# Window Sharing with Collaborative Interface Agents

Charles Rich

TR95-12    April 1995

**Abstract**

An implemented system is described which allows a software agent to collaborate with a human user using a shared application window. The system automatically controls input permission and also provides mechanisms for signalling and communication. A generalization of the system to multiple users and agents, called NShare, is compared with common window-sharing tools, such as SharedX. This work is part of a larger agenda to apply principles of human collaboration and discourse structure to human-computer interaction using the interface agent paradigm.

*SIGCHI Bulletin, Vol. 28, No. 1, January 1996, pp. 70-78*

**Publication History:–**

1. First printing, TR-95-12, April 1995

# Window Sharing with Collaborative Interface Agents

Charles Rich
Mitsubishi Electric Research Laboratories
201 Broadway
Cambridge, MA 02139
rich@merl.com

**ABSTRACT**
An implemented system is described which allows a software agent to collaborate with a human user using a shared application window. The system automatically controls input permission and also provides mechanisms for signalling and communication. A generalization of the system to multiple users and agents, called NShare, is compared with common window-sharing tools, such as SharedX. This work is part of a larger agenda to apply principles of human collaboration and discourse structure to human-computer interaction using the interface agent paradigm.

**KEYWORDS:** software agents, interface agents, window sharing, collaboration, multi-user, multi-input.

## 1 INTRODUCTION

This paper describes a generic interface paradigm for use with software agents. Software agents are currently a new research area in which generally accepted terminology and principles have yet to emerge. Roughly speaking, a software agent is an autonomous software process which interacts with humans as well as with elements of its software environment, such as the operating system, mail programs, and other applications. Usually, what the agent does is some kind of intelligent assistance and is presented in a human-like fashion.[1]

Most current work on software agents is very application-specific. This paper reports on the design and implementation of an application-independent graphical interface, based on window sharing, for what I call *collaborative interface agents.* Figure 1 illustrates some of the key properties of a collaborative interface agent.

- *Collaborative:* The purpose of the agent is to assist and cooperate with a human user in the performance of some computer-based task. This implies, among other things, that the agent must be able to communicate with and observe the actions of the human user and must be able to interact with whatever application programs are used to perform the task.

[1]In fact, my personal working definition of a software agent is any program for which anthropomorphization (ascribing some human properties) is *helpful.* Obviously, ascribing human properties to most software is very unhelpful.

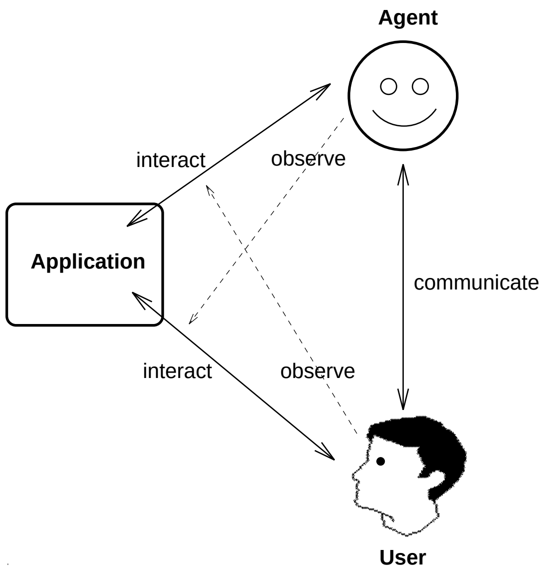
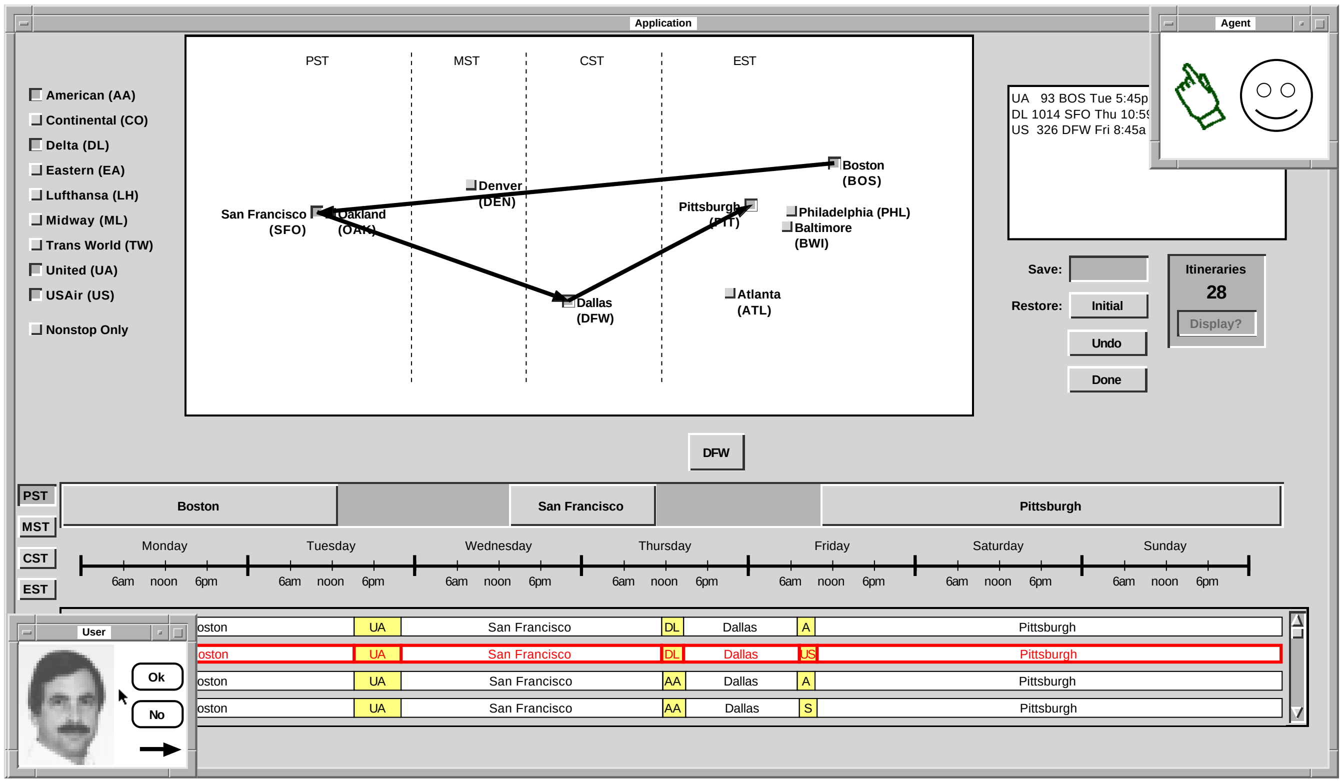


