Experiences with object-oriented group support software development

by S. C. Hayne
M. Pendergast

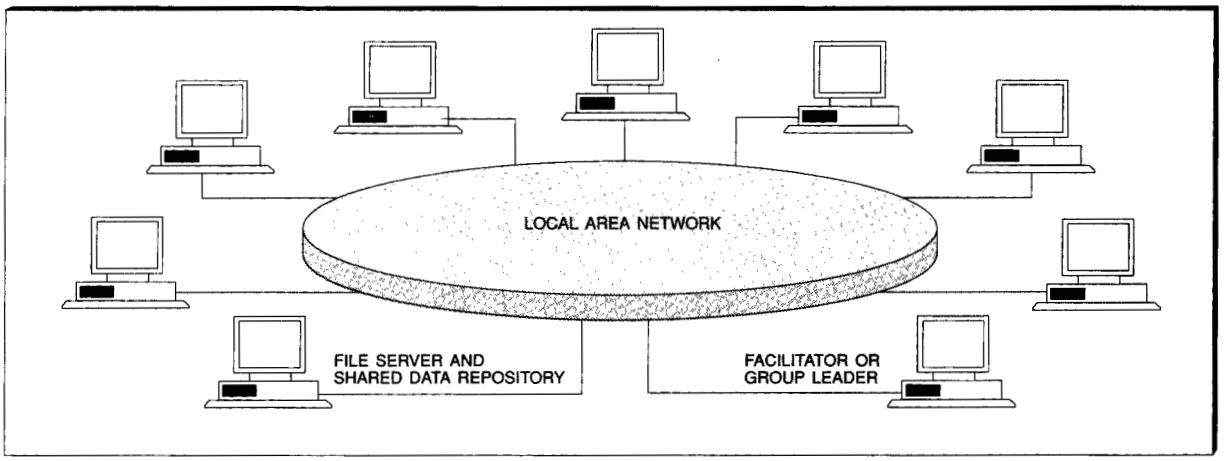
This paper describes practical design and implementation experiences gained when creating Group Support Systems (GSS) in a networked personal computer environment. Examples of GSS based on the shared context model and implemented using C, C++, and Actor languages are presented. Graphical user interfaces and multitasking extend traditional methods for supporting group work. An object-oriented communication system is introduced comprised of objects that provide support for all inter- and intraprocessor communications between the GSS applications. Multiple levels of data service are provided to maintain shared data, coordinate user views, and transmit cursor positions in a convenient and efficient manner. The applications presented not only demonstrate the viability of implementing GSS on personal computer-based systems, but also show the ability to develop complex applications in different programming environments that make use of common routines. The unique properties of the object-oriented paradigm greatly facilitate the creation and use of Group Support Systems.

Over the past ten years academic and industry researchers have developed computer systems that increase the productivity of work groups. This research has progressed somewhat independently along two parallel tracks: Group Decision Support Systems (GDSS) and computer-supported cooperative work (CSCW). DeSanctis and Gallupe defined a Group Decision Support System in 1987 as "an interactive computer-based system that facilitates the solution of unstructured problems by a set of decision-makers working together as a group."[1] Tasks commonly supported by GDSS include brainstorming, idea organization, voting, strategic planning, policy formation, total quality management, and communication. These systems are typically implemented for personal computers running the disk operating system (DOS) on low-cost machines that have allowed corporations to adopt and experiment with this new method of group work. Several research prototypes have been developed to prove feasibility; the most notable of these has evolved into the commercial product GroupSystems V**. Computer-supported cooperative work was defined by Ellis, Gibbs, and Rein in 1991 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."[2] CSCW applications include systems design, collaborative writing, project management, and process control. The research prototypes available have been implemented in UNIX** platforms[2–7] and, although these systems support different tasks, they are very similar from an architectural standpoint.

For example, consider five individuals working on a proposal. Each wishes to work on a section of the document at the same time. The group requires real-time access to the shared data, com-

0018-8670/95/$3.00 © 1995 IBM

Figure 1 A common Group Support System environment



puter applications that provide structure to group work, and advanced user interface concepts. Hardware requirements include a processor fast enough so that all inputs (from everyone in the group) can be processed in real time, a reliable and fast network so that all participants feel that they are working together, and enough storage for both individual and group work. Rather than be rigorous about which label should be used, we use the more generic term Group Support Systems (GSS), defined by Jessup and Valacich in 1993,[8] throughout this paper.

The keys to making groups more productive is to allow a high degree of parallel activity and unrestricted access to shared data. If concurrent access to data is controlled via locks, then locking must be performed with a fine level of granularity such that one person's work activity will not be restricted by another's.

The most common platform for the development of GSS is a set of workstations that communicate over a local area network and employ a file server as a shared data repository (see Figure 1). Until recently, GSS applications were confined to running under a single tasking operating system (like DOS) and employed a mixture of text and primitive graphical user interfaces. This environment seriously limited the complexity of the tasks and the degree of interaction that could be supported. For example, while the group is brainstorming a solution by using the computer applications, an individual cannot access any external information in order to support ideas. Nor can individuals work on something else if they are finished contributing.

With the maturity of multitasking operating systems that support the IBM Common User Access* (CUA*) protocols for graphical user interfaces, limitations such as those just illustrated have been removed. IBM's Operating System/2* (OS/2*) with Presentation Manager* and Microsoft Corporation's Windows** (hereafter called Windows) are examples of such environments[9,10] without the limitations. With these advanced environments and the proper software, users should be able to migrate (usually with few changes) from single-user software to multiuser software. In the multiuser environment, users would develop ideas or designs in a private work space, then move the data, or paste the work space into the shared area.

Most programmers use C or Pascal run-time libraries of applications to perform standard input, output, memory management, and other activities. These applications assume a standard operating environment using character-based terminals for user input and output, and exclusive access to system memory and the input/output devices of the personal computer. Under OS/2 and Windows, these assumptions are no longer valid because all applications share the resources of the computer. These operating environments have many features that extend the capabilities of the basic DOS environment and are mandatory for development of functionally advanced GSS.

These features are:

- A graphical user interface—Applications share the display by using a window for interaction with users. This window is a combination of useful visual devices, i.e., menus, controls, and scroll bars, that the user manipulates to direct the actions of the application. Those GSS that follow user access standards help to ensure that users can learn applications quickly and that the applications behave in a predictable manner. Currently, all GSS behave differently.
- Device-independent graphics—Device-independent graphics are supported by inserting a *device context* between the specific graphic operation performed and the specific device. This device context is comprised of the device driver, the output device, and the output port. Various size screen displays and printers are thus supported with the same drawing functions. GSS can easily be used on a variety of available platforms.
- Multitasking—Multitasking is performed and applications are protected from the difficulties of memory management. Most virtual memory systems page unused code and data segments to disk using a combination of the least recently used and demand paging algorithms. GSS can coexist with a melange of other support tools to further enhance interaction.
- Threaded message queues for device input—Input from supported devices (i.e., keyboard, mouse) is provided automatically to every window created in a uniform format called an *input message*. For example, every time a key is pressed, two messages are sent (e.g., for Windows, WM_KEYDOWN and WM_KEYUP). System level message dispatchers collect all hardware device and application messages, queue them, and redirect them to the destination application. These types of messages allow easy coordination of group applications because the message can be sent from another machine on the network. Using the above message queue, applications can send messages to each other to share information dynamically or to trigger actions.

These features have some useful strengths. Device-independent graphical support makes it very easy to develop graphical applications that will operate on a multitude of delivery platforms. Virtual memory frees the developer from memory management and allows initial development of memory-inefficient prototypes. The messaging capability of OS/2 and Windows is a unique strength for GSS software. Most GSS applications need to send and receive messages from other applications that are taking part in the group session. OS/2 and Windows allow this to be done in a trivial way at a single workstation. One only needs to provide the appropriate network support to allow those same messages to travel to other stations. This communications layer has been implemented and will be described later.

In this paper we first discuss the data sharing models and peer-to-peer communications that are fundamental to supporting group work. Next we report on the lessons learned from developing group software using the object-oriented paradigm in both traditional (C) and nontraditional (C++, Actor) environments, respectively. Finally, we outline the conclusions and directions for future research.

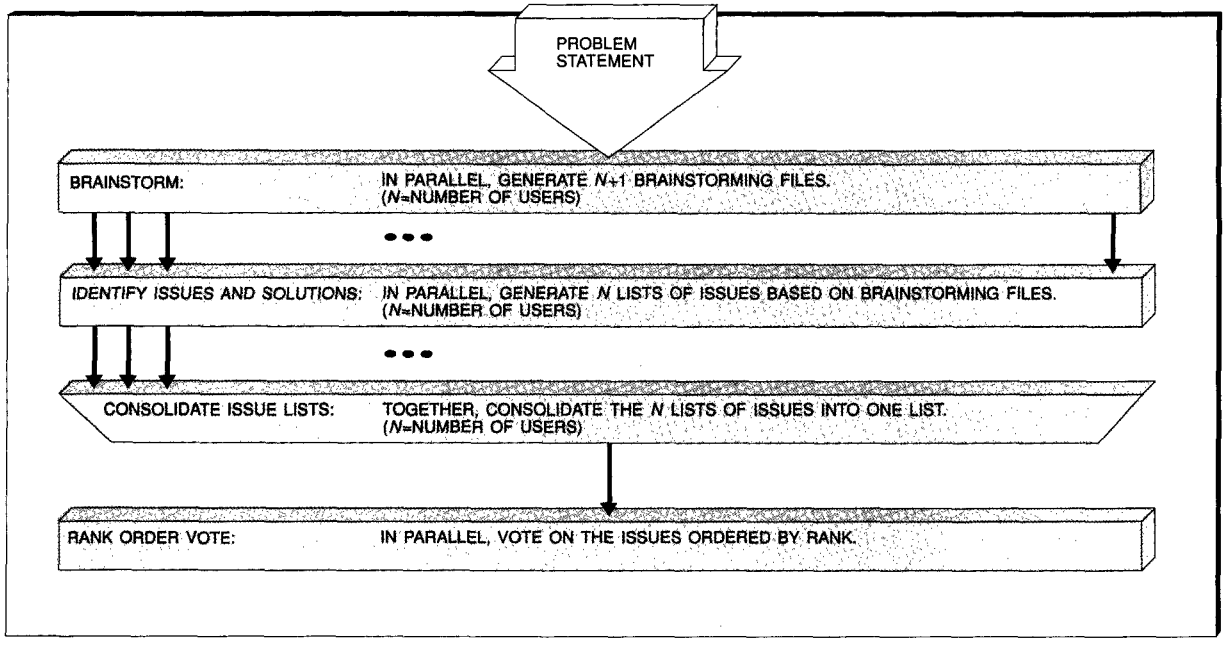
Alternative GSS process models

In order to understand the problems presented to GSS developers, it is first necessary to describe some basic interactions between users, and between users and programs. We briefly look at two existing group work paradigms: group decision-making and coauthoring.

