exist a general mechanism whereby conference objects can be made persistent.

Technical support of a graphics model. A visual work surface will require graphical and textual primitives. Yet shared graphics require several capabilities that are not present in single user systems.

k) Provide primitives to a shared graphics library. Many groupware applications require graphical library primitives for creating multi-user objects such as shared lines, rectangles, circles and text. Greenberg, Roseman et al's discussion of GROUPDRAW [15] describes technical issues of a shared object-oriented drawing package, and provides their design of an abstract drawing object that can be sub-classed into concrete objects such as shared lines [15]. Similarly, the fine-grained editing of simple graphics and text objects in Bier and Freeman's MMM system gives insight into how shared objects should behave [1]. Similar extendible graphics libraries should be provided by toolkits, so that programmers can easily create shared interactive graphical objects on the display.

l) Provide object concurrency control. Many groupware conferencing systems support access to some type of shared object, be it structured graphics or a text buffer. Concurrency control is often needed to mediate access to the object, for example, two people trying to manipulate the same point on a line. In fact several concurrency schemes have already been implemented in groupware toolkits [25,26,11]. Concurrency can be achieved through simple locking, transaction mechanisms, or numerous other schemes [8]. In addition, the degree of concurrency and access to shared objects can be specified through the notion of flexible coupling [6].

m) Separate the view of an object from its underlying representation. Many single-user graphical systems separate the properties of an object from its view on the screen. Patterson argues that this separation is critical in groupware [25,26], and that abstractions should be used to create an interface-independent representation of data. As a consequence, users can have multiple perspectives on the same data.

GROUPKIT

GROUPKIT is a toolkit a programmer can use to develop real-time computer conferencing applications for geographically distributed or face to face meetings. It provides not only support for basic connectivity, but also support for higher-level human factors concerns. As it is still being developed, we have not yet incorporated all our design principles. However components are provided for flexible policy conference registration, communications support, flexible floor control, and gestural and annotative communication.

GROUPKIT is written in C++, on top of the INTERVIEWS toolkit [21]. Applications built using GROUPKIT will run on any machine supporting INTERVIEWS, that is, Unix workstations running X-Windows. GROUPKIT relies on the INTERVIEWS glyph mechanism for its user interface constructs. Glyphs are lightweight objects (similar to widgets in other toolkits) that are composed to make interfaces. The INTERVIEWS Dispatch library — a front end to normal Unix sockets — is used for communications.

Process and Communications Run-Time Support

GROUPKIT's technical infrastructure is based upon a replicated architecture and communications support, as illustrated in Figure 1. The infrastructure supports conference registration and the subsequent communication between processes owned by conference participants. This is comparable to the communications support in RENDEZVOUS [25] and the CONFERENCE TOOLKIT [3]. The remainder of this section describes the infrastructure components by walking through an example scenario.

Registration. Suppose a user wants to create a shared drawing surface, perhaps to discuss a design problem with a remotely located colleague. On initiating the GROUPKIT-based program, the user must first create a conference. This is done through the *Registrar Client*, which provides an interface to the central *Registrar*.

The Registrar allows the user to create, join or leave one or more conferences. The Registrar Client is responsible for implementing particular registration policies, eg deciding who enters the conference, how they do so, and what the interface looks like. While GROUPKIT allows new Clients to be programmed, it also provides a library of predefined Registrar Clients implementing particular registration policies and user interfaces. One novel aspect of this scheme is that it allows group members to use different Registrar Clients to enter a single conference.

The Registrar is an independent process (invisible to the user) that maintains a list of all conferences and their users. One central Registrar would exist at each installation. The Registrar itself is *policy-free*, and leaves it to the Registrar Clients to implement a particular registration policy and to present a reasonable interface to the user. This allows different policies and interfaces to be implemented in order to accommodate group differences.

Conference Initiation. In our scenario, the user has just requested a new conference through the Registrar Client, which in turn passes the request on to the Registrar. Next, the Registrar Client asks the *Coordinator* to create a new *Conference* object. The Coordinator acts as an intermediary between the Registrar Client and application Conferences, permitting multiple conferences (eg sketching and editing applications) to share a common registration mechanism. Its main duties are to create Conferences at the request of the Registrar Client, and to direct requests from the Client to the appropriate Conference.

