Architectural support for cooperative multi-user interfaces

Richard Bentley, Tom Rodden, Pete Sawyer, Ian Sommerville

Computing Department
Lancaster University
Lancaster LA1 4YR
UK

Tel: +44 524 65201 x3119
Fax: +44 524 593608
Email: dik@comp.lancs.ac.uk

Introduction

Computer support for cooperative work requires the construction of applications which support interaction by multiple users. These applications exploit multi-user interfaces to promote their cooperative use by a community of users. Users may be distributed across a number of locations and the associated interfaces run across a number of workstations. The need to support user interface execution in a distributed environment has resulted in a merging of the concerns of user interface software and distributed systems. A major focus in CSCW is the development of models and tools to support the construction of these distributed, multi-user interfaces and the development of associated software architectures.

This paper is concerned with the design and development of a software architecture that provides mechanisms to support rapid multi-user interface construction and distributed user interface management. The developed architecture addresses the demands of both user interface systems and the constraints of the distributed infrastructure. A central concern is the need for rapid user interface prototyping resulting from the highly dynamic and flexible nature of cooperative work. Rapid prototyping requires mechanisms which make the information that determines interface configuration visible, accessible and tailorable.

In this paper, we examine the implications of rapid prototyping for distributed multi-user interface architectures. We also discuss the demands of multi-user interface management, and the problems of meeting these in a distributed environment. Finally we present a multi-user interface architecture which has been designed to address these problems, and demonstrate the architecture's suitability by discussing its implementation in a multi-user interface prototyping environment.

The multi-user interface architecture reported here has been developed as part of a project investigating support for the cooperative work of air traffic controllers. The project has made extensive use of prolonged ethnographic investigation to uncover the nature of cooperation in air traffic control (interested readers are referred to [1] for a full discussion

of the empirical findings). The aim of the architecture is to support an environment which allows a multi-disciplinary team to experiment with a wide range of alternate user interface designs for air traffic controllers. Examples used to illustrate the architecture in this paper are drawn from this domain.

Multi-user interfaces

Many computer systems already exist which support simultaneous interaction by more than one user. Multi-user operating systems, office information systems and databases are all examples of mature software technology which support multiple users. Although these systems support multi-user interaction they do so in manner which prohibits cooperation. Most existing multi-user systems present the illusion to each user that they are the only user of the system. The result is that users are unaware of the presence of others as a 'protective wall' is maintained around them hiding other users' activities.

In contrast, cooperative applications need to provide users with an awareness of the activities of others to support and encourage cooperation to take place. For these applications, the purpose of a multi-user interface is to establish and maintain a common context. This context allows the activities of one user to be reflected on other users screens and is supported by sharing application information. This sharing is the principle means of promoting cooperation, and the real-time presentation and manipulation of shared information is the main function of cooperative, multi-user interfaces.

Different forms of cooperative work require varying levels of awareness between users and place different demands on the strength of sharing required. The extent to which multi-user interfaces support this sharing through the propagation of activities is termed *interface coupling*; the greater the level of awareness between users the closer the interface coupling. Three levels of sharing can be identified in current cooperative systems which correspond to different degrees of user interface coupling:

- *Presentation level sharing* (Tight coupling): Each user is presented with the same display of the same subset of a common information space. When this presentation is changed in any way, presentations on all display screens are updated. This level of sharing is also known as What-you-see-is-what-I-see (WYSIWIS).
- View level sharing (Medium coupling): Each user has presentations of the same subset of an information space but the actual presentations may be different. For example, different users may simultaneously interact with tabular or graphical displays of the same data.

- Object level sharing (Loose coupling): Each user has presentations of different (possibly overlapping) subsets of the information space. For example, a group of users may each edit different sections of the same document.

Lauwers and Lantz [2] describe two approaches to developing multi-user interfaces. The first allows existing single-user applications to be shared in a *collaboration-transparent* manner, so that no facilities for handling collaboration are embedded in the application itself. The second involves the development of special purpose *collaboration-aware* applications which explicitly recognise the cooperation involved. Each of these approaches has different implications for sharing (see sidebar 1).