The group decision-making model. The *group-decision making model* employs an automated form of the three-phase model of intelligence, design, and choice.[11] Systems that follow this model are predicated on assumptions that the best decisions are reached only if an adequate amount of time is spent in the discussion and generation of alternatives, and that the more alternatives generated and evaluated the better the outcome will be. The principal drawback to this technique is the difficult task of consolidating similar alternatives and reaching a group consensus.

Figure 2 represents the data flow and propagation of the group decision-making process model common in commercial Group Decision Support Systems. Under this model a session begins with a problem or focus item. The problem is replicated in $N+1$ discussion files (where N is the number of users). Users may comment upon the problem and others' responses during brainstorming. Users make one entry into a file and are then randomly switched to another file. This allows

Figure 2 Traditional Group Decision Support Systems data flow

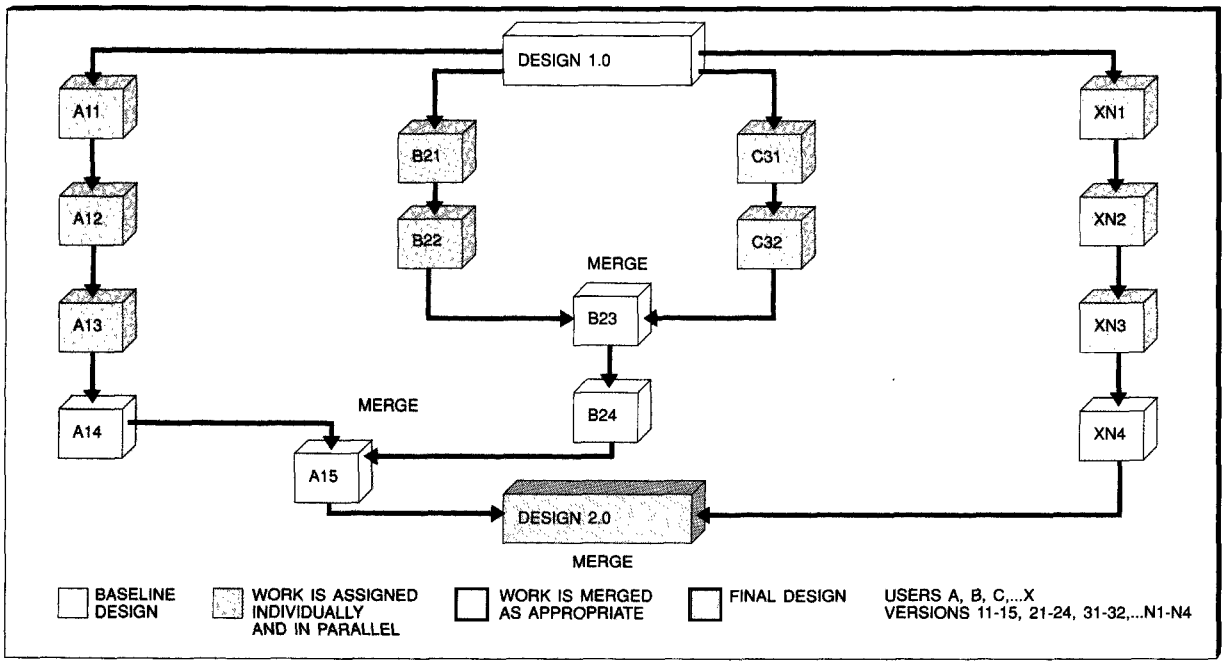


$N+1$ different discussion streams to develop and encourages all viewpoints to be considered. Following brainstorming, users work individually to generate lists of issues or solutions to the problem. The group then merges and consolidates the lists. The lists are distributed to each workstation and voted upon by each user. Several methods of voting exist such as sorting from most acceptable to least acceptable, or sorting by a predefined criteria using a Likert scale (1 being best and 7 being worst). When the users have finished voting, the ranked lists are collected, aggregated, and the results displayed. The lists are voted upon with this process repeating as necessary. In general, we found that participants enjoyed the brainstorming and voting phases, but found the consolidation phase to be tedious. The longer participants were allowed to work on parallel tasks (increasing productivity and satisfaction), the greater the amount of time needed in the integration tasks.

This approach has been shown to be successful,[12] but can be improved by addressing several problems that are inherent in the data sharing protocols. The brainstorming of $N+1$ files is a technique derived from a manual (paper) method.[13] While it does promote many contributions and can provide a high degree of parallel activity, it does not allow a user to address fully a question in a particular file or respond to all comments in a file. In very large groups ($N>16$) it is possible that a user might only obtain a given file once or not at all. Having a separate issue and solution generation phase is good for focusing the user's attention on that task, but it does not provide the capability for users to discuss proposals electronically and revise proposals before a vote is taken. This technique generates copious alternative solutions, but leads to the creation of many duplicate issues, which in turn requires a separate and lengthy serial consolidation phase. Also, iterating the process is difficult due to the number of phases and overhead of starting and stopping a tool.

The coauthoring model. The *coauthoring model*, or the merging or integrating of work, has also been a very difficult problem in the areas of multiperson authoring and conceptual modeling and design.[14,15] It involves providing the group with access to a single document or design that enables a single group view to evolve over time from inputs of the group members.[2] The coauthoring paradigm allows access to the document, but en-

Figure 3 Group coauthoring process model



forces very little structure on the group. However, each author's inputs must continually be merged with those of the other authors. Posner et al.[14] studied writing strategies and found four predominant ones:

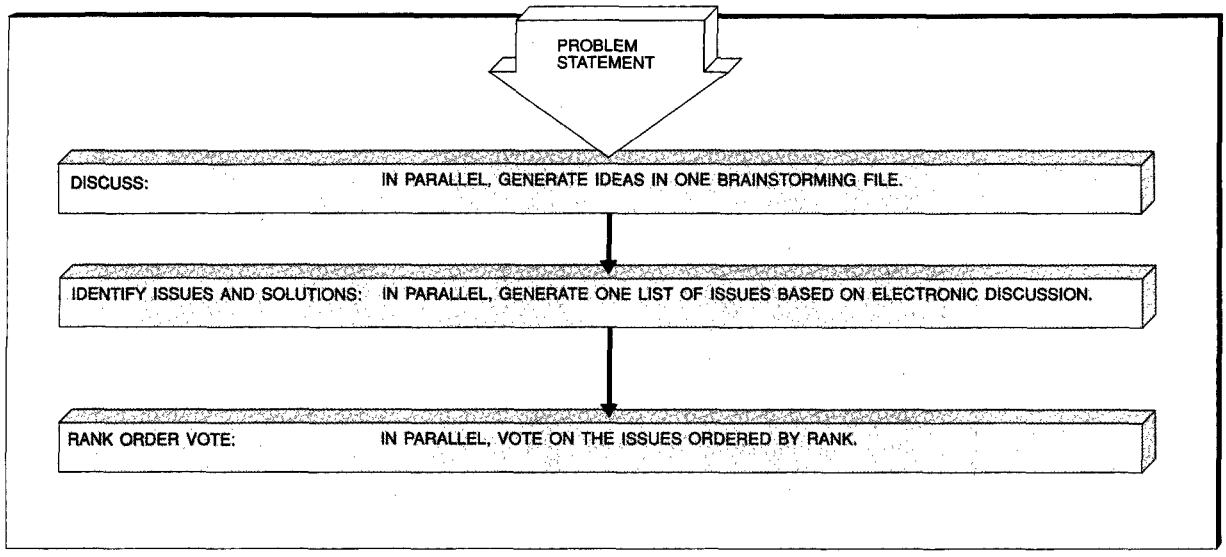
- *Single*, where one person writes the document based on inputs from the group
- *Scribe*, where one person transcribes the thoughts of the group during a meeting
- *Separate*, where the document is separated into sections with each member of the group responsible for a part
- *Joint*, where the entire group writes the document together, jointly deciding the phrasing of every sentence

The results of the study indicated that the use of the separate strategy, with independent document control, dominated all forms of collaborative writing. An implication is that current technology, in the form of single-user word processors, can dictate the group's writing strategy. The increased availability of local and wide area networks facilitates the relaying of documents between authors, but the data access model employed by single-user tools restricts the choices of writing strategies.

Figure 3 represents the work flow for users employing the separate strategy. It begins with a baseline revision level or starting point. Each user is assigned a portion of the work and individually works through the details of that piece. As segments of the work are completed, they are merged with the portions being completed by other users or with the original baseline. Thus for any single revision level there are a minimum of $N-1$ (N=number of users) integration operations. Each operation could require moderate to extensive changes to the segments being merged, depending on the level of communications supported during the independent work phase. These changes can be necessitated by inconsistent and varying work methodologies (or writing styles), duplication of work, omission of components, and incompatibility.

The Shared Context Model. The problems of the group decision-making and coauthoring models addressed in the previous section led to the adop-

Figure 4 Shared Context Model for group decision support systems applications



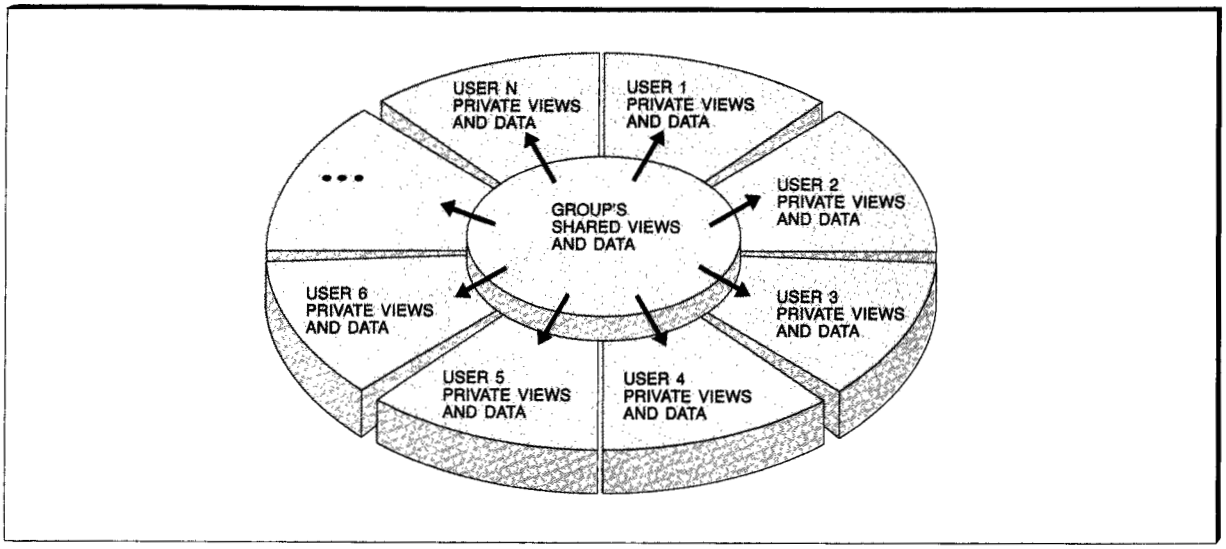
tion of an alternative approach: the Shared Context Model.[16] Shared contexts are by no means a new concept. Variants of the Shared Context Model have been embodied in many, if not most, computer-supported cooperative work applications. Establishing and maintaining a joint understanding of the information is what differentiates GSS applications that support the Shared Context Model from those that provide "what you see is what I see" (WYSIWIS) views of information. For our purposes, *shared contexts* are defined as a common work environment consisting of information, representations of the information (text or graphics), and most importantly, a joint understanding of the information.