Figure 1: Collaborative interface agent paradigm.

- *Interface Agent:* The agent interacts with shared application programs through the same interface used by a human user in a way that can be observed by a human user. This approach facilitates the reuse of existing applications and supports collaboration by making it it easy for the user to know what the agent is doing.

This concept of a collaborative interface agent is closest to the work of Patti Maes[5], although she uses the term "collaborative" to refer to the sharing of information between agents.

The agent paradigm in Figure 1 intentionally mimics the relationships that hold when two humans collaborate in a task involving a shared artifact, such as two mechanics working on a car engine together, or two computer users working on a spreadsheet together. This is intentional because this work is part of a larger effort [7, 9] to apply known principles of human discourse and collaboration to human-computer interaction. The work reported in this paper primarily concerns the mechanisms of the interaction between humans and interface agents, rather than the content. Work in progress concerning discourse



UA 93 BOS Tue 5:45p
DL 1014 SFO Thu 10:5
US 326 DFW Fri 8:45a

| | | | | | | |
|---|---|---|---|---|---|---|
| oston | UA | San Francisco | DL | Dallas | A | Pittsburgh |
| oston | UA | San Francisco | DL | Dallas | US | Pittsburgh |
| oston | UA | San Francisco | AA | Dallas | A | Pittsburgh |
| oston | UA | San Francisco | AA | Dallas | S | Pittsburgh |

Figure 2: Screen shot of test tube system.

and collaboration issues with interface agents is briefly discussed in the Section 6.2.

The first part of this paper concentrates on the situation illustrated in Figure 1, in which there is one human user and one interface agent. Section 4 describes a generalization of the paradigm to multiple human users and/or software agents, including a comparison with other window-sharing systems, such as SharedX.

## 2 EXAMPLE SCENARIO

Figure 2 is a screen shot of an application that has served as a "test tube" for developing the ideas described here. Although careful attention has been paid to making the agent interface application-independent, it is often helpful to see a concrete example that motivates the work.

The shared application in Figure 2 is an air travel planning system (implemented for this research) that provides a direct manipulation interface to an airline schedule database and a simple constraint checker. By pressing buttons, moving sliders, and so on, the user can specify and modify the geographical, temporal, and other constraints on a planned trip. The user can also retrieve and display possible itineraries satisfying the given constraints.

A typical session using this application to solve a difficult travel problem (without the help of a software agent) takes about 15 minutes. The role of the collaborative interface agent is to help the user by, for example, suggesting strategies for systematically relaxing constraints, explaining how to recover from errors, and taking over parts of the search space itself.

### 2.1 Home Windows

In addition to the shared application window, Figure 2 shows that the user and agent each have an individual *home window*. As we will see below, these home windows are the main mechanism for direct communication between the user and the agent. The home windows start out in the default corner locations shown in the figure, but may be moved to different screen locations by the user in the usual ways provided by the window system. Furthemore, given a distributed window system like X, the agent process can be running on different machine than the application.