The Coordinator then spawns a new process which instantiates a Conference object. It is the Conference object that actually runs the specific groupware application. GROUPKIT provides a generic Conference object, which the developer may subclass to provide extra functionality required by the application. In this case, the generic

Figure 1. Communications infrastructure of GROUPKIT, showing message passing between objects. Here, objects owned by one user (rightmost Registrar Client, Coordinator, and Conference) interact with objects owned by another user, as well as the central Registrar. Small font text indicates the message passing protocol.

Conference object is used, along with a programmer-defined glyph (graphical interface) supporting a shared drawing surface.

Conference Maintenance. Other users can also create Conference objects. As each Conference locates the meeting participants via the Registrar, communications channels will be opened between all Conference objects. Facilities in the generic Conference are provided for exchanging messages with other user processes.

Communications between distributed processes are maintained by *messaging objects*. The two types of messaging objects ("Writer" and "Reader") provide a convenient method of communicating with processes owned by remote conference participants. These objects, derived from the INTERVIEWS Dispatch library, provide a primitive Remote Procedure Call (RPC) facility. Writer objects specify messages they can send, while Reader objects provide functions to be called when messages are received.

Conference Leaving and Termination. As with initiation, conference leaving and termination is handled through the Registrar Client. If a user wishes to leave the conference without terminating it, their Registrar Client sends the `delete-user` message (Figure 1) to the Registrar. Some Clients may permit explicit conference termination, allowing any user to terminate the conference, while more typically the Registrar Client for the last user to leave will terminate the conference. This is done by sending a `delete-conference` message to the Registrar, which will be rebroadcast to any remaining users' Registrar Clients.

Overlays

GROUPKIT provides components that may be included in any Conference object through an *overlay* strategy. Overlays are transparent "windows" placed on top of the main application graphics shown by the Conference object. Currently, overlay components have been implemented for gestural communication (via multiple cursors over a surface) and annotative communication (via freehand drawing over a surface). The motivation is that such components could be useful for a variety of groupware applications, as mentioned earlier.

Figure 2 provides a conceptual picture of adding an overlay for displaying multiple cursors on an existing application graphic. The transparent cursor overlay is written as an INTERVIEWS glyph that overlays any other glyph. Neither the cursor glyph nor the main application glyph need any knowledge of the other. Input events are received by the cursor glyph, which updates cursors as necessary. The event (eg cursor motion) is then passed to the application glyph, to use as needed. The application glyph need not even know the event went through the cursor glyph. As with single-cursor window systems, event-driven drawing operations are performed normally by the application glyph, based on these events, with the cursor glyph sketching the cursors on top of the normal graphics.

Figure 2. Adding a multiple cursor overlay.

To incorporate the multiple cursor overlay into an application, the programmer instantiates a CursorOverlay which surrounds the application's graphic. Code in the overlay's constructor allows the CursorOverlay to communicate with remote conference objects and inquire about new conference users.

This technique, also used to implement the freehand sketching overlay, seems very promising. Through the composition mechanism, adding overlay components to applications is extremely straightforward. As well, the overlays are kept separate — conceptually and in the code — from the underlying applications. Its strength is that the overlay does not interfere with the underlying graphics of the application, even if those graphics are changing. Because of this, it is a trivial matter to add, for example, annotation capabilities on top of a "live" shared terminal application. Unlike other systems that only allow annotation of screen snapshots (eg MMCONF [5]), the underlying application can be fully active. It is expected that further research will suggest other components that could be transparently added to a variety of conferences.

Open Protocols

One design requirement for GROUPKIT is to provide flexible policies where appropriate, allowing group processes to be structured during a meeting, and to accommodate group differences. *Open protocols* provide a means of implementing this flexibility. They have three components: a controlled object, a controller object, and a protocol describing how the two communicate. The controlled object's behavior does not incorporate any policy determining how its state can be manipulated. Instead, a protocol is defined, and the object will obey any external requests made to it to change its state. The controller is just an external object that implements a particular policy by the requests it sends to the controlled object.