Figure 4 illustrates the Shared Context Model for GDSS applications. Like Figure 2, the model starts with a problem or group focus item. Users are free to make comments on the problem and raise subissues or related focus items. When the discussion has reached a mature state, users will begin to propose solutions. The discussion proceeds with users commenting on each other's solutions and submitting alternate proposals. When a vote is taken the solution list is condensed and the process continues.

The Shared Context Model improves on the traditional methods using several techniques. Data redundancy, which occurs in classical brainstorming implementations, is reduced by giving users concurrent access to all information in the decision process. Users are allowed to categorize inputs as general comments, issues, or solutions, giving more direction and focus to electronic conversations. Finally, the ability to cycle quickly through the discussion, alternative analysis, and voting phases provides a better environment for creating compromise solutions and reaching a consensus. The important differences are that users are given more freedom to progress from one stage of the decision process to the next, have greater access to data, and are free to respond to any issue or proposal at any time.

The Shared Context Model developed for GDSS applications can be extended to encompass coauthoring applications as well (see Figure 5). In the coauthoring perspective each user has access to shared data and may manipulate the data using an individual view. Work is coordinated via shared views of the data. Private data, i.e., data used in the work but derived from external sources, are also available. The key point is that all users have concurrent read and write access to all the shared data. They work in parallel, using individual views of the data, and may coordinate their work using shared views. Individual views may overlap; when this is the case, a change

Figure 5 Shared Context Model with a coauthoring perspective



causes all views to be updated in real time. The only merging or integration steps necessary are those that are required when adding data or work from sources external to the shared context. Because shared data are maintained in one context, standards can be enforced consistently. Compatibility of work segments is enhanced, since they are created and joined concurrently. Having a single context also prevents work duplication and decreases the possibility of omissions. The shared views provide a common focus and augment group communications. Several examples of multiuser text editors exist that employ shared workspaces.[2,17,18]

Maintaining a joint understanding of the information in coauthoring applications is much more difficult than in GDSS applications. In GDSS applications, the bulk of the shared information is in the form of ideas and comments generated by participants as they work toward the solution to the problem. This allows each participant to form a mental picture of the group's understanding of the problem. On the other hand, the shared information in coauthoring systems is restricted to the work product (i.e., the diagram or document). The mental processes that participants go through during the creation of the document are generally not captured. Instead, this communication is restricted to verbal interaction. This verbal communication is suspended when participants are working in parallel on different segments of the document[19] and when participants work on the document at different times.

Data sharing techniques

Group interfaces differ from single-user interfaces in that they depict group activity and are controlled by multiple users rather than a single user. This introduces problems not associated with single-user interfaces, i.e., concurrency control, view management, and work space management. Multiple users can produce a higher level of activity with a greater degree of concurrency in a shorter amount of time compared to a single user. Therefore, multiuser interfaces must support this behavior.

We have chosen to manage this complexity through the implementation of a *relaxed* WYSIWIS.[6] WYSIWIS alone implies that the shared context is guaranteed to appear the same to all participants.[2] WYSIWIS can be relaxed, or slightly different, along four dimensions: display space, time of display, subgroup population, and congruence of view. The systems described herein are in the relaxed subgroup population. The coordinator of the meeting can create subgroups and change their access security dynamically. The

systems are also relaxed along congruence of view in that individual users can change their perspective on the shared context, e.g., zoom in on parts of the context. This has been done to reduce distraction. If individuals are allowed to maintain their own view of group data, these data have to be shared in some rationally robust manner. Thus, we have implemented the Shared Context Model using peer-to-peer communications, shared files, and a hybrid file- and message-based system.

Program-to-program communications. This section describes an object-based communications system that was specifically designed to support group work through the use of channel objects. Channel objects are different from network objects in that they are created dynamically, depending on the requirements of the group application. Our channel objects are defined as dynamic multicast connections that allow stations to enter and exit a session at will, i.e., multicast nonpermanent sessions. The ability to maintain shared data and views requires capabilities not present in most data communications systems, such as the ability to perform secure broadcast communications and the ability to maintain dynamic multicast connections.[3] Implementing distributed applications with the characteristics and requirements previously specified with standard peer-to-peer connections is possible, but would present several major complications. The communication system described in the following paragraphs is designed to relieve the task of low-level, data communication session management from the applications and make efficient use of transport layer resources. This is accomplished via a set of network and channel objects that act as an interface between applications and the network.

The communication system is comprised of three specialized communication objects (see Figure 6): a network base object (NBO), a network interface object (NIO), and a name server object (NSO) for object names.[20] They provide support for all inter- and intraprocessor communications between the GSS applications presented later. The base object (NBO) provides transport services to the interface object (NIO), which in turn queries the name server (NSO). These three objects work together in order to create and maintain GSS channel objects. The object-oriented nature of their design has the benefit of separating (hiding) the transport level communications calls from applications, allowing these protocols to be modified without affecting the applications. For example, the NIO object was recently updated to support Novell, Inc.'s IPX** transport protocol in addition to IBM's NetBIOS (TCP/IP will be added in the near future). If the NIO did not exist, then all applications making network calls would have had to be modified to support IPX. This design has been implemented in both a C++ object-oriented programming environment and in a Microsoft Windows environment using standard C.

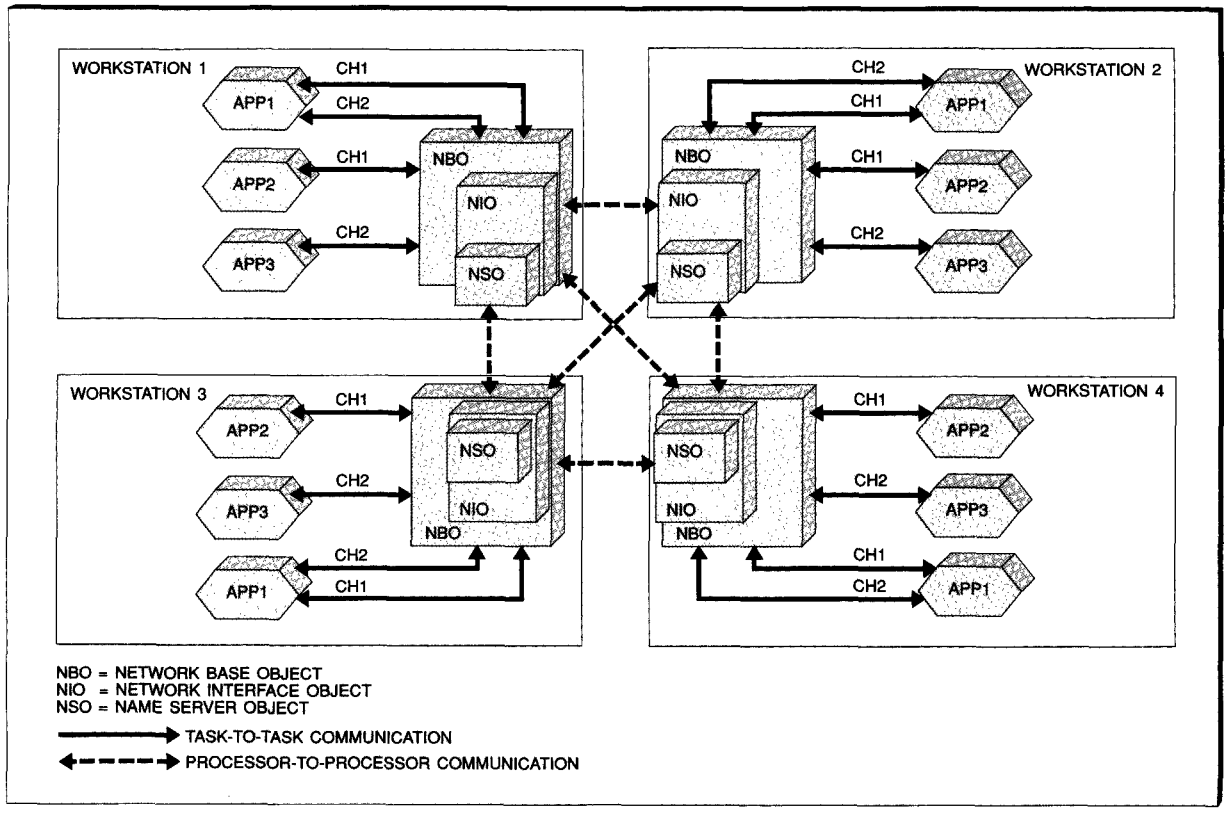
Network base object. This is an abstract object designed to be inherited by all GSS applications and provides the interface between GSS applications and the network interface object. The NBO provides functions for connecting with remote objects, transmitting messages to objects, registering and deregistering objects, and requesting object connection lists (lists where the object is active). In a single program environment the NBO takes the form of a C++ abstract object. In multiprogramming environments such as Microsoft Windows, the NBO takes the form of a dynamic link library[10] (DLL).

In order for an application to transmit data on a GSS channel, the application must register with the NBO and request the data to be transmitted. The system performs the actual session calls, network addresses, and transmits and receives.

Depending upon the implementation, the processing of incoming data is performed in one of two ways. In the C++ implementation, the NBO directly initiates an applications message processing method when the message arrives, whereas in the Windows environment, a Windows event message is generated for each incoming message.

Network interface object. Communications between different workstations are handled by the NIO (see Figure 7). As seen from Figure 6, the base object (NBO) provides a wrapper around the NIO. The NIO resides on each workstation and is responsible for controlling the interface between GSS channel objects and the transport layer drivers served by the base object (NBO), e.g., NetBIOS, IPX, TCP/IP. This includes maintaining service access points for each channel that needs to perform network communications, grouping of outgoing messages, and reassembly and queuing of incoming messages. Transport services are multiplexed by

Figure 6 ***Network objects in a typical communication system***



the base object (NBO), therefore, the NIO maintains logical session numbers (LSN in Figure 7).