Each home window contains an identifying face and a cursor. The user's cursor is his usual mouse pointer, seen in the middle of his home window. The agent's cursor is the pointing hand icon shown in its home window. The agent uses this icon to point and click on the shared application window just as the user does. The agent's eyes blink periodically to indicate that its process is still running.

The example scenario continues in Figure 3. To save space in this and the remaining screen shots in the paper, the air travel application window is replaced by a simple nine-button panel representing a generic graphical application.

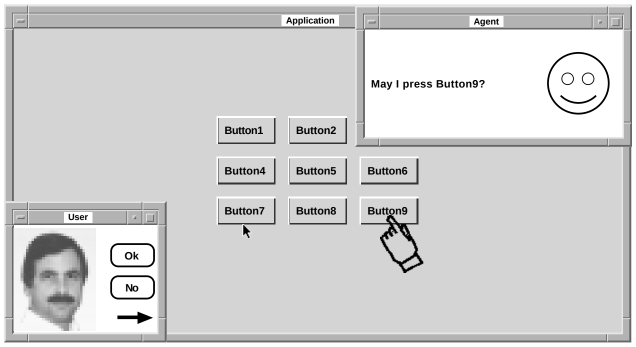


Figure 3: The agent points to and asks permission to press Button9.

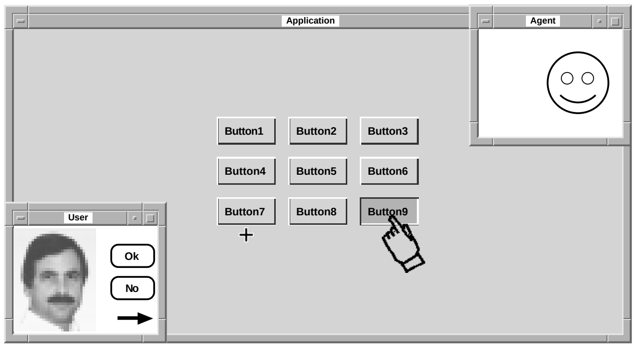


Figure 4: The agent grabs input permission and presses Button9.

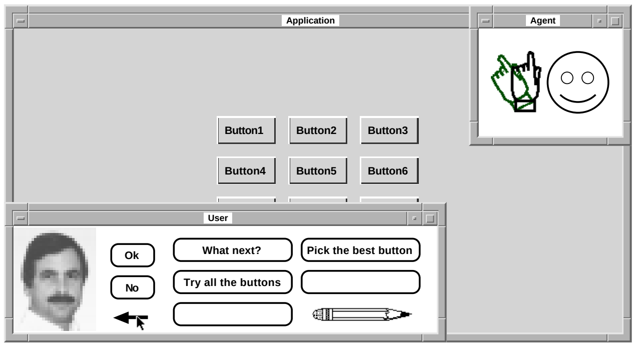


Figure 5: The agent signals for attention.

In Figure 3, several things have happened. First, the user has moved his cursor out of his home window onto the shared window and is pointing at Button7. The agent has also moved its cursor out of its home window and is pointing at Button9. Second, the agent has displayed a message (question) in its home window for the user to read. At this point, the user may reply to the agent's question by, for example, clicking on the "Ok" button. However, the user is also free to do anything else he wants on his home window or the shared window. The interface agent paradigm differs here from the typical pop-up dialogue-box kind of interaction, in which the main window is "hung" until the question is answered or dismissed. The interface agent interaction is asynchronous and mixed initiative, just like a two people collaborating with a shared artifact.

### 2.2 Input Permission

In Figure 4, the agent has grabbed input permission and is in the midst of pressing Button9. The user and the agent never simultaneously have input permission (see further discussion of this point in Section 4). Notice that the user's cursor, located on the shared window below Button7, has changed from its usual arrow shape to a cross. This is a visual indicator to the user that he does not currently have input permission. If the user presses a key or mouse button while in this state, the interface will beep and no events will be sent to the shared application. The user may, however, move his cursor while in this state.

When the agent has finished pressing Button9, it should relinquish input permission to the user, at which time the user's cursor will return to being an arrow. The user may also force the agent to relinquish input permission at any time by clicking on the agent's face in the agent's home window.

### 2.3 Signalling and Communicating

Finally, in Figure 5, two things are happening. First, the agent is signalling for attention by "waving:" the two hand icons shown overprinted in the agent's home window are displayed in quick alternation to give the appearance of movement. Signalling is a polite way for the agent to request control in a mixed initiative interaction. The user may, of course, choose to ignore the signalling, in which case a well-designed agent will desist.

For example, in the air travel planning application, the agent may signal because it has noticed that the user has overconstrained his trip. If the user clicks "Ok" in response to this signalling, the agent may then offer to appropriately relax some of the user's constraints.