Currently, two components have been implemented using open protocols, a floor control module and a registration module. We concentrate here on the registration module. Figure 3 shows the message protocol (beside the arrows) between the Registrar (the controlled object) and its Clients

(the controllers). Here, the Registrar responds to *any* request from its clients, allowing any client to ask the Registrar to create a new conference, or conceivably even to delete any user out of any conference. While this does make it possible to create a "super-user" version of the Client, it also provides the flexibility to create any number of other Clients interfacing to the Registrar, without making any changes to the Registrar itself. As examples of this, the implementations of both a *free* and a *restricted* registration policy are now described. Under a free registration policy, new users may join any existing conference. The implementation here is straightforward. The Registrar Client sends an `add-user` message to the Registrar, which is broadcast to the other Registrar Clients in the selected conference. The Registrar Client also requests the Coordinator to create a new application Conference. The Conference makes connections with the other users, and the interaction proceeds normally.

Figure 3. Registration Protocol.

In contrast, a restricted registration policy does not permit new users to join an existing conference unless "sponsored" by an existing conference participant. Here, the Registrar Client again sends an `add-user` message, which is rebroadcast to the other users. At this point, the local Registrar Client does *not* ask the Coordinator to create a new conference. The remote users are asked by their Registrar Clients if the new user should be accepted. A remote user can accept the new user, and sends them a message, prompting the new user to create the conference as before. If rejected (either explicitly or by timeout), the `delete-user` message is sent to the Registrar.

Floor control policies are handled similarly [14]. Each user has a flag, specifying if their actions should affect the conference or not. Any user can examine or change any other user's flag. A preemptive floor control scheme for serial interaction, where a user can immediately seize control from the current floor holder, can be implemented by setting the local flag to `write` and setting all others to `no-write`. The end result is that the local user gains control of the floor until preempted by another user. A ring-passing policy, where the floor may be seized only if it is free (ie the previous floor holder has released it), is implemented by setting the local flag to `write` only if all others are currently set to `no-write`. MMCONF [5] uses a similar strategy for floor control, but manipulates only a single token, thus allowing less flexibility in protocols.

Again, this strategy seems useful in general. By providing a simple protocol to change states, building new policies becomes a simple matter of expressing the policy's semantics in the language of the protocol.

BUILDING AN EXAMPLE APPLICATION

This section describes the steps necessary to build applications using GROUPKIT. The steps include specifying the application-specific graphical presentation and interaction, initializing objects to send and receive application messages, selecting an application Conference object, and initializing a Coordinator object.

As an illustration, we describe the construction of a simple multi-user freehand sketching program using the multiple cursor overlay. The interface is similar to GROUPSKETCH [13]: multiple cursors are always visible, any user can draw at any time, and fine-grained actions are immediately visible on all displays. Note that although GROUPKIT provides a sketching overlay, for illustration we will not use it.

Graphical presentation

The application needs one or more INTERVIEWS glyphs to manage the graphical presentation and user interaction aspects of the interface, as well as any internal data structures. For a freehand sketching application, this involves creating a glyph holding a bitmap, and providing tools (pencil and eraser) that respond to mouse events for changing the bitmap. To incorporate the cursor overlay, the bitmap glyph is "composed within" the cursor glyph.

Designing this part of the sketching application is comparable to designing a single user version of the group application. There are some general conventions that are helpful to follow, such as separating event handling (cause) and the result of events (effect) into different routines. This facilitates use of common routines for local and remote invocations [15].

Messaging objects

Writer and Reader objects are used to send and receive application-specific messages. The routines in Writer objects are invoked as a result of local actions, for example, transmitting coordinates of a drawn line segment to the other replicated applications. Callbacks in the Reader objects interpret these messages, usually calling routines in the graphical presentation object to handle requests. In the example, the Reader instructs the sketchpad glyph to draw the line specified in the message. The standard objects must be initialized to include the required callbacks.

Coordinator

The Coordinator connects the registration mechanism (via the Registrar Client) to the application Conference objects running as separate processes. Available conference types

must be specified to the Coordinator, using the standard X resource mechanisms (ie .XDefaults).