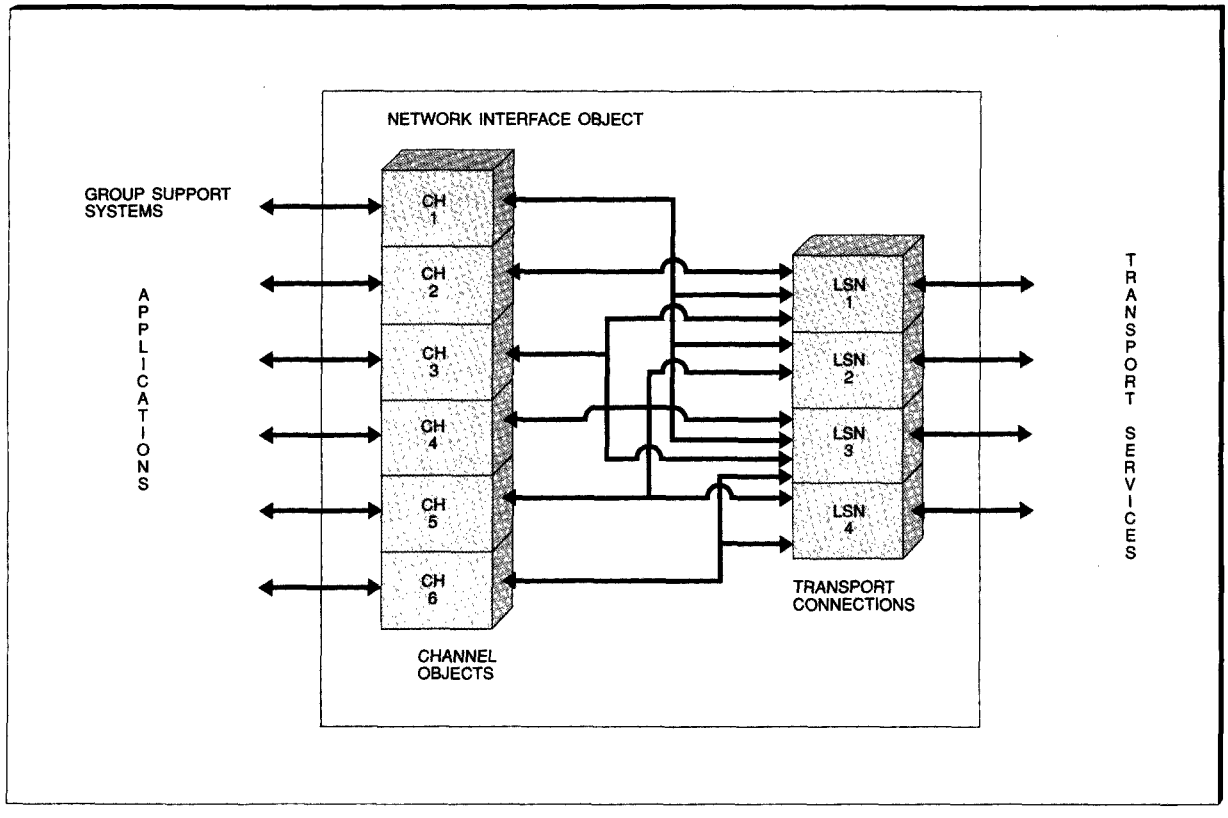
The NIO uses two types of data transmissions: broadcast datagrams and reliable session connections. *Broadcast datagrams* are used for messages where speed is more important than reliability. For example, telepointer positions (X,Y) are transmitted more than 20 times a second in order to generate smooth movement of a pointer across a screen, but the loss of a message creates no dire consequences. *Reliable session connections* are used to transmit data that must not be lost, e.g., voting results, text document updates. The application can select the mode of transmission (secure or unsecured) when transmitting a message.

When an application registers a channel object, the base object (NBO) informs the interface object (NIO) of its presence. Then the NIO registers the application with the name server object (NSO), described below, and enters the application's address in a local address table. Transport level connections are then established with the NIO on each workstation in the broadcast list of the GSS channel, if one does not already exist.

When an application is ready to send a message on a GSS channel, it invokes a base object (NBO) transmit function. The base object (NBO) transmit function formats the data and passes the data to the interface object (NIO) for transmission. The NIO transmits the message to each workstation in the broadcast list (reliable) or broadcasts a datagram (unreliable). When a message is received from a remote workstation, the NIO on that workstation passes it along to all applications that have registered with the channel.

The NIO maintains only one transport level connection with each workstation, regardless of the

Figure 7 Network interface object (NIO) internals



number of channel objects on the workstations that are exchanging information. This is done by multiplexing many session level (channel) connections over a single transport level connection, thereby reducing the overhead of creating and maintaining multiple transport connections between stations and making better use of local area network (LAN) adapter resources.

The design of the NIO has been derived from a reliable broadcast communication system previously developed in a conventional C environment. This communication system was successfully used to develop electronic market programs for experimental economics research and for implementing multiuser GSS tools.[20–23] The design relied on a central workstation to sequence and broadcast messages to other user workstations. The central workstation scheme was dropped because of loading problems created by large groups, delays due to the extra transmissions required, and delays due to the complexity of the take-over protocol that was required when the central workstation failed. For some applications, such as text editors, sequencing of updates is mandatory for maintaining consistency on all workstations. For these tasks, a sequencing application could be developed that used a separate channel to receive messages and then rebroadcast them (a many-to-one in combination with a one-to-many channel).

Name server object. The NSO maintains multicast connections for network objects. To do this, each object must know which workstations on the local area network (LAN) have active shared objects. The NSO is designed to maintain a system-wide list of shared objects and the workstations that have active instances of these objects. These objects are formed on a user group basis and are implemented as shared files under a server directory accessible to the user group. Each channel

object is represented by a record in the file. Each object record stores the mode and type of the object as well as a list of active stations.

Whenever a new instance of a shared object is created on a user workstation, it is added to the list of active workstations by the interface object (NIO). That NIO then broadcasts this change to all NIOs in the active list. The user workstation NIO then uses the workstation list to establish connections with other workstations currently using the object. Once connections have been established, the current state of the object can be solicited from one of the other user workstations. When an object becomes inactive on a user workstation it is responsible for communicating this to facilitate deletion from the active list. The NIO will then relay this information to user workstation NIOs that remain active so they may update their object connection tables. Objects may also query the NSO to determine the station address of a singular object or to receive an updated list of station addresses for duplicated objects.

Shared file access. DOS currently provides low level protection for files through the use of file access and sharing modes. Access modes are Read, Write, and ReadWrite. To support the shared context paradigm, either files must be shared or a database management system must be used. We have chosen to use shared files because they have the advantage of easy implementation, fast execution, and a small memory overhead. The file-sharing modes implemented are DenyRead, DenyWrite, DenyNone, DenyReadWrite. Access and sharing modes are specified by the application when the file is opened. These modes can be used in combination in order to ensure consistency while providing dual access. For example, an application updating a shared file should open it with ReadWrite/DenyWrite, while an application simply reading the file would open it with Read/DenyNone.

A common practice for single-user Windows applications is to keep a disk file open only while processing the current Windows message. For multiuser applications, a Windows message that requires a file update should cause the file to be opened (ReadWrite/DenyWrite), updated, and then closed to allow other users access to the file. An error on opening the file would indicate that the file is currently being updated by another user; the application should delay a few tenths of a second, then try again, i.e., binary backoff. A Windows message that does not require changing the file, e.g., WM_PAINT, could open the file (Read/DenyNone) and read the data. This simple protocol provides concurrent data access to many user applications.

Hybrid shared data implementation. In order to maintain both shared data and shared views, it is necessary to combine the functions of shared files and program-to-program communications. A shared files dynamic link library (SFILES.DLL) was created to perform both functions for a GSS application. Input and output functions within SFILES look like standard C input/output functions and are:

```
sfopen(HWnd,File_Name,Sharing_Mode)
sfclose(HWnd,File_Handle)
sfread(HWnd,File_Handle,Offset,Addr,Count)
sfwrite(HWnd,File_Handle,Offset,Addr,Count)
sfappend(HWnd,File_Handle,Addr,Count)
```

Sfopen creates a special file handle and registers the file as a shared object with the interface object (NIO). Sfclose releases the handle and unregisters the file object. Each time the file is modified using sfwrite or sfappend, routines update the disk image and broadcast a message via the NIO to all active users. Each application then receives a special Windows message containing the type of update operation and the data written. Since this message is related to a particular object, applications can use this information to update their views of the data (screen or internal). SFILES also provides local buffering of file data in order to enhance performance of read operations. The same NIO messages that inform applications of view changes are automatically applied to the local buffer, ensuring that it always reflects the image stored on the server disk. SFILES adheres to the protocol for accessing shared files described in the previous section. The server file is kept open for write access only during actual updates. Reads and writes are performed on a record basis.

During the update sequence, the object is not locked, since some researchers suggest that small groups will develop a social protocol for concurrency control.[2,21] Other stations can change the object, but must wait their turn to write in the shared file. In our experience with groups of less than six people, a 16 megabyte (MB) token ring network and file server is fast enough so that up-

dates often occur before other users even attempt to change the same data. However, we believe that as group size increases and as groups are dispersed in space, a locking model will become more critical.

All a developer must do in order to create a WYSIWIS environment is to include the SFILES library and provide a routine that processes SFILES Windows messages. The file access coordination, local buffering of data, and network session management are handled automatically.

The reasons for choosing the OO paradigm are code reuse, messaging, and polymorphism.

C++ development environment

The three reasons for choosing the object oriented (OO) paradigm are ease of code reuse (illustrated below), messaging, and polymorphism. Messaging is extremely important for GSS because, as mentioned earlier, group applications must inform each other about their actions. The best way to inform is by sending each other messages. When using the OO approach, the application and the application developer are already heavily involved in exchanging messages. Therefore the transition to handling group messages is elementary. Polymorphism allows the same message to be sent to different objects, which then in turn can decide how to handle each one. This is very useful when GSS users are viewing the same basic data in different ways, i.e., a new object can be displayed differently for one user in a graphical window versus another in a text window. C++ has become the industry standard for commercial applications working with the OO paradigm.

This section describes our experiences developing Group Support Systems for Windows in C and C++. However, the techniques described here are equally applicable to the OS/2 Presentation Manager environment.[9] The C and C++ development environment for Windows consists primarily of a C compiler and the Microsoft Windows Software Development Kit. Compilers are generally available from companies such as Microsoft, Borland, Zortech, and IBM. In addition, there are many libraries and toolkits available that aid designers in constructing dialog boxes, icons, menus, and windows. Products such as Protogen**, WindowsMaker**, Visual C++** with AppStudio**/AppWizard**, and Microsoft QuickC** with CASE:W** allow designers to interactively construct the primary window constructs, menus, submenus, and dialog boxes, then produce a skeleton program in C that serves as a starting point for application coding. It is only a matter of time before full-scale computer-aided software engineering (CASE) tools will provide Windows support. The two most important criteria for selecting a compiler is whether or not it supports object-oriented programming and whether it provides a general user interface (e.g., Windows or Presentation Manager) interactive development environment.