SIDEBAR 1: Collaboration transparent and aware user interfaces

To construct a collaboration-aware application, the developer must make decisions about how end-users require shared information to be presented and how they manipulate these representations. Although providing the flexibility to support different forms of presentation and interaction, these decisions are often embedded within the application itself. This fixes these design decisions within the application and they become amend a future date. These problems have led to a number of researchers arguing that although collaboration awareness is necessary for cooperative systems, it should not be embedded in the application. Rather, frameworks are required which manage the visualisation and manipulation of information outside the application itself.

Such frameworks hide the physical distribution of components from the application developer and allow the visualisation and interaction policies to be tailored independently of the application. This separation promotes rapid prototyping by significantly reducing the effort required to amend the interface. This separation is advocated in the purely centralised Rendezvous system [3], one of the best developed examples of such a CSCW framework.

Multi-user interface requirements

Multi-user interfaces need to provide a number of facilities. Before considering multi-user interface development and the features of supporting infrastructures it is worth reviewing two key end-user needs which constrain the development of the supporting infrastructure.

Support for multiple displays

Cooperating users access shared information through individual workstations often supplemented by informal communication. Given the reliance of cooperating users on an awareness of the activities of others, a multi-user interface architecture should:

- Allow the set of cooperative displays to be dynamically changed with different workstations being easily added and removed.

- Allow the manipulation of shared information entities on different screens.
- Support the visualisation of shared entities across different users' screens.
- Support the propagation of interaction across users' screens.

The need for rapid prototyping within cooperative systems development is such that these facilities need to be provided through a set of robust and readily understood mechanisms which allow reconfiguration by both developers and users.

Support for different views

Cooperating users may require information to be presented in different ways corresponding to different levels of sharing. Where users must engage in tightly-coupled cooperative work, it may be necessary to share information at the presentation level. However, sharing is also required at both view and object levels to support the differing requirements of end-users and the tasks they undertake. Thus the architecture must be able to support sharing at presentation, view and object levels. As a result the supporting architecture should:-

- Allow the definition of different interactive representations of shared entities.
- Maintain these different representations as underlying entities change.
- Allow the updating of entities through interaction with these representations.

Multi-user interface systems are used within distributed environments to support simultaneous interaction by a number of end-users. Consequently, it is important that such systems are sensitive to the properties of the supporting distributed infrastructures. Consistency between displays of the shared information and the information itself must be maintained and mechanisms must be provided to handle the propagation of change.

Support for multi-user interface development

The task of constructing a user interface is complex and requires skills from a number of disciplines. Most existing approaches to developing cooperative systems have tended to consider user interface development as a task for application developers and have provided little support for user-centred development and interface tailoring. The manner in which interfaces are shared is often pre-determined and embedded within the application. The focus of cooperative interface development is on shared information so support should allow experimentation with different visualisation and interaction techniques. As a result, any architecture to support rapid prototyping requires details of the interaction and presentation to be made explicit and de-coupled from the application.

SIDEBAR 2: Representation in the user interface

As well as requiring the rapid refinement and modification of alternative interface designs, prototyping also involves the evaluation of the user interface in realistic settings. Results from such assessment feed back into the refinement process to guide future development (see sidebar 2). Mechanisms to execute prototype interface designs are therefore required in addition to high-level development tools which support interface construction. These should make visible the policies for visualising and interacting with shared information to allow high-level tailoring and rapid revision of user interface designs.

The principles supporting user interface design have emerged from considering a single end-user on a single machine, performing discrete and individual tasks. A corresponding set of accepted principles have yet to emerge for multi-user interface development. However, a number of key features must be supported to promote effective multi-user interface construction.

Separation

An accepted feature of single user interface architectures is the separation of user interface and application components. This arrangement has many advantages which include:

- *Portability:* the application may be made portable while the user interface is device dependent
- *Re-usability:* both user interface and application can be reused independently
- *Multiple interfaces:* the same application can be accessed from different user interfaces
- *Customisation:* the user interface can be tailored in isolation by both developer and user
- *Adaptability:* The design decisions embedded within applications can be identified and made accessible to allow ease of modification

Logical separation is desirable for single-user interfaces but the multi-user case requires such separation to support the alternative visualisations needed for view-level sharing [4]. In addition, physical separation provides a degree of fault tolerance; a user interface process may crash without crashing other user interface or application processes. Physical separation also allows execution of interface and application processes on different machines providing local feedback and supporting a high degree of user adaptation in a distributed setting.