The second feature illustrated in Figure 5 is the user's extended menu accessed by clicking on the horizontal arrow in his home window. Clicking on one of these menu items sends the corresponding message to the agent. This message may be the literal text string shown in the menu or some corresponding internal symbol. The list of menu items may be fixed or may be modifiable by the agent. Depending on the capabilities of the agent, the user may also, by clicking on the pencil icon, enter a text-editing interface in which a message may be composed in a natural or artificial language

## 3 ARCHITECTURE

Figure 6 shows the architecture of the system that implements the example scenario above. At the bottom of the figure are the application program and its associated window that are being shared between the agent and the user. At the top of the figure is the user's display which is generating window events in response to use of the mouse and keyboard.

In order to provide a well-defined software interface for agents, the system is divided into two modules. The *translator* module intercepts certain window events from the user's display, such as entering/leaving a home window or clicking the mouse button, and translates them into requests that are sent to the *controller* module. For example, whenever the translator sees a mouse click on the agent's face, it generates an Interrupt request. The controller request types and the translator's rules are discussed in detail in Section 5.

Agents send requests directly to the controller. For example, in order to grab input permission, the agent sends an Activate request to the controller. The return value from this request lets the agent know if its request was successful or not. There are also requests to query the state of the controller, such as to find out the current location of the user's cursor.

The controller generates window events on the home windows and the shared window. For example, in response to an Execute request, the controller will, assuming the requestor has input permission, generate the appropriate window events to simulate a mouse or key click on the shared window. The controller's main responsibility is to make sure that only one participant in the interaction has input permission at a time. The controller also also takes care of updating the graphics on the home windows.

As mentioned above, given a distributed window system like X, the agent process, the user's display process, the translator, the controller, and the application program may all be running on different machines.

The dashed line in Figure 6 indicates that a mechanism needs to be provided for agents to directly query the state of the application program. For example, the air travel planning agent needs know the sequence of cities the user plans to visit without trying to "reverse engineer" the pixels on the map display or asking the user for information that has already been entered. The best way to provide this mechanism in a generic way is to adopt a model-based approach to user interface design, which has other benefits relative to implementing collaborative interface agents, as discussed in Section 6.1.

## 4 MULTIPLE USERS AND AGENTS

Some collaborations involve more than two participants. The architecture in Figure 6 requires no fundamental changes to generalize to multiple users and agents, other than to say that the translator receives window events from multiple displays and the controller receives requests from multiple agents. In most window systems, window events are already coded with the source window, so that it is easy for the translator to generate separate controller requests for each user. Furthermore, the controller in the single user/agent case already keeps track of requests from multiple sources.

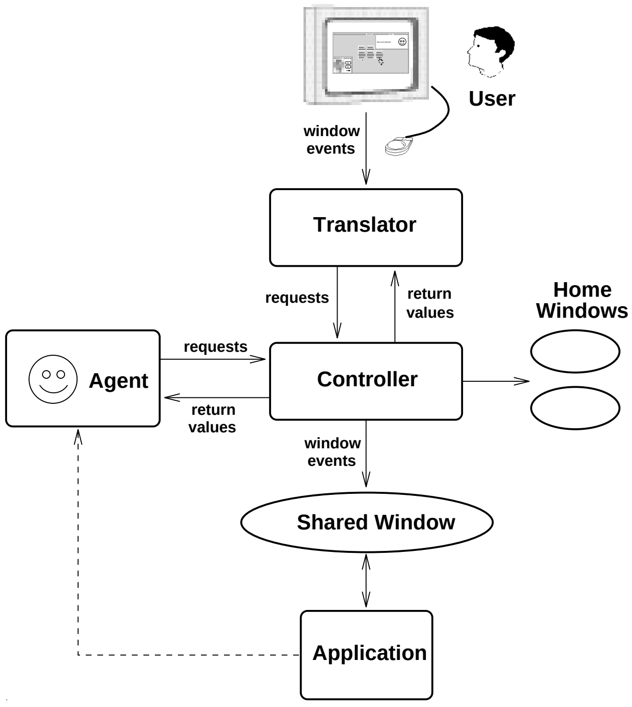


Figure 6: Collaborative interface agent architecture.

### 4.1 The NShare Interface

In moving from the special case of one agent and user, certain cosmetic changes described below need to be made to the interface in order to make the presentation more logical and symmetric.

Figure 7 shows the interface to a system, called NShare, for window sharing with multiple users and collaborative interface agents. In this example, there are two users, named Chuck and Dick, who are at different locations, and two agents, named Smurk and Glurk. Figures 7 and 8 show what is seen on Chuck's and Dick's displays, respectively, at the start of a short example scenario. (The agents don't have displays).