Capabilities. The concept of classes, objects, and inheritance in C++ is similar to that in Lisp/SCOOPS, Smalltalk and other OO programming languages. C++ classes are an extension of the C language STRUCT (record structure definition). C++ classes allow the definition of data members (class variables) and member functions (methods). Hierarchies of classes are implemented in C++ through the creation of derived classes. Derived classes (or subclasses) are classes that inherit the member data and functions of its parent or base class and can have multiple base classes.

An interesting and useful capability of C++ is the virtual function. A virtual function is a method of a base class that can be redefined in a derived class, therefore the actual function that is referenced is determined at run time. Virtual functions are invoked through a pointer or a reference. This dynamic binding feature of C++ provides a common means for referencing methods in any class without having to compile in the derived class definitions. This is especially useful when developing libraries of system objects. Virtual functions can be called by system routines in order to allow asynchronous processing of error conditions and incoming messages. Another useful extension to the C language provided by C++ is function overloading. A function name is said to

be overloaded if two or more distinct C++ functions are defined with the same name. Calls to overloaded functions are resolved by the C++ compiler through a process of parameter and argument matching. This enables the designer to relax C-type checking and makes coding more flexible. Refer to Reference 24 for more information on object-oriented programming conventions.

There are two paradigms for interobject communication in the object-oriented programming arena: sending messages and executing methods. C++ is structured such that objects communicate with one another by directly executing their methods. Windows, on the other hand, has built-in user interface objects (buttons, edit boxes, list boxes, etc.) that must be communicated with via messages (i.e., SendMessage, PostMessage). In order to write programs in this environment, the programmer must be familiar with both paradigms.

For our purposes, the strengths of C and C++ lie in their ease of code reuse, performance, flexibility, and compatibility with other development environments. DLLs developed in C can be accessed by high-level Actor programs (see a later section) and are also capable of performing low-level DOS BIOS and NetBIOS functions. Memory requirements of programs are often one-quarter to one-half that of high-level environments, with the size of executable programs under 100KB. This allows for an extensive amount of multitasking without the performance penalties imposed by disk swapping.

The two most popular C++ Windows development environments are Borland's C++ and Microsoft Visual C++. Both provide rich class libraries (Object Windows Library and Foundation Classes), which serve to ease development of Windows programs and hide some of the details. For example, an Edit Box class is provided to create and manipulate an editing window. The constructor of this class creates the window, sizes it, and initializes it with text; the destructor destroys the window. Member functions provide the ability to get and put text, cut, copy, paste, select, etc. If this class did not exist, then the programmer would have to code the procedure calls to create a child window and send or receive messages to perform operations on it. This is an example of a *wraparound* class, a class that provides a C++ interface to a system level object. In addition to edit boxes and other user interface elements, wraparound classes exist for files, streams, and relational database tables (e.g., the Paradox Database Engine).

One major benefit to these classes is that they hide some of the details of manipulating system objects (e.g., the CreateWindow function in Windows has 11 different parameters, and there are 36 different message types exchanged between an edit box and its parent window). By making use of these classes the programmer is relieved of the burden of writing redundant message processing code and eliminates the need to debug and maintain this code. Thus code reuse comes in the form of using existing class library code. Class libraries generally provide standard interfaces. Often a programmer will need to deviate from the class standard to change the appearance of a user interface, add error checking, or extend functionality. The best way to do this is to create a class that inherits properties from the class library. Thus, code is reused from the class library and is already debugged, providing a double benefit.

The strengths of C and C++ are code reuse, performance, flexibility, and compatibility.

Shared graphic objects. Within TeamGraphics**, a Windows-based GSS tool by Ventura, graphical objects are shared between multiple users. The following class hierarchy exists:

```
TObject
    TGraphicObj
        TLine
        TSquare
        TCircle
        TPolyLine
        ...
        TTrapezoid
```

TObject is an object Windows class library object that provides the methods for working with collections of objects (arrays, sorted arrays, etc.). TGraphicObj is an abstract or virtual base class that inherits TObject and extends it to add virtual methods such as Draw, Erase, Store, Lock, Unlock. TLine, TSquare, and TCircle are some of the actual objects created by the user. They fill in virtual functions and override functions that are peculiar to their operation. For example, there exists a function to determine if a mouse click occurs within the bounds of an object. Most objects use the OverObject function defined by TGraphicObj. TPolyLines overrides this with a function that computes the distance between each line segment and the mouse click. Store, Lock, and Unlock methods are added to enable the objects for a shared group. Store saves the object in a database and transmits the object to other stations (reference the earlier SFILES discussion). Lock and Unlock extend the single user Select operation to lock the objects in the database and transmit a message to other stations, which causes the object to change appearance on screens in the multiuser environment.

The use of Gesturing (often referred to as telepointers) enables one person to use a mouse to point at objects on the other users' screens. Gesturing is described more fully later in this paper. The TGesture object was developed for TeamGraphics and broadcasts the cursor locations among users. It has methods for sending, receiving, showing, and hiding gestures. Our analysis of using TeamGraphics determined that gesturing capabilities should be part of other GSS projects. The gesture object was modified to make it more general and easier to integrate, and it became part of a GSS class library. To date it has been used in five or six other applications.

In order to take advantage of the potential in C++ for code reuse, it is mandatory to include a review of existing objects (class libraries or homegrown objects) as part of the initial functional design of a program and to perform a postproject review of new objects identifying those that should be generalized, documented, and added to class libraries for future reuse.

Issues, experiences, and pitfalls. A major issue concerning development in C and C++ is the selection of a memory mode. For all but the largest applications the small-memory model is suggested because current large models require too much overhead. Perhaps with the advent of 32-bit operating environments, this limitation will disappear. The small model is adequate, due to global memory and Windows providing most of the user input/output routines within its DLLs. Several applications can share code and data space, thus reducing the size of the executable modules and dynamic memory required for each active instance. Developers creating their own libraries should make as many of them into DLLs as practical.

There are several implementation pitfalls developers should keep in mind. First, developers should beware of implementation differences between network file servers. A particularly unfavorable feature of the Novell file server software is that the file access and sharing modes are not maintained for each file handle. Rather it seems that they are maintained on a workstation basis. For example, if an application opens a file with Read/DenyNone, and then opens the file ReadWrite/DenyWrite without closing the first file handle, the access mode of the first handle will be magically changed from Read/DenyNone to ReadWrite/DenyWrite. This will unintentionally prevent write access to the file. For single-user applications this would not often occur because a file is normally opened for exclusive write access. However, GSS applications must be written such that the elapsed time a file is opened for write access is minimized. A function that requires the scanning of a large data file to change certain records would ideally perform the scan in a read-only mode, opening the file for write access only when a record is located that requires an update.

In order to avoid this problem on a Novell network, the application should scan the file (read-only), find the record that needs updating, and close the file. Then, the file should be reopened for write access, updated, and then closed. Finally, the file should be opened again for read access, seeking the position where the previous scan left off.[25] We acknowledge that this implementation work-around could lead to update anomalies.

Second, the use of shared modes on files precludes local buffering of file data, thereby reducing file access performance. Applications should read and write data in large blocks or on a record basis to avoid this problem. A byte-by-byte input

of a text file should be avoided at all costs. The SFILES.DLL described earlier solves this problem by providing local buffering of shared file data.

Group features include: pointers, gesturing, view synchronization, and file access modes.

SFILES routines keep the contents of local buffers consistent by broadcasting updates (via the network interface object) and by making temporary use of exclusive write access (DenyWrite) to the server in order to synchronize updates.

When writing Windows routines that serve as drivers to direct memory access devices, it is necessary to ensure that the input/output buffers of the driver are locked in memory. With Windows running in *real mode* this can be done by declaring the appropriate data segment as Fixed. In 386 enhanced mode, the Fixed definition of data segments does not preclude them from being moved. To ensure that the data block is not physically relocated by Windows during input/output operations, use the GlobalPageLock function.

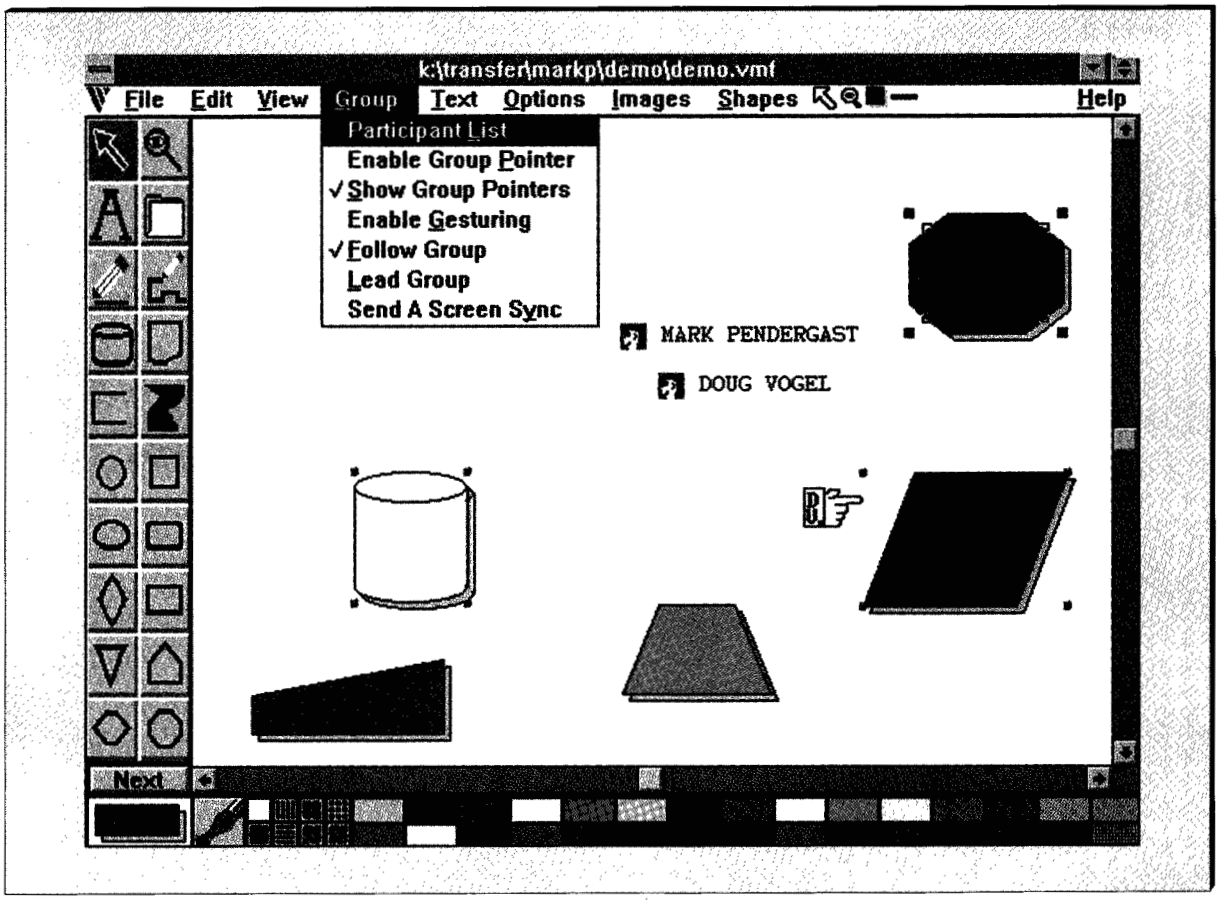
Finally, message queues for Windows are limited in size. If two applications on the same station are communicating via the PostMessage function, it is necessary to control the exchange such that a queue overflow does not occur. This can be done by having each application acknowledge the receipt and processing of each message or by having a sending application allow the receiver to execute between message transmissions. The context switch necessary for the latter method can be generated using the PeekMessage function in Windows. PeekMessage examines the caller's message queues for a range of messages, returning control to the caller only if there is a message present or if all messages for all programs have been processed. In order to guarantee that the context switch occurs, the PeekMessage should be invoked for a message that is not likely to be in the queue of the caller.