Feedback

Most operations require *feedback* from the display in response to user actions. The form of this feedback may depend on the semantics of the underlying application. When the application and user interface components reside on different machines, the *feedback loop*

involves transmission over a network and it may therefore be hard to achieve acceptable response times.

Feedthrough

Besides providing rapid feedback, multi-user interfaces also need to support rapid *feedthrough*. Feedthrough means updating users' display screens in response to actions performed by other users working on different machines. The importance of this principle is dependent upon the granularity of the updates broadcast to other users. In *tightly-coupled* cooperative activities, such as group drawing, the process of creating an object with associated explanation and gesturing is very often as important than the resulting object itself [5]. For applications requiring such tight coupling the granularity of updates is very small and rapid update feedthrough is vital. For other, more *loosely-coupled* applications, it may be possible to significantly reduce the granularity of updates.

End-user tailoring

When data is shared, end-users may adopt different methods of working even when performing similar tasks. In such situations, the user interface designer cannot provide interface representations which are appropriate to all users in all task contexts. One solution is to allow users to tailor their interface to suit their own requirements.

These principles, in combination with the requirements of cooperative interfaces, have motivated our design of a novel architecture to support multi-user interfaces. A distinctive feature of these multi-user interfaces is that they exist within distributed environments. Consequently, it is important that any supporting architecture is also sensitive to the properties of the underlying distributed infrastructure.

The supporting infrastructure

The construction of multi-user interfaces requires the needs of the interface to be met in a manner which is sensitive to the properties of the supporting infrastructure. To implement multi-user interfaces requires underlying support to manage the interface in execution. This support must address problems of distributed computing, such as network delay, loading and latency, failure of both network and machines and data consistency. The basis for providing such support is a computational or programming architecture.

Architectures for multi-user interfaces originate from distributed systems research. It is possible to identify two architectural extremes of pure *centralisation* and *replication*, between which lie a continuum of hybrid arrangements. The technological factors of implementing cooperative systems are mainly independent of the architecture - shared window systems have been built on top of both replicated and centralised architectures.

However, the properties of the supporting architecture delimits many of the features of the cooperative interfaces they support.

Centralised architectures

In a pure centralised arrangement, also known as the *client-server* architecture, a central server program handles all user input events and display output events which are routed by way of local client programs. Local workstations act as graphical terminals and window servers. A variant is the *master-slave* architecture where one client is merged with the server and all other nodes run clients.

The primary advantage of the client-server approach is its simplicity; application and all data are held centrally simplifying access management and data consistency. Implementation is easier still when using a networked window system such as X Windows. This approach is used within existing shared window systems such as shared X [6] and is also widely adopted within computer conferencing systems [7]. It is relatively easy to support WYSIWIS (presentation level) sharing using a centralised architecture as the server can replicate display directives to all clients.

It is also possible to support view level sharing using the client-server approach. For example, the Rendezvous system, introduced above, is based on a client-server architecture with all user interaction and display management handled centrally by the server. Each user has an associated *view process* which interprets input events and display directives. In this way, visualisation and interaction are detached from the information being shared as each view process can interpret events and display directives differently, supporting alternative presentations of information.

The embedding of the sharing policy in the central server component orients tailoring towards the user interface developer. Rendezvous, for example, does not make the sharing policy visible and therefore does not support end-user tailoring. Providing support for end-users to tailor their own interface may be important if individual methods of working are to be supported. The centralised architecture is also vulnerable to failure of the central node (or the network connections to it), and delayed feedback as all input and output events must travel over a network.

Replicated architectures

At the other extreme to a centralised approach, a replicated architecture maintains exact copies or *replicas* of the application on each workstation. Each replica handles screen management and feedback locally and must broadcast any change in application data to all other replicas to maintain data and interface consistency. Local management of the display means that different views are easily supported and end-user interface tailoring is relatively

easy to provide. Each replica can adapt its visualisation and interaction policies to an end-user's preferences.

The major difficulties with replicated architectures concern synchronisation and data consistency. Users can perform actions simultaneously which are executed locally before being broadcast to other machines. If these actions conflict - for example one user deletes a selected object in a WYSIWIS group drawing program at the same time as a second user changes the selection to a different object - inconsistent interfaces can result due to events arriving in a different order at each machine.