In NShare, the shared application window is replicated on all displays as shown. Each user has a *local* home window, i.e., his home window on his own display, and one or more replicated *remote* home windows, i.e., home windows on other user's displays. (Agents only have remote home windows.) Notice that a cursor hand icon now appears in all (both agents' and users') home windows. The cursors in these black-and-white screen shots are distinguished only by their orientation; on a color

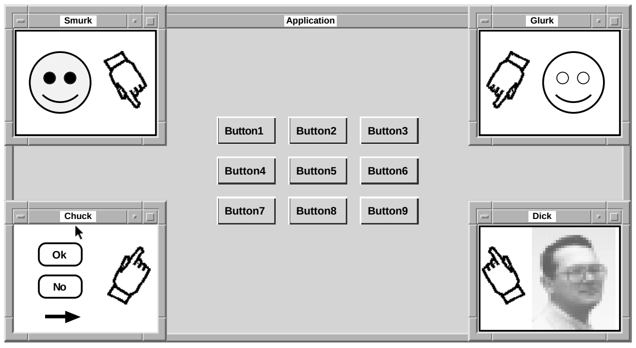


Figure 7: Chuck's display at start of scenario.

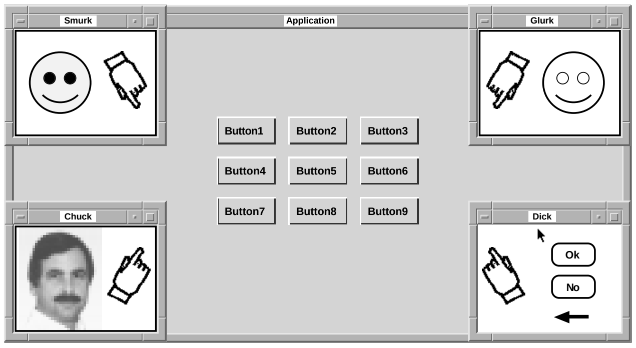


Figure 8: Dick's display at start of scenario.

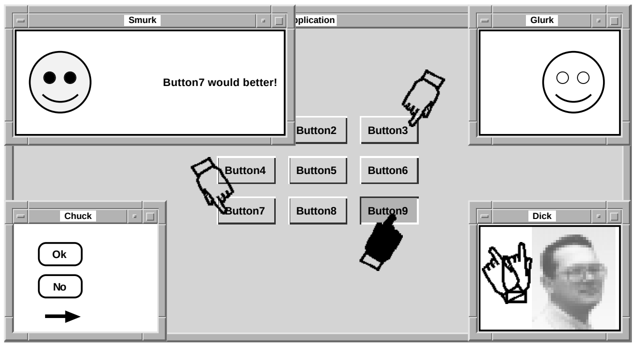


Figure 9: Chuck pressing a button (Chuck's display).

display the cursors are also color-coded to their corresponding home windows.

The hand icon in a user's home window serves two functions. First, with multiple human collaborators, it can be useful for users, not only agents, to have a way of signalling. A user can therefore click on the hand icon in his local home window to issue a Signal request, in which case his hand icon will start waving on all users' displays. Second, as we will see below, in NShare a user's mouse pointer becomes a hand icon when it moves onto the shared window.

Also, comparing Chuck's local home window in Figure 7 with his remote home window in Figure 8, notice that the communication menu that appears in a user's local home window is replaced by the user's face in his remote home windows.

### 4.2 Automatic Input Permission Control

With multiple users and agents, keeping track of input permission becomes slightly more complicated (see discussion of SharedX below). Each on-window cursor therefore has two modes. In the *active* mode, the cursor icon is filled in, meaning that the corresponding participant has input permission. In the *passive* mode, the hand icon is drawn in outline, meaning that the corresponding participant does not have input permission. For example, in Figure 9, Chuck's cursor is active and is pressing Button9, while Smurk and Glurk are passively pointing to other buttons and Dick is signalling.

Input permission is granted and relinquished in NShare as follows. All cursors start out in passive mode on their home windows, as in Figure 7. Agents change their cursor modes by sending controller requests. When a user moves his cursor onto the shared window, it remains in passive mode until he attempts to provide input by pressing a key or mouse button. At this point, the translator sends an Activate request to the controller, which succeeds if and only if no other participant is active. Users relinquish input permission by moving their cursor off the shared window.

### 4.3 Inter-Participant Communication

In a realistic application of NShare, particularly with current developments in network technology, the human users will most likely be in voice communication, as well as having shared windows. Therefore, although users can compose text messages in their home windows for other users to read, this is not likely to be the preferred mode of user-user communication.

However, until software agents start to use voice recognition technology, user-agent communication in the multiple user and agent interface will be based on the home window menus as in illustrated Section 2. An additional concern in this situation, however, is that, like a shared voice channel, these text/symbolic messages should be observable by all participants.

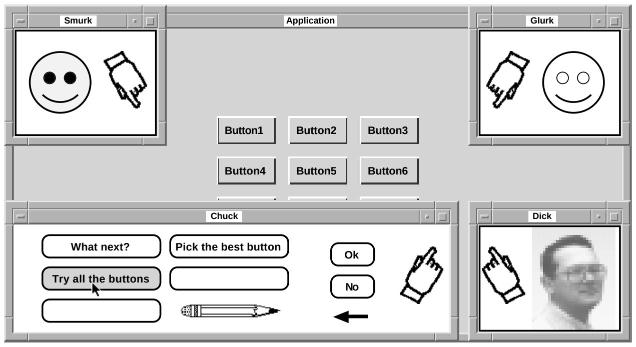


Figure 10: Chuck sending a message.