Other incompatibilities between network systems and bugs in the Windows software will not be discussed here. Suffice it to say that GSS debugging is a challenge. Currently, we have built only one application to assist in debugging; we use a network 'spy' program to show which messages are being sent and when.

Group work with TeamGraphics. TeamGraphics (formerly named MUGE) supports logical design methods such as data flow diagrams and entity-relationship diagrams, as well as unstructured drawing and annotation. It also provides standard editing features such as cut, paste, copy, delete, multiple fonts, pen styles, colors, and drill down. Drill down is a technique used in computer-aided design/computer-aided manufacturing (CAD/CAM) tools and graphics tools where one diagram is linked to another. The second diagram usually presents a closer view of the item, e.g., a structure chart or tree format might be used to represent a program. "Drilling down" through any node on the tree might result in the retrieval of a second diagram that represents the data flow for that program. In the editor window (see Figure 8) a variety of graphical objects are available, i.e., rectangles, diamonds, lines, circles, ellipses, and round rectangles. TeamGraphics has incorporated both synchronous and asynchronous work group coordination features. Synchronous work requires features to aid in the coordination of group efforts, while asynchronous work necessitates features that encourage the continuity of the work. Group features include: stationary pointers, gesturing, view coordination, view synchronization, and multiple file access modes.

The original prototype development of TeamGraphics took eight person-months. With TeamGraphics, each user is free to work on any part of a diagram, with any zoom factor, independent of other users. Each user may be editing multiple documents as well as multiple views of the same document. Editing may be performed in a private mode that provides exclusive access to a shared design, a shared mode that allows access by any number of users, or a local mode. Local mode is used for documents residing on the user's individual workstation, which are not known or not available to the group (i.e., private scratch pads, or past designs being imported into current work). Design repositories are stored on a shared disk. Changes to a shared context are broadcast to all stations concurrently accessing the design to al-

Figure 8 TeamGraphics editor window; an example of a windows-based Group Support Systems tool



low for real-time view updates. TeamGraphics employs the shared file system (SFILES) previously described for data storage, and object-oriented communications (NIO) for synchronous control functions such as group pointers and view coordination.

TeamGraphics supports specialized access to drawings via a configurable menu mechanism. For example, if read-only access is desired, then the menus could be configured to remove editing and object creation features. If append or add-only access is desired, then editing functions could be removed, but object creation features could be left in place. This configurable menu mechanism also provides a means for limiting the different objects that could be added, e.g., allow only rounded rectangles, boxes, lines for data flow diagrams, or for changing the names of objects on the menus (Diamond could be changed to Decision for flow charting, Rounded Rectangle could be changed to Process for data flow diagrams). This allows TeamGraphics to support many diagramming methodologies, while hiding unwanted or unneeded features from the users.

Locking performed by TeamGraphics has evolved with use. Initially, no locking of objects was done; thus if two people modified an object at the same time, the second modification was applied. This presented the problem of losing work without giving proper notification. A second method, duplication, was then tried. Under this method, if two users change an object at the same time, the original object is deleted and both of the revised objects are added. This provides a good mechanism

for side-by-side comparison of changes, but requires additional steps to integrate work and resolve changes (defeating the Shared Context Model). Finally, an object-level locking mechanism was implemented. When a user selects an object, black "handles" appear around the object on the selection screen, and red "handles" appear on all other screens. The user with the black handles is then allowed to change the attributes of the object. Once the object is unselected all handles are erased, the updated object is drawn on all stations, and other users are then allowed to modify the object.

Common views. There are times when all users must coordinate their respective views of the data set to discuss design alternatives. TeamGraphics uses three mechanisms for establishing common views of a shared design: (1) screen synchronization, (2) view coordination (or view slaving), and (3) diagram labeling. *Screen synchronization* gives users the ability to broadcast their view parameters (origin and scale factors) to all users editing a design. Upon reception the user is given the option of accepting or rejecting the screen synchronization. *View coordination* or slaving allows users to follow automatically all view changes of a leader. Each time the leader changes a view via zooming or scrolling mechanisms, the new view parameters are broadcast and applied to those users who have "slaved" their view window to the leader. *Diagram labeling* allows users to add diagram marker objects to a drawing. Users can then perform a Go-to-marker operation to move their view to a given work space. Additionally, user markers exist for each participant. User markers serve a dual purpose. First, they let a user know which other users are currently working in a given region of the diagram, and second, they can be used in the Go-to-marker operation if a user wishes to synchronize with someone else. Since users are allowed to have multiple views of a design at one time, they may choose to work in one window and monitor the work of someone else in another.

Stationary pointers. An essential aid in conducting group discussions in both the local and distributed modes is the ability to point at something on another user's screen. This ability is sometimes called "telepointing."[2] Stationary pointers in TeamGraphics are implemented using a mouse button to initiate a broadcast of the cursor location. Those users that have the stationary pointer enabled and are actively working in the corresponding window will have a special pointer drawn at the appropriate location. A variation of the stationary pointer is a point and jitter operation. Under *point and jitter* the pointer is drawn and erased multiple times around the desired location. This simulates movement of the cursor, which draws attention to the pointer. This is particularly useful with large groups using multiple stationary pointers or for distributed groups.

There are times when all users must coordinate their views of the data to discuss alternatives.

Gesturing. Gesturing is essentially full motion telepointing. In order to use gesturing, the user must select the appropriate menu option to turn it on. TeamGraphics will then automatically broadcast the cursor location whenever the mouse is moved. Depending on network bandwidth, the gesture broadcast routine can limit the number of messages transmitted each second. That is, Windows may generate 30 or 40 mouse movement (WM_MOUSEMOVE) messages a second, but TeamGraphics will only broadcast five sets of cursor coordinates a second. If too many gesture messages per second are broadcast, then the local area network will get bogged down, creating flow control and congestion control problems. If too few gesture messages are transmitted each second, then gesturing will appear jerky at the receiving end. Gesture movements can be smoothed at the receiving end by drawing and erasing the cursor multiple times while varying the X, Y position from the previous location to the new location.

Stationary pointers and gesturing, when used with screen coordination, provide a powerful method for maintaining a group focus during the presentation and discussion of designs.[22] TeamGraphics also enhances group work by allowing for increased parallel activity on a project without

the penalty of design integration experienced by single user editing tools. The synergistic effects created by a shared workspace and greater communications bandwidth could result in an increase in design quality. The key for achieving these gains is to minimize process losses using the special group control and coordination features incorporated into TeamGraphics.

Change tours. Change tours are automated presentations of specific areas of a design that have been modified. As users create a diagram they have the ability to add diagram marker objects. These objects are represented as a small flag and are given a name at the time of creation. Change tours consist of an ordered set of these flags. When users execute a tour, their screen is moved from one flag to another. Annotation objects can be added to explain and question a portion of the diagram.

Summary. TeamGraphics enhances group work by allowing for increased parallel activity on a project without the penalty of design integration experienced by single-user editing tools. The synergistic effects created by a shared work space and greater communications bandwidth could result in an increase in design quality. The key for achieving these gains is to minimize process losses using the special group control and coordination features incorporated into TeamGraphics.

Actor development environment

Actor is a complete object-oriented development environment and programming language for Windows.[26] Actor allows the creation and modification of windows, menus, and dialog boxes, as well as the running of applications and interactively debugging them within the development shell. The object-oriented programming language[27] for Actor consistently takes advantage of the benefits of inheritance and messaging. Its hierarchy closely mimics that of Smalltalk, but does not have the drawbacks of the model-view-controller paradigm[28] in that only a window that responds to messages needs to be developed. Applications are incrementally compiled and can be *sealed*, which means removing an application from the development environment so that it can run as an independent Windows program.

Capabilities. In a completely object-oriented environment like Actor, everything is an object: numbers, arrays, files, even windows. All objects have their own data (attributes, class attributes) and behavior (methods, class methods). Strong data typing is not enforced. Polymorphism, the sending of the same message to different types of objects, is supported. Polymorphism of the Draw message into each class of objects that must be displayed in the window is extremely powerful. An application merely maintains a set of objects to draw, and when Windows paints the user's view, each object may or may not be made visible.

Actor also supports all calls to Windows functions (automatically translating arguments from Actor representation to the required format), direct calls to C or Assembler primitives, and calls to dynamic link libraries (DLLs). Applications of up to 2 megabytes (MB) in size can be built (1MB code, 1MB data). Actor has an object-oriented interface to global memory management, Windows message translation, Windows procedure calls, and Windows callbacks. Using a profiler function, performance bottlenecks can be identified and rewritten in C or assembler. Actor provides an optimized, dynamic, incremental garbage collector that automatically removes unreferenced objects from memory. It also has an object change notification system that allows an object to be alerted when any attempt is made to change its associated data. The resource compiler for Windows is supplied and allows custom bit maps, icons, dialogs, cursors, menus, and defined constants to be changed in the executable module without changing the image.

Data that are sent back and forth between Actor and C are straightforward for simple data types, (i.e., integer, character, and string objects) because Actor handles conversions automatically. However, for more complex objects such as a data package that will be sent between stations, data must be converted to a C STRUCT data structure. Actor provides flexible support for C data structures by providing a CStruct class that supports STRUCT definition and manipulation.

There are also classes that facilitate calling a DLL. Interacting with DLLs can present a few problems if the argument lists for the procedures in the DLL are not specified correctly (e.g., an unrecoverable application error message). However, once the

mapping from Actor objects to C STRUCTS is made, a DLL becomes part of the application. Thus, the previously developed DLLs can be seamlessly reused.