To prevent such *race conditions* requires complex synchronisation algorithms. The standard solution in distributed systems is to use a global clock to timestamp each event and then to *rollback* should inconsistency arise, replaying events in temporal order. This is unacceptable for multi-user interfaces where display screens would already have been updated and alternatives based on transforming updates to prevent rollback have been developed [8].

A further problem occurs when users wish to join a group session after it has started. This *dynamic registration* is relatively straightforward under a centralised approach as new clients need only contact the central server. The server can then broadcast the current state of the application to bring the new client up to date. Using a replicated approach, however, a new replica must contact all other replicas to tell them that it needs to receive any updates. This means that new replicas must know or can find out the locations of all other current replicas.

Hybrid architectures

Both pure centralised and pure replicated architectures offer benefits and limitations. As neither of these architectures fully meet the requirements of multi-user interface systems a hybrid solution is required where appropriate portions of the system are either centralised or replicated depending on the application requirements. A continuum of such hybrid arrangements exists between the extremes of pure centralisation and replication. Each of these supports the requirements for multi-user interfaces in different ways and with different degrees of complexity.

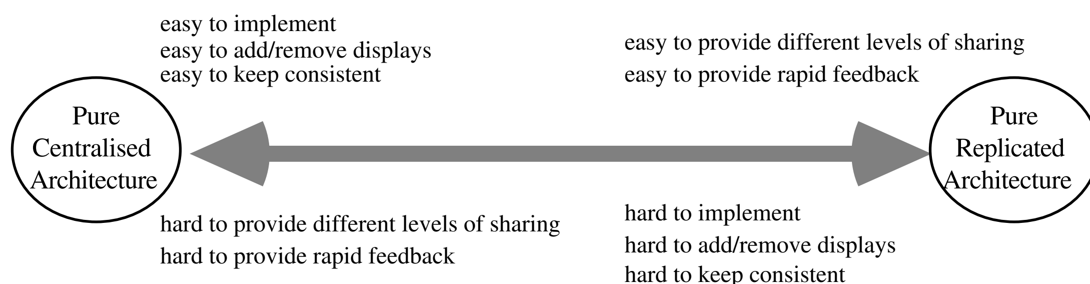

Figure 1 Summary of characteristics of distributed computational architectures

Our objective was to develop an infrastructure which meets the needs of multi-user interfaces in a such a way that effective run-time support is provided along with facilities to promote tailorability. Our architecture is based on autonomous agents that embody the details of information visualisation and interaction necessary to manage multi-user interfaces independently of the behavioural semantics of the application. The architecture is a hybrid one with shared information kept consistent through a centralised component while the presentation and interaction semantics are replicated and held in distributed display agents.

User Display Agents

Our architecture considers individual user's information displays as autonomous entities with properties that can be tailored by both interface developers and users. The states of these entities characterise the way information is presented to users, who interact directly with these entities to update the underlying information. Such updates are immediately propagated to other users' displays to maintain consistency. We refer to these entities as User Display agents and the parts of a user's screen managed by an agent is called a User Display (UD).

In our cooperative setting, users browsing and manipulating a shared information space each hold a *working set* of UD agents. Each agent manages one display of the shared information and can present this UD in multiple screen windows (figure 2). An agent can be a member of a number of working sets and the UD it manages can therefore be displayed on multiple users' screens. Agents can be added to and removed from working sets as required.

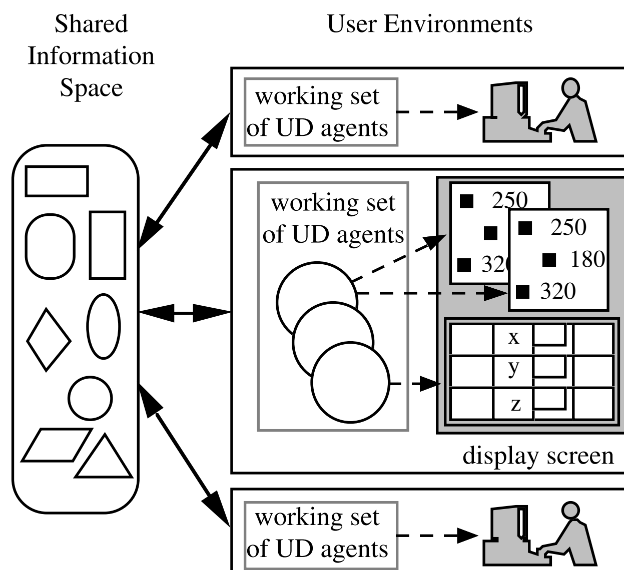

Figure 2 Working sets of User Display agents cooperating via a shared information space

This arrangement supports all three levels of interface coupling identified previously. As agents can display the UDs they manage on multiple screens, it is possible to support presentation-level or WYSIWIS information sharing. Different UDs can be designed which represent the same information in different ways to support sharing at the view-level. Sharing at the object-level is possible as users can have completely different UDs managed by the UD agents in their working sets.