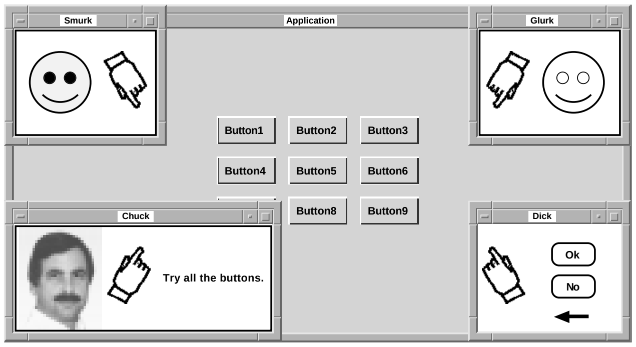


Figure 11: Dick's display at same time as Figure 10.

For example, in Figure 10, Chuck has selected the "Try all the buttons" message from his local home menu. Notice that this message text immediately shows up on all Chuck's remote home windows, e.g., on Dick's display in Figure 11. Smurk and Glurk, the interface agents, can receive Chuck's message by sending a GetMessage request to the controller with 'Chuck' as the argument.

As mentioned earlier, the work described here focuses on the mechanisms for communication rather than the content. It is up to the participants in NShare to identify the intended recipients of messages based on their content. For example, Glurk may respond to the "Try all the buttons" message rather than Smurk because Glurk is programmed to respond to this kind of message and Smurk is not. Alternatively, the user may explicitly identify an intended recipient through a submenu or as part of the text of the message. Managing the flow of conversation in multi-participant communication is an area for future research.

### 4.4 Comparison with SharedX

With the increasing recent interest in collaborative work, there have been many systems developed for sharing windows between multiple users. A typical such system is Hewlett Packard's SharedX. In addition to providing a well-defined interface (controller requests) for software agents, NShare makes some other modest improvements to SharedX and similar products.

Most current window sharing systems are extensions to the X Window System that allow real-time sharing of X-protocol-based applications between two or more remote displays. In such systems, the shared application window is automatically replicated on all users' displays, so that all users can simultaneously see changes in the state of the application. However, only users who have input permission are allowed to provide input to the application by mouse or keyboard. Typically, any number of users can have input permission at the same time. The list of users and their input permission state is displayed in a separate control window and changed via toggle buttons in that window.

Unfortunately, as acknowledged in the SharedX User's Guide:

> When sharing a window with several receivers [users] who have input permission, it may become confusing who is inputting at any given time.[2]

An obvious solution to this problem is to only let one user have input permission at a time. However, this is not the way such systems are usually used, because changing input permissions requires going to a separate control window, which is inconvenient and interrupts the flow of the collaboration. The NShare system remedies this difficulty by automatically granting and relinquishing input permission as described above.

When two people share a display in the traditional way, i.e., standing or sitting side by side, they often point to items on the display with their fingers or use the mouse to point out items. Unfortunately, SharedX does not replicate mouse pointers on all users' displays. Instead, SharedX provides separate "telepointer" icons that users have to explicitly click and drag around, which is less convenient than just pointing with the mouse as in NShare.

## 5 IMPLEMENTATION

The air travel and NShare prototypes described above were implemented in Common Lisp using the X Window System and the Garnet [6] graphics package. Window replication in the NShare prototype was handled inside the application program. A more practical implementation of NShare would make use of one of the many X protocol multiplexers, such as Xy [1].

The rest of this section describes in detail the controller requests and translator rules for the multiple user and agent system, which apply to the single user and agent system as a special case.

### 5.1 Controller Requests

Controller requests are generated by the translator in response to users' mouse and keyboard events or are received directly from agents (see Figure 6). Requests either *succeed* or *fail*. When a request succeeds, it may also return a *value* if specified below. Failing requests have no effect on the controller's state.

In each of the following specifications, "the requestor" refers to the user or agent issuing the request. Also, all

[2]HP SharedX 2.0 User's Guide, Chapter 4.1.

aspects of the controller state not explicitly specified to change are unchanged by a request.

**Permission Requests** The following requests may change the state of participants' input permission.

*Activate* – Request input control.

> Succeeds iff the requestor is on the shared window and no other participant is active. After successful completion, the requestor's cursor is active. If the request is successful and the requestor was not active when the request was issued, then a MotionNotify[3] window event is generated with the requestor's current location as argument.

*Deactivate* – Relinquish input control.

> Always succeeds. If the requestor is on the shared window, then the requestor's cursor is passive at end of the request; otherwise the request has no effect.

*Home* – Return home.