Issues, experiences, and pitfalls. There are two major advantages to prototyping research software in Actor. The first is the object-oriented paradigm and the second (to be discussed later) is the development environment itself. In the object-oriented paradigm, generic classes can be developed, each with a certain level of functionality (see Figure 9). These classes can then be completely reused when new prototypes are built. In addition to the TObject classes described earlier in TeamGraphics, the first classes to be developed were:

Object
- *GroupObject*
  - *Issue*
  - *Comment*
- *Network*
- WindowsObject
  - Window
    - *GroupWindow*
      - *GraphicWindow*

Object, WindowsObject, and Window classes are provided as part of Actor. The Network class shields the system from the network base object (NBO) dynamic link library and thus the messages that the class responds to are:

- New—A new network object is created and global memory reserved for buffers. The NBO DLL is loaded.
- Open—A session for the object passed is created. All stations can now see this object.
- Send—The passed data block to the object currently open is sent. All stations with a session "opened" on that object will receive the data block.
- Receive—The data block is pulled out of the network buffer and converted to an Actor object. This object is then parsed and acted upon depending on type. If it is a message, the message is sent to the local object.
- Close—A session on the object is shut down.
- Destroy—A session is closed, the network object removed, buffers are deallocated, and the DLL is discarded.

The GraphicWindow class handles all tasks related to graphic diagrams, i.e., mouse clicks and drags, and zooming. It performs "world coordinate system to viewport" translations and manages scrolling functions. Transformation of cursor coordinates between the sending and receiving views must be performed. Different computer screens have differing resolutions and if a windowing environment is used, windows can be placed on different parts of the screen. This transformation is easily done by mapping the cursor location to a world coordinate system at the sender and, (1) if that location is outside the receiver's view, discard the message, or (2) map the world coordinate to the receiver's viewpoint coordinates. It is extremely important that when one user points at a place in a local window or screen, the same logical place is "gestured at" on the remote stations. This class was reused, unchanged for every new graphical prototype built. The graphical issue analyzer (GIAWindow) class, described later, inherits all of the functionality of the GraphicWindow class.

Other classes include GroupObject, Issue, and Comment. The Issue and Comment classes support the Graphical Issue Analyzer; a Comment differs from an Issue in the amount of text, the color and size drawn on the screen, and the links maintained with Issues. Instead of making a Comment a subclass of Issue and adding functionality, a generalization class (GroupObject) was created. GroupObject handles the commonality that exists in the two classes, i.e., draw, drawLabel, getSurrogateKey. GroupObject was reused and extended for other applications. Once group functionality was embedded, implementation details could be forgotten and the object merely used. Concepts such as surrogate keys are extremely important in GSS, and apply when a message is sent from one station to the others requiring the label of an object to be updated and the other stations must have a way of uniquely identifying the object.

Our experience with building several applications has shown that building the first application (Brainstorming) required three months. The second application (Graphical Issue Analyzer) required less than two months, and a third application (Group ScratchPad) was built in 10 days. The class reusability and debugging facilities contributed entirely to this successive reduction in development time.

The second major advantage to prototyping research software in Actor is the development environment itself. Class browsers aid in the creation and editing of new and existing classes. Applications are shielded from the basic Windows environment and, if they terminate abnormally, they fail gracefully through interactive debugger dialogs. Code can be altered while the application is executing, which allows actively fixing the error. Objects can also be inspected during execution to further aid in the debugging process. High-level errors are easily handled in this manner. Low-level errors are more difficult to detect, e.g., Windows has a classic error where available memory will decline (and the session crash) if handles to the display context are not released in the proper sequence. This error was almost impossible to detect using the Actor debugging facilities, but was eventually discovered by observing the decline of system resources.

One issue with using Actor is the size of the executable module and its related image. For a simple application this can typically be 300 kilobytes (KB) and, when loaded, this reduces working memory by more than this amount due to allocations for local heap space. Two mitigating factors exist: (1) the cost of extra memory is quite low—a large GSS lab with 34 stations could be upgraded to make this problem disappear; (2) a complex and large application is typically only 400KB—an indication that a certain set of resources is required, but additional code is well optimized.

Another issue with Actor is that performance is less than that of an environment using C. Through our network testing we have found that an Actor application performing the same task as a C application will take approximately 20 percent longer. We used the gesturing capabilities of TeamGraphics and the graphical issue analyzer, both merely pass point messages to the NBO and *bitblt* new cursors on the window. Bitblt is a graphic term for when one image is drawn over another. This performance penalty is due to Actor's constant searching for object lists for actual data addresses in contrast to direct addressing by C. For prototype development or delivery to high performance platforms, this penalty is minimal compared to the ease of development advantage. Once the object-oriented paradigm has been learned, Actor provides an excellent platform for prototyping and testing GSS research software.

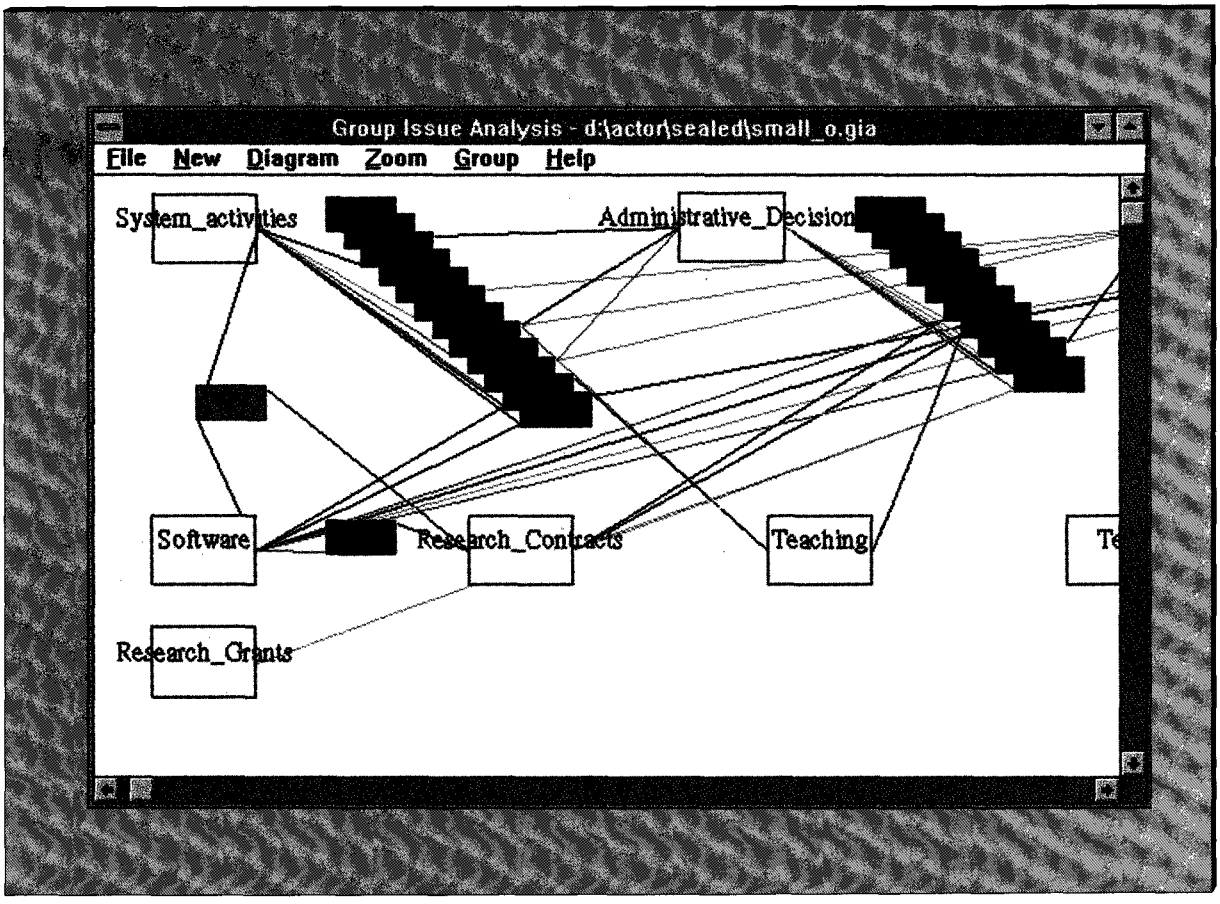
Graphical issue analyzer. The second of four GSS applications prototyped in Actor was the graphical issue analyzer (GIA).[29] For a description of others see Reference 16. The process of issue analysis is one of revealing the assumptions that a group is operating under. Once the core issues are uncovered, the group can begin the work of prioritizing the activities needed to deal with the issues (if they are to be dealt with at all). Research has shown the efficiency and effectiveness gains that exist when groups share text electronically. Further effectiveness can be realized by groups sharing graphical symbols (and text) such as Issues and related Comments. Also, the output may more accurately reflect the group's thoughts by having the group follow a process model consisting of three steps: (1) create your own issues and comments, (2) share them with others, and (3) resolve simple naming conflicts. We believe that computer-mediated groups can share and represent their views with less redundancy and more effectiveness using a graphical representation, than groups merely sharing text or individuals working in isolation.

GIA supports simultaneous, multiuser, anonymous entry of Issue and Comment objects. GIA presents the user with what appears to be a blank sheet of paper. The mouse is used to select objects, to place the various objects on the screen, and to perform diagram management, e.g., zooming and suppression of displayed objects. Common views and gesturing (as previously discussed) are supported. A user can be either a leader, a follower, or both.

The following communication primitives are used by GIA to control the group operation:

- Register—Each workstation running GIA is registered with all other workstations using the same shared context.
- SendObject—Objects are broadcast to all nodes.
- ReceiveObject—Objects are received and queued for insertion into the graphic. This serializes the process and avoids contention. Naming conflicts are resolved as described later.
- Gesture—A special cursor is drawn for use by the originator of the message to draw other users' attention. This cursor is labeled with the user's abbreviated name (if provided).

Figure 9 The graphical issue analyzer (GIA) issue analysis window



- StopGesture—The cursor is erased from all participants' work surfaces.
- RequestVote—Affected users can be asked to vote on proposed changes to the issue diagram. Voting is anonymous.
- ResolveConflict—If the vote is not unanimous, an electronic discussion system is invoked among affected users.
- UnRegister—Outstanding work is finished and the connection is closed.
- ViewHere—The world coordinates of a leader's window is sent to all connected windows. Those windows following a leader will be set to the passed coordinates.
- AddIssue, UpdateIssue—Issue information to all stations connected to the shared context is sent.
- AddComment, UpdateComment—Comment information to all stations connected to the shared context is sent.

When the GIA is invoked, the main issue analysis window appears and users may proceed to create issues and comments that represent their thoughts (see Figure 9). Common views, gesturing, and version control (as described above) are embedded. Issues must be created before they can be commented on, but any issue can be commented on no matter who created it. When a new issue is to be entered, the user types in the name of the issue in a small pop-up dialog box. This issue is drawn on the screen as a blue square. If the diagram has been zoomed large enough so that text can be read, it is labeled. Users can type comments in a larger pop-up edit dialog box. When finished typing their comment, the user

must choose which issue or issues this comment is related to from a dialog box containing a list of issues created. As each issue is selected, the user must designate whether the Comment is for, against, or indifferent to the issue. Lines are drawn from the comment to the related issues in green, red, and black, respectively. The comment is placed beside the issue. As more comments are added, they are made to overlap each other like fanned playing cards, clustered around the issue first chosen.