Application Conference

The application Conference maintains communications channels with other distributed applications. In our example, the generic GroupKit conference has sufficient functionality. The main program instantiates this object, and "attaches" to it the bitmap glyph described earlier, so that the glyph can send and receive messages.

The Conference is notified when users join or leave. Routines in the base Conference class manage the low-level socket connections between users. However, other classes may be notified when new users join or leave, to manipulate application data structures maintained for each conference user. For example, the cursor overlay uses this information to add or remove cursors as users join or leave.

FUTURE WORK

The work presented here should be seen as an initial attempt to formalize the design and implementation of general groupware conferencing toolkits. The design requirements emphasize the important abstractions needed in real-time CSCW applications, and provide a basis for generalizing existing application or toolkit features. The three strategies presented — a run-time process and communication architecture, overlays and flexible policies — should be seen as general strategies that can be used to implement certain design features. It is expected that further design principles and strategies will evolve.

GROUPKIT has proven to be a flexible platform for testing our ideas. We have already built prototype drawing programs and shared terminals, and we will be constructing more elaborate and robust applications shortly. Currently, several of the design principles have not yet been embodied in the toolkit. Our immediate plans are to address the concurrency control issues, building a layer of support for generic shared graphical objects, that follow ideas presented in [15]. Ideally, a framework for building domain-specific group graphical editors could be created, drawn from the ideas in Unidraw [29].

Some other work has focussed on blurring the distinction between synchronous and asynchronous groupware, by providing a system that determines the appropriate means of conferencing (synchronous or asynchronous) and makes available various communication channels (text, voice, video). The choices offered the user depend on environmental information (ie who is currently around, what communication channels are available). The registration mechanisms in GROUPKIT provide the flexibility to implement such a scheme.

Groupware toolkits still have a long way to go to catch up to their single-user counterparts. We look forward to the day when all toolkits will incorporate multi-user features, and its interface components (such as control panels and editing widgets) have multi-user capabilities built into them. When that day comes, the artificial distinction between single and multi-user systems will disappear.

Note: GROUPKIT will be available for anonymous ftp by conference time, from the Department of Computer Science, University of Calgary (`cpsc.ucalgary.ca`).

REFERENCES

1. Bier, E.A. and Freeman, S. MMM: A User Interface Architecture for Shared Editors on a Single Screen. In *User Interface Software and Technology (UIST '91),* Nov. 11–13 1991, pp. 79-86.

2. Bly, S.A. and Minneman, S.L. Commune: A shared drawing surface. In *Proceedings of the Conference on Office Information Systems (COIS '90),* Apr. 25–27 1990, pp. 184-192.

3. Bonfiglio, A., Malatesa, G., and Tisato, F. Conference Toolkit: A framework for real-time conferencing. In *Proceedings of the 1st European Conference on Computer Supported Cooperative Work (EC-CSCW '89),* Sept. 1989.

4. Chapanis, A. Interactive human communication. *Scientific American 232*, 3 (1975), 36-42.

5. Crowley, T., Baker, E., Forsdick, H., Milazzo, P., and Tomlinson, R. MMConf: An infrastructure for building shared applications. In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '90),* Oct. 7–10 1990.

6. Dewan, P. Flexible user interface coupling in collaborative systems. In *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems,* Apr. 27–May 2, 1991, pp. 41-48.

7. Dykstra, E.A. and Carasik, R.P. Structure and support in cooperative environments: The Amsterdam Conversation Environment. *IJMMS 34*, 3 (Mar. 1991), pp. 419–434.

8. Ellis, C.A., Gibbs, S.J., and Rein, G.L. Groupware: Some issues and experiences. *Comm. ACM 34*, 1 (1991).

9. Francik, E., Rudman, S.E., Cooper, D., and Levine, S. Putting innovation to work: Adoption strategies for multimedia communication systems. *Comm. ACM 34*, 12 (Dec. 1991), pp. 37–63.

10. Garfinkel, D., Gust, P., Lemon, M., and Lowder, S., "The SharedX multi-user interface user's guide, version 2.0," , HP Research report, no. STL-TM-89-07, Palo Alto, California, 1989.