> Always succeeds. If the requestor is on the shared window when the request is issued, then the requestor is at home and passive at the completion of the request; otherwise the request has no effect.

*Interrupt* – Deactivate currently active participant.

> Succeeds iff there is an active participant at the time the request is issued. After successful completion of the request, the previously active participant is passive. The return value is the name of the previously active participant.

*Signal(on?)* – Turn signalling on/off.

> Succeeds iff the requestor is at home. After successful completion, the requestor's cursor is in the signalling mode if the given flag is true or in the passive mode if it is false.

**Geometric Requests** The following requests depend on the geometry of the shared application's graphical user interface. See Section 6.1 for a discussion of how to write the logic of an agent so that it does not have to deal with this geometry explicitly.

*Move(x,y)* – Move cursor.

> Always succeeds. The requestor's cursor is moved to the given x-y location on the shared window and, if the requestor is active, a MotionNotify(x,y) window event is generated. If the requestor is at home when the request issued, then the requestor is passive at end of the request; otherwise the requestor's cursor mode is unchanged.

*Execute(event)* – Execute window event.

> Succeeds iff the requestor is active. If the requestor is active, then the given window event is generated at the requestor's current location. The given event may be any window event supported by the window system, except MotionNotify (see Move request).

[3] I am using the X terminology here for the window event normally generated when the user moves his mouse.

**Status Requests** The following requests query, but do not change, the state of the controller.

*QueryAll* – All participants.

> Always succeeds. The return value is a list of the names of all current participants.

*QueryActive* – Active participant.

> Succeeds iff there is an active participant. The return value when successful is the name of the active participant.

*QueryDisplay(name)* – Participant's display.

> Succeeds iff the named participant has a display (i.e., it is a human user). The return value when successful is the name of the display.

*QueryLocation(name)* – Participant's cursor location.

> Succeeds iff named participant is on the shared window. The return value when successful is the x-y location of the participant's cursor.

*QuerySignalling(name)* – Participant's signalling state.

> Succeeds iff participant's cursor mode is signalling.

**Message Requests** The following requests concern communication between participants and do not modify the state of the controller.

Note that the message argument to these requests may be any symbolic object, not necessarily text in a natural language. It may also be convenient to define both an internal and external form of each message, such that the external form (e.g., English text) is shown in the message areas of the home windows, and the corrresponding internal form (e.g., a symbolic code) is returned by GetMessage.

*PutMessage(message,append?)* – Display message.

> Always succeeds. The (external form of the) given message appears in the requestor's message area. If the append flag is true, then the given message is appended to any message already being shown; otherwise previous message is cleared.

*GetMessage(name)* – Get named participant's message.

> Succeeds iff the given participant's message area is not clear. The return value when successful is the (internal form of the) message(s) being shown.

*ClearMessage* – Clear message area.

> Always succeeds. The requestor's message area is cleared.

### 5.2 Translator Rules

The following rules determine how the translator (see Figure 6) generates controller requests. In the following, "the pointer" refers to the user's pointer.

- Whenever the pointer leaves the shared window a Home request is generated.
- Whenever the pointer enters the shared window, a Move request is generated with the pointer's initial window coordinates as arguments. A ClearMessage request is also generated.
- Whenever the pointer changes location within the shared window, a Move request is generated with the new coordinates as arguments.
- Whenever the pointer is on the shared window and the user presses/releases a key or mouse button, an Activate request is generated. If the Activate request succeeds, then an Execute request is generated with the appropriate window event argument. (If the Activate request fails, then the local display beeps.)
- Whenever a user clicks on the face of another participant, an Interrupt request is generated iff that participant is active (otherwise the local display beeps).
- (NShare only) Whenever a user clicks on the cursor hand icon in his local home window, a Signal request is generated with a true flag if the user is passive, or a false flag if the user is signalling, i.e., signalling mode is toggled.

Note that from the standpoint of the controller, each participant's cursor location is always either on the shared window or at home. However, at the local display, a third possibility exists, namely that the user's pointer may be on another window entirely. From the standpoint of remote users, the local user's pointer being on an another window is indistinguishable from it being at home.

## 6 FURTHER WORK

The work reported above provides the basic low-level communication and interaction mechanisms needed to support collaboration with interface agents. Further work is now concentrating on the content of such collaborations, using the single user and agent test-tube system shown in Figure 2.

### 6.1 Model-Based User Interfaces

Notice that in the system architecture described above, the agent is required to send requests to the controller, such as Move requests, that include geometric information about the screen layout of the application interface. For example, in order to press Button1, the agent needs to know the x-y coordinates of the button's location on the screen.