This mechanism separates *what is being shared* (the application information) from the *method of sharing* (the presentation and means of interaction with the information). The method of sharing can therefore be tailored independently of the application, as UD agents support collaboration-aware sharing without requiring the application to handle the sharing mechanisms itself. The rest of this paper discusses the properties of UDs and the UD agent architecture to show how rapid user interface construction and distributed management are supported.

Properties of a User Display

User Displays support the sharing of information between multiple users and allow users to update information through interaction with local representations. The need to support different levels of sharing and degrees of interface coupling highlights three effects of changes in the state of shared information:

- *FOCUS:* Cooperating users may only be interested in a subset of the information entities from the shared information space. Entity representations may therefore need to be added to or removed from users' displays dynamically.
- *REPRESENTATION:* Whilst cooperating users may require the same information, the way this is represented may vary depending on the tasks being supported and the level of experience and domain knowledge of the user.
- *POSITION:* The spatial arrangement of entity representations on the user's screen may provide information about relationships between entities. It may be necessary to adjust the arrangement dynamically to reflect changes in these relationships.

An example of a UD is an interactive Radar display that presents the geographical position of aircraft in a bounded portion of airspace on a two-dimensional display screen. Aircraft's longitude and latitude data are mapped onto the x,y position of a 'blip' representing each aircraft's location on the screen. These blips can contain information such as identification codes and height information, and can be manipulated by the user to display additional information about the aircraft they represent.

In terms of *focus*, *visualisation* and *position*, the process of modelling the Radar is quite straightforward. The focus is concerned with the Radar's calibration (i.e. the portion of airspace represented), the visualisation with the type of blip used to represent each aircraft

and the position with the mapping from aircraft location to the position of corresponding blips on the screen. Any change in location may cause changes in focus, visualisation and/or position.

Components of a User Display

To reflect the emphasis on the display and manipulation of shared information entities, the model of a UD directly realises the concepts of focus, representation and position. A UD is described as a triple, comprising a *Selection*, a *Presentation* and a *Composition*:

- A *Selection* is a set of information entities dynamically chosen from an information space according to *Selection criteria.* Selection criteria are predicates over entity *attributes* and are specified by the interface designer. Selection criteria act as a filter to pick out those entities which should be displayed. As the state of entities is updated, the Selection changes accordingly.
- A *Presentation* is a set of *Views* used to represent entities in the Selection. A View is a graphical representation which defines the appearance of a single entity, the position and representation of that entities attributes and the means of interaction by which the user can update the entity's state. Views are dynamically selected for each entity through the application of the *Presentation Criteria.* These define a filter for each View which an entity must pass to be represented by that View. Changes in the state of an entity may require changes in the Presentation, i.e. the selection of a different View to display it on the screen.
- A *Composition* is a set of positions which represent the spatial arrangement of Views in the UD. These positions can be either absolute or relative to other Views. As an entity's state changes, these positions may also have to change to remain consistent with the arrangement defined in the *Composition Criteria.*

Using this model, one abstract definition of the Radar UD described above might be:

- *Selection criteria*: Aircraft longitude from x' to x'', latitude from y' to y''
- *Presentation criteria*: *large blip* for passenger aircraft, *small blip* for private aircraft
- *Composition criteria*: Map longitude to x position, latitude to y position

The abstraction of UDs as autonomous entities allows multi-user interfaces to be constructed as a federation of such entities, interacting and responding to changes in a shared information space. Changes in the state of information entities potentially effects the Selection, Presentation and Composition components of each UD. For example, a change in the longitude of an aircraft may require a change in Selection or Composition of the Radar UD described above.

The encapsulation of both the definition criteria and the state of an information display in an autonomous UD entity allows the tailoring of information displays without system re-configuration. Any changes to definition criteria of a UD result in immediate computation of the new state and the update of users' screens. This tailoring, re-computation and display management is managed by the associated UD agent.