Comments and the linking information are simply broadcast to all nodes, but when a new issue is created, its label is compared with all other issues currently available locally, and if there is a conflict, the user is not allowed to use that name. If no conflict exists, the issue is broadcast to all other nodes. When a node receives an incoming issue, it immediately compares the name of the issue with all other issues in its outbound queue. Again, if a conflict does not exist, the issue is added to the local object base and appears on the screen for use. If there is a conflict, the local object in the outbound queue is discarded (after informing the user). If the broadcast of objects is assumed to be atomic, this algorithm has been shown to avoid duplicate objects. Changes to the discussion occur in real time. New or updated objects are also written to the shared file.

When a label of an issue is altered, a dialog is carried out with all nodes to see if the proposed changes are acceptable. This dialog interrupts users with a window displaying the initial and changed object. Users must vote on the change before proceeding. If a majority of the nodes accept, the changed object is accepted. If the proposed changes are not acceptable, an electronic (or verbal) discussion between users can be initiated to discuss the proposed changes. A similar process is followed when deleting objects, although the object in the shared file is merely flagged as deleted to preserve a history of the meeting.

The GIA allows both divergent and convergent processes to coexist simultaneously. Some users can be merging issues and their related comments, while others can be creating new issues or comments. GIA discussions can also be arranged in a network by creating new diagrams. In this way, issues can be consolidated or exploded. Users can participate in multiple discussions as well as participate in multiple views of the same discussion. The final output of GIA is a text file with the comments following each issue.

Conclusions and future directions

In this paper we have addressed some of the practical considerations for Group Support Systems application developers. Two development environments, C++ and Actor, were discussed along with sample applications. Actor provides a powerful object-oriented environment that enables rapid prototyping and development of GSS tools. C and C++ provide the flexibility and performance necessary to support low-level communications protocols and file handling. Neither development environment taken by itself is the complete solution.

Maintaining a shared context and coordinating the actions of the users are the two most difficult challenges to Group Support Systems developers. This paper has presented how two applications, TeamGraphics and Graphical Issue Analysis, have made use of a specialized object-oriented communications system (NIO) and a distributed shared file system (SFILES) to deal with these challenges. The reliable broadcast capability of NIO is employed to maintain shared data, coordinate views, and transmit group pointers. SFILES provides a convenient and efficient mechanism for coordinating access to shared files and the maintenance of what-you-see-is-what-I-see (WYSIWIS) views. In addition to demonstrating the viability of GSS on PC-based systems, these programs also demonstrate the ability to develop complex applications in different programming environments that make use of common routines and share a common network object architecture.

The future direction of personal computer systems includes a major role for Group Support Systems. Products such as Lotus Notes**, GroupSystems V, and a multitude of database systems that have been enhanced to support group work (FlexBase, Paradox, DBase IV) on personal computers provides evidence for this prediction. More research into communications architectures and user interaction with shared contexts is required before more traditional single-user PC programs, e.g., spread sheets, can be adapted for Group Support Systems. In addition, standards and certification processes must be created and enforced for network file systems and peer-to-

peer communications. Only preliminary evidence exists as to whether the shared context model as described in this paper will allow Group Support Systems groups to outperform manual groups or groups supported by other software.[29–31] More research into how groups interact using graphical systems is required. Defining the maximum group size for graphical interaction, and how dispersed groups can be supported given multitasking, graphical platforms, are unanswered issues. OS/2 and Windows have opened a new door for Group Support Systems.

Acknowledgments

This research was sponsored in part by IBM Research Grant #436800 (United States), and by IBM Canada.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Ventana Corporation, X/Open Co. Ltd., Microsoft Corporation, ProtoView Inc., MegaSoft Inc., Paradox, or Lotus Corporation.

Cited references and note

1. G. DeSanctis and B. Gallupe, "A Foundation for the Study of Group Decision Support Systems," *Management Science* **33**, No. 5, 589–609 (May, 1987).
2. S. Ellis, J. Gibbs, and G. Rein, "GroupWare: Some Issues and Experiences," *Communications of the ACM* **34**, No. 1, 38–58 (1991).
3. S. Ellis and J. Gibbs, "Concurrency Control in Groupware Systems," *SIGMOD Record* **18**, No. 2, 38–59 (June, 1989).
4. J. Goodman and M. Abel, "Communication and Collaboration: Facilitating Cooperative Work Through Communication," *Office: Technology and People* **3**, No. 2, 129–145 (1987).
5. S. Greenberg, M. Roseman, D. Webster, and R. Bohnet, "Issues and Experiences Designing and Implementing Two Group Drawing Tools," *Proceedings of Hawaii International Conference on System Sciences* **4**, (1991), pp. 139–148.
6. D. Stefik, G. Bobrow, G. Foster, S. Lanning, and D. Tartar, "WYSIWIS Revised: Early Experiences with Multi-User Interfaces," *ACM Transactions on Office Information Systems* **5**, No. 2, 147–186 (April, 1987).
7. G. Stefik, D. Foster, G. Bobrow, K. Kahn, S. Lanning, and L. Suchman, "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings," *Communications of the ACM* **30**, No. 1, 32–48 (January, 1987).
8. *Group Support Systems: A New Frontier*, L. Jessup and J. Valacich, Editors, Macmillan Publishing Co., New York (1993).
9. S. Franklin and A. Peters, "Effective Application Development for Presentation Manager Programs," *IBM Systems Journal* **29**, No. 1, 44–58 (1990).
10. "Microsoft Windows 3.0," *Software Development Kit Manual*, No. 050051052-I02-1087, Microsoft Corporation (1991).
11. H. Simon, *Administrative Behavior: A Study of Decision Making Processes in Administrative Organizations*, The Free Press, New York (1976).
12. A. Dennis, J. George, L. Jessup, J. Nunamaker, and D. Vogel, "Information Technology to Support Electronic Meetings," *MIS Quarterly* **12**, No. 4, 591–694 (1988).
13. A. Osborn, *Applied Imagination: Principles and Procedures of Creative Thinking*, Charles Schribner's and Sons, New York (1953).
14. I. Posner, R. Baecker, and M. Mantei, "How People Write Together," *Proceedings of the Hawaii International Conference on System Sciences* (1992), pp. 127–138.
15. S. Navathe, R. Elmasri, and J. Larson, "Integrating User Views in Database Design," *IEEE Computer*, 50–62 (January, 1986).
16. M. Pendergast and S. Hayne, "A Collaborative Systems Approach for Alleviating Group Convergence Problems in CSCW Processes," *Journal of Computer-Supported Cooperative Work*, forthcoming.
17. M. Pendergast and D. Vogel, "Design and Implementation of a PC/LAN-based Multi-User Text Editor," *Proceedings of the IFIP 1990 Conference on Multi-User Interfaces* (1990).
18. L. Killey, "ShrEdit 1.0: A Shared Editor for the Apple Macintosh User's Guide and Technical Description," ShrEdit Development Team, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, Ann Arbor, MI (1990).
19. K. Aytes, "An Empirical Investigation of Collaborative Drawing Tools," unpublished doctoral dissertation, MIS Dept., University of Arizona, Tucson, AZ (1993).
20. M. Pendergast, "Multicast Channels for Collaborative Applications: Design and Performance Evaluation," *ACM Computer Communications Review* **23**, No. 2, 25–39 (April, 1993).
21. S. Hayne and S. Ram, "Group Database Design: Addressing the View Modeling Problem," *Journal of Systems and Software*, forthcoming.
22. S. Hayne, M. Pendergast, and S. Greenberg, "Implementing Gesturing with Cursors in Group Support Systems," *Journal of Management Information Systems* **10**, No. 3, 43–62 (1994).
23. M. Pendergast, "Distributed Object-Oriented Environment for EMS Application Development," *Proceedings of the Hawaii International Conference on System Sciences* **1** (1991), pp. 59–66.
24. R. Ten Dyke and J. Kunz, "Object-Oriented Programming," *IBM Systems Journal* **28**, No. 3, 465–478 (1989).
25. This unfavorable situation only appears in Novell versions after 2.0 (up to the time this paper was written) and does not occur in IBM local area network software.
26. WhiteWater Group, "Actor 2.0," *User and Reference Manual* (1991).
27. C. Duff, "Designing an Efficient Language," *BYTE* **12**, No. 8, 211–224 (August, 1987).
28. S. Goldberg, "Introducing the SmallTalk-80 System," *Byte Magazine* **6**, No. 8, 14–26 (August, 1981).
29. S. Hayne and T. Purdin, "A Distributed Tool for Issue Analysis," *Proceedings of the Symposium on Applied Computing* (April 1990), pp. 325–329.
30. J. Conklin and M. L. Begeman, "gIBIS: A Hypertext

Tool for Exploratory Policy Discussion," *ACM Transactions on Office Information Systems* **6**, No. 4, 303–331 (October, 1988).
31. G. Easton, J. George, J. Nunamaker, and M. Pendergast, "Using Two Different Electronic Meeting System Tools for the Same Task: An Experimental Comparison," *Journal of Management Information Systems* **7**, No. 1, 85–100 (1990).

Accepted for publication August 24, 1994.

Stephen C. Hayne *Business Programs, Arizona State University West, 4701 W. Thunderbird Road, Phoenix, Arizona 85069-7100 (electronic mail: hayne@asu.edu).* Dr. Hayne recently joined Arizona State University West as an assistant professor in management information systems. He received his Ph.D. from the University of Arizona in 1990; his dissertation addressed the distributed database design process as conducted by groups. While teaching courses at the University of Calgary on software engineering, telecommunications, and database systems management, his teaching was recognized with awards for excellence. Dr. Hayne's research interests lie mainly in distributed database design, software engineering, knowledge-based technology, and group support systems. His papers have been published in major conferences as well as the *Journal of Information and Management*, the *Journal of Computer Supported Collaborative Work*, the *Journal of Systems and Software*, the *Journal of Management Information Systems*, and others. Much of his research is rooted in the desire to use innovative technology to solve real business problems. His current work involves Group Support Systems, and as such he has implemented tools in graphical environments to assist groups in communication and decision-making, i.e., shared drawing, group brainstorming, concurrent issue surfacing, and consolidation. He is applying this technology to support decision-making during time pressure situations.

Mark Pendergast *MIS Department, McClelland Hall 430, College of Business and Public Administration, University of Arizona, Tucson, Arizona 85721 (electronic mail: pendergast@mis.arizona.edu).* Mr. Pendergast is a research director in the Center for Information Management, Management Information Systems Department, at the University of Arizona. He received his Ph.D. in MIS in 1989. His research interests include computer-supported cooperative work, data communications, software engineering, process re-engineering, and group support systems human computer interaction. He has worked as an analyst and engineer for Control Data Corporation, Harris Controls, Ventana Corporation, and as an assistant professor at the University of Florida. His work has appeared in several books and journals, and he has presented his work at numerous conferences.

Reprint Order No. G321-5560.