From a modularity point of view, it would be desirable to separate this kind detailed geometric information

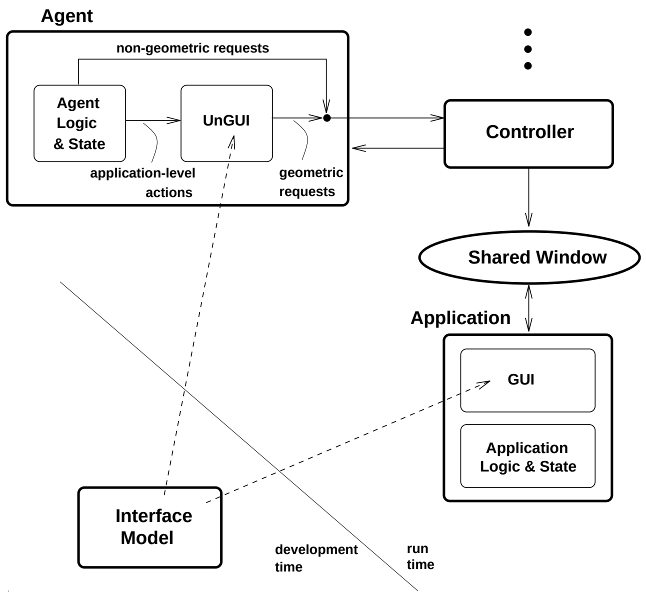


Figure 12: Using an interface model.

from the application-level logic of the agent's behavior. At a logical level, the agent should operate with primitive actions such as "Press Button1" or, in the case of the air-travel planning application, "make American be the preferred airline."

Figure 12 shows an approach to achieving this kind of modularity by partitioning both the application and the interface agent into a logical and a presentation component. For the application, the presentation component is the graphical user interface (GUI). In its purest form, this separation is what has come to called a "direct manipulation interface," i.e., one in which all aspects of the application state are viewable and modifiable through the GUI.

The agent is decomposed into its logical part and a module called an "UnGUI." Conceptually, a GUI maps (sequences of) window events into applications actions. An UnGUI performs the inverse mapping, i.e., given an application action, it computes a sequence of window events that achieve it. Said more mathematically, the composition of an UnGUI and the corresponding GUI compute the identity function.

In the prototypes described here, the UnGUI module was hand-coded. However, in the long run, a better approach is to use the model-based user interface paradigm [10], in which an explicit, largely declarative representation, called the *interface model*, is developed which describes the relationship between the application semantics and its interface appearance. It should then be possible to automatically generate both the GUI and the UnGUI for a given system from this model. Further research needs to be done on the appropriate representations for the interface model and algorithms for the automatic generation [2].

### 6.2 Collaboration and Discourse Structure

The larger agenda underlying this work is to apply principles of human collaboration [4] and discourse structure [3] to human-computer interaction using the interface agent paradigm. These principles include:

- dividing the steps of an interaction into *segments* based on their purposes
- modelling how the purposes of segments relate to each other and to the overall goals of the collaboration
- establishing mutual beliefs (between the user and agent) about the division of labor and about their shared artifacts
- negotiation of mutual beliefs

These principles have been validated across a wide range of tasks involving human collaboration in natural language. Because they address the structure of collaboration at the information-theoretic level, I believe they apply equally well to human-computer collaboration using an restricted artificial language [8] instead of natural language for communication.

## REFERENCES

1. Bormann, C., and Hoffmann, G. Xmc and Xy — Scalable window sharing and mobility. In *Proc. 8th Annual X Technical Conf.* (Boston, MA, Jan. 1994).

2. Edwards, W., and Mynatt, E. An architecture for transforming graphical interfaces. In *Proc. ACM Symposium on User Interface Software and Technology* (Marina del Rey, CA, Nov. 1994), pp. 39–48.

3. Grosz, B. J., and Sidner, C. L. Attention, intentions, and the structure of discourse. *Computational Linguistics 12*, 3 (1986), 175–204.

4. Grosz, B. J., and Sidner, C. L. Plans for discourse. In *Intentions and Communication*, P. R. Cohen, J. L. Morgan, and M. E. Pollack, Eds. MIT Press, Cambridge, MA, 1990, ch. 20, pp. 417–444.

5. Maes, P. Agents that reduce work and information overload. *Comm. ACM 37*, 17 (July 1994), 30–40. Special Issue on Intelligent Agents.

6. Meyers et al, B. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer 23*, 11 (Nov. 1990), 71–85.

7. Rich, C. Negotiation in collaborative activity: An implementation experiment. *Knowledge-Based Systems 7*, 4 (Dec. 1994), 268–270.

8. Sidner, C. L. An artificial discourse language for collaborative negotiation. In *Proc. 12th National Conf. on Artificial Intelligence* (Seattle, WA, Aug. 1994).

9. Sidner, C. L. Negotiation in collaborative activity: A discourse analysis. *Knowledge-Based Systems 7*, 4 (Dec. 1994), 265–267.

10. Sukaviriya, N., and Kovacevic, S. Model-based user interfaces: What are they and why should we care? In *Proc. ACM Symposium on User Interface Software and Technology* (Marina del Rey, CA, Nov. 1994), pp. 133–135.