Maintaining User Displays

A UD agent can be a member of more than one users' working set to support presentation-level or WYSIWIS information sharing. Each screen representation of the UD managed by such an agent is affected by updates in the shared information space in exactly the same way, and thus the effects of information updates on Selection, Presentation and Composition need only be calculated once.

To support this each user may hold a copy or *Surrogate* of each UD agent as part of their working sets. These Surrogates are minimal agents that hold only the current state of the Selection, Presentation and Composition of the UD. The definition of the UD is held by the *Master* UD agent which receives notification of relevant changes in the states of information entities, uses the definition criteria to compute the effects on the UD and informs each of its Surrogates of the new state (figure 3).

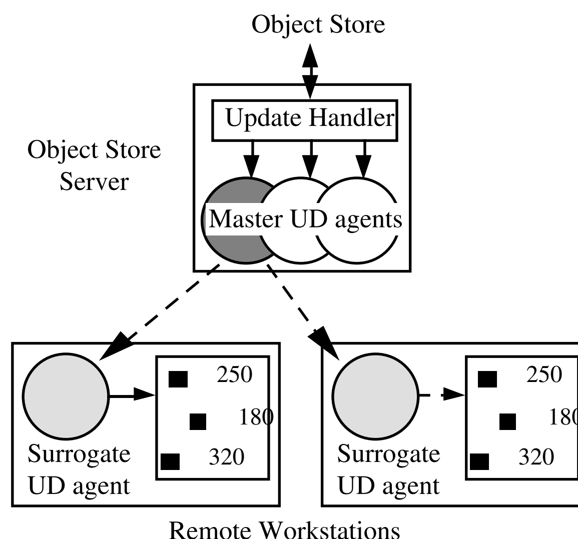

Figure 3 Master/Surrogate User Display agent arrangement

In our realisation of the agent architecture, shared information entities are held as objects within an Object Store. All updates to the state of objects in the Object Store are handled by the *Object Store Server* (OSS), illustrated in figure 3. This component also holds the Master UD agents, allowing straightforward registration and de-registration of new users. Machines can be added without other users' machines being informed, and need only contact the OSS to register their existence, create the required Surrogates and establish

links to the Master agents. A machine can de-register by informing the OSS that it no longer wishes to receive update information.

The Master/Surrogate arrangement of UD agents allows local tailoring of information displays without requiring system re-configuration. Local tailoring operations performed on a window presenting a UD are retained by the Surrogate agent. When informed of updates to the state of the UD by the Master, Surrogates can take into account these operations before displaying the new state.

Consistency maintenance

Updates to shared information objects potentially affect the Selection, Presentation and Composition components of each UD. For example, consider the definition of the Radar UD described above; a change in an aircraft's position (latitude or longitude) may require it to be added to or removed from the display or its blip representation to move. UD agents must be aware of such updates so they can use the definition criteria to calculate effects on the state of their UDs and maintain consistency with the shared information space.

There are two options for detecting changes in the state of information entities; agents can periodically poll the information space, checking to see if it has changed, or information objects can notify agents when changes occur. The former is inefficient when there are a large number of information objects to be polled, whilst the latter requires each information object either to record which agents are interested in it and be able to notify them whenever it is updated, or to broadcast updates to all agents, many of which may not be effected by the update.

The agent architecture uses a variant of the second option which does not place any requirements on the information objects to notify UD agents of changes in state, and therefore supports collaboration-transparency of the application being shared. All updates to shared information objects are delivered to a component within the OSS known as the *Update Handler*. This forwards these updates to the Object Store, as well as notifying the UD agents potentially effected. Figure 4 illustrates this *dispatching* role of the Update Handler, which ensures that only those UD agents interested in updates are actually notified of them. Using this mechanism, the communication (and thus the cost) of informing UD agents of updates is minimised.

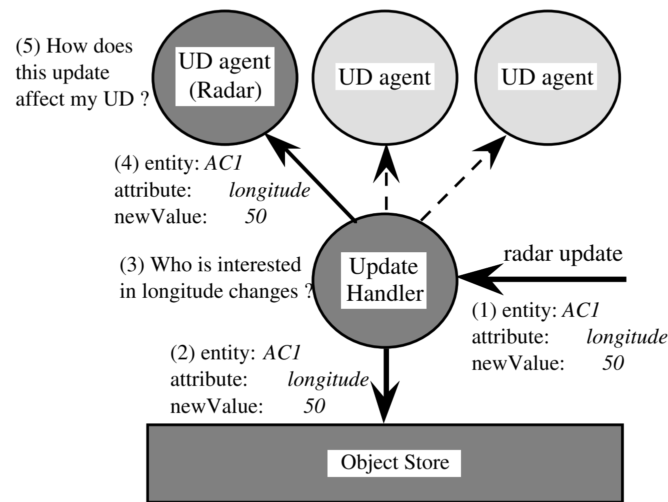

Figure 4 The dispatching role of the Update Handler

Whenever the user interface developer creates a new UD, a UD agent is automatically created to manage it. The first task of this agent is to register with the Update Handler in order to receive update information. As part of the registration process, agents must inform the Update Handler of their *interest set* which it then uses to determine which updates should be forwarded to the agent; for example, the Radar UD agent defined above will register changes in longitude and latitude of aircraft objects as its interest set. Agents may have to re-register their interest set should the interface developer modify the definition criteria of their UDs, and de-register if their UDs are removed (prior to the agent itself being destroyed).

This mechanism ensures that agents can maintain the consistency of their UDs with the shared information whilst not being overloaded with irrelevant update notifications. The filtering of updates by the Update Handler thus minimises the communication overhead (and therefore the cost) of notification and allows applications to remain unaware of the agents that manage representations of their data, supporting collaboration-transparency.

Supporting multiple Views

Shared information objects can be represented in different UDs by different Views. As updates occur, these Views must also be updated to remain consistent with the entities they represent. To maintain this consistency, Views are linked to the objects they represent for each UD, as in figure 5. These links allow update information to flow between Views and shared information objects when the end-user modifies a View, and in the opposite direction when objects are changed through other end-users' interactions or by external updates. As shown in figure 5, each object in a UD's Selection may have links between itself and more than one View.

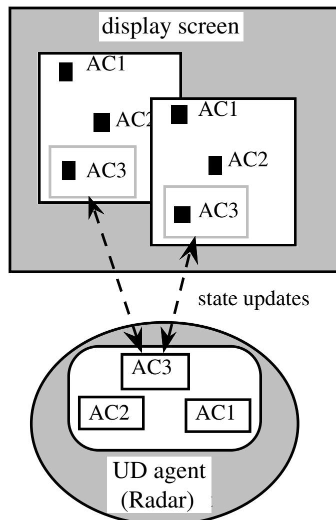

Figure 5 Links between Views and shared information objects

To minimise the communication between OSS and remote machines, the agent architecture adopts a local caching mechanism where subsets of the shared information objects maintained in the Object Store are *cloned* and held in a local cache (figure 6). Updates to shared information objects are not routed through the UD agents but are sent to the caches by the Update Handler, which maintains a table of the locations of all object clones. The links between the Update Handler and the local caches are bi-directional to allow the Update Handler to receive updates in the states of clones resulting from end-user interaction with object Views.

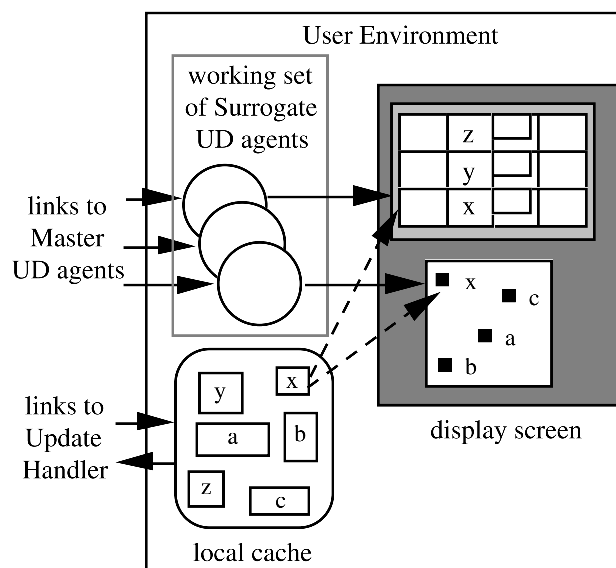

Figure 6 Local caching mechanism

Views are therefore linked directly to clones of shared information objects. Should the state of a clone be updated by end-user interaction with a View, the local cache informs the Update Handler of the change. The Update Handler can then notify the cooperative application of the update, selectively send updates to caches which maintain a clone of the

affected object (and thus update all the object's Views), and inform the relevant Master UD agents in case the update requires changes in the states of any UDs.

The local caching mechanism eliminates needless communication between the OSS and remote machines, as well as removing the need for mechanisms to maintain object-View links across machines. By recording the location of object clones the Update Handler is able to ensure consistency of local caches whilst filtering update information.

The UD model and agent architecture described here form the basis for a prototyping environment called MEAD (see sidebar 3). The architecture is flexible enough to allow View and UD definitions to be added, removed and modified in the OSS whilst Slave machines are connected, without requiring suspension of user activities. The agent architecture can therefore support rapid modification and execution of alternative user interface designs necessary to meet the requirements of rapid prototyping.

SIDEBAR 3: The MEAD multi-user interface prototyping environment

Summary and conclusions

The development of multi-user interfaces is a central concern to CSCW. Our limited knowledge of the nature of group work and the lack of proven interface principles ensures that rapid prototyping is essential to ensure the development of effective user interfaces. Prototyping of this form demands a robust a set of mechanisms to support user interface definition which can be readily made available to developers and users alike.

This paper has presented an architecture which supports multi-user interfaces and allows the rapid construction of user interfaces to support prototyping as a means of user interface development. The hybrid architecture is based around the concepts of User Displays which are managed by autonomous User Display agents. User Displays are defined in terms of the relationship between different presentations of shared objects. This definition can be presented to both users and developers and dynamically altered at run time to allow the construction of different multi-user interfaces.

The developed architecture provides the basis for the MEAD rapid prototyping environment for multi-user interfaces. The environment has been constructed as part of a project to support the development of user interface displays for air traffic controllers. The environment exploits the natural centralisation of cooperative applications within a hybrid architecture which masks many of the complexities of the underlying distributed systems. In contrast to the properties of distributed systems, the developer is presented with a set of simple mechanisms which determine the properties of the multi-user interface. These mechanisms provide the basis for the construction of a wide range of cooperative multi-user interfaces.

References

[1] Bentley, R., Rodden, T., Sawyer, P., Sommerville, I., Hughes, J., Randall, D., Shapiro, D., Ethnographically informed systems design for Air Traffic Control, Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW'92), October 31-November 4, 1992, Toronto, Canada, ACM Press, pp 123-130

[2] Lauwers, J. C., Lantz, K. A., Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems, Proceedings of CHI '90, April 1-5, 1990, Seattle, Washington, ACM Press, pp 303-311

[3] Patterson, J. F., Hill, R. D., Rohall, S. L., Meeks, W. S., Rendezvous: an architecture for synchronous multi-user applications, Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '90), October 7-10, 1990, Los Angeles, California, ACM Press, pp 317-328

[4] Patterson, J. F., Comparing the programming demands of single-user and multi-user applications, Proceedings of the Conference on User Interface Software Technology (UIST'91), November 11-13, 1991, Hilton Head, South Carolina, ACM Press, pp 87-94

[5] Tang, J., Findings from observational studies of collaborative work, International Journal of Man-Machines Studies, 3(4), pp 143-160

[6] Gust, P., Shared X: X in a distributed group work environment, presented at the 2nd Annual X conference, MIT, Boston, January 1988

[7] Crowley, T., Milazzo, P., Baker, E., Forsdick, H., Tomlinson, R., MMConf: an infrastructure for building shared multimedia applications, Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '90), October 7-10, 1990, Los Angeles, California, ACM Press, pp 329-342

[8] Ellis, C. A.,Gibbs, S.J., Concurrency control in groupware systems, Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, 1989, Portland, Oregon, ACM Press, pp 399-407

SIDEBAR 1 : Collaboration transparent and aware user interfaces

Whilst a range of different multi-user interfaces have been developed, each providing a variety of functions and different levels of support for cooperative work, two broad classes of multi-user interface are important. The first approach allows existing single-user applications to be used cooperatively by a number of users. The second involves the development of special purpose applications which handle collaboration explicitly. Lauwers and Lantz [1] identify these approaches as *collaboration transparency* and