11. Gibbs, S.J. LIZA: An Extensible Groupware Toolkit. In *Proceedings of the ACM CHI'89 Conference on Human Factors in Computing Systems,* Apr. 30–May 4 1989, pp. 29-35.

12. Greenberg, S. Sharing views and interactions with single-user applications. In *Proceedings of the Conference on Office Information Systems (COIS '90),* Apr. 25–27 1990, pp. 227–237.

13. Greenberg, S. and Bohnet, R. GroupSketch: A multi-user sketchpad for geographically-distributed small groups. In *Proc. Graphics Interface (1991).*

14. Greenberg, S. Personalizable groupware: Accomodating individual roles and group differences. In *Proceedings of the 2nd European Conference on Computer Supported Cooperative Work (EC-CSCW '91),* Sept. 1991.

15. Greenberg, S., Roseman, M., Webster, D., and Bohnet, R. Human and technical factors of distributed group drawing tools. In *Interacting with Computers (Special Edition on CSCW, in press).*

16. Johnson-Lenz, P. and Johnson-Lenz, T. Post-mechanistic groupware primitives: rhythms, boundaries and containers. *IJMMS 34*, 3 (Mar. 1991), pp. 385–418.

17. Kraut, R.E., Egido, C., and Galegher, J. Patterns of contact and communication in scientific research collaborations. In *Intellectual Teamwork: Social Foundations of Cooperative Work.* Lawrence Erlbaum Associates, pp. 149–172, 1990.

18. Lakin, F. Visual languages for cooperation: A performing medium approach to systems for cooperative work. In *Intellectual Teamwork: Social Foundations of Cooperative Work.* Lawrence Erlbaum Associates, pp. 453–488, 1990.

19. Lauwers, J.C. *Collaboration transparency in desktop teleconferencing environments*, Ph.D. dissertation, Technical Report CSL-TR-990-435, Stanford University, Computer Systems Lab, CA, 1990.

20. Lauwers, J.C., Joseph, T.A., Lantz, K.A., and Romanow, A.L. Replicated architectures for shared window systems: A critique. In *Proceedings of the Conference on Office Information Systems (COIS '90),* Apr. 25–27 1990, pp. 249-260.

21. Linton, M.A., Calder, P.R., Interrante, J.A., Tang, S., and Vlissides, J.M., *InterViews Reference Manual*, Stanford University, Sept. 1991.

22. Minneman, S.L. and Bly, S.A. Managing a trois: A study of a multi-user drawing tool in distributed design work. In *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems,* Apr. 27–May 2 1991, pp. 217-224.

23. Neuwirth, C.M., Kaufer, D.S., Chandhok, R., and Morris, J.H. Issues in the Design of Computer Support for Co-Authoring and Commenting. In *Proc. of the Conference on Computer-Supported Cooperative Work (CSCW '90),* Oct. 7–10 1990.

24. Nunamaker, J.F., Dennis, A.R., Valacich, J.S., Vogel, D.R., and George, J.F. Electronic meeting systems to support group work. *Communications of the ACM 34*, 7 (July 1991), pp. 40-61.

25. Patterson, J.F., Hill, R.D., Rohall, S.L., and Meeks, W.S. Rendezvous: An architecture for synchronous multi-user applications. In *Proce. of the Conference on Computer-Supported Cooperative Work (CSCW '90),* Oct. 7–10 1990.

26. Patterson, J.F. Comparing the programming demands of single-user and multi-user applications. In *Proc. User Interface Software and Technology (UIST '91),* Nov. 11–13 1991, pp. 87-91.

27. Stefik, M., Bobrow, D.G., Foster, G., Lanning, S., and Tatar, D. WYSIWIS revised: Early experiences with multiuser interfaces. *ACM Trans. on Office Information Systems 5*, 2 (1987), pp. 147–167.

28. Tang, J.C. Findings from observational studies of collaborative work. *IJMMS 34*, 2 (1991), pp. 143–160.

29. Vlissides, J.M. and Linton, M.A. Unidraw: A Framework for Building Domain-Specific Graphical Editors. In *Proc. of User Interface Software and Technology,* Oct. 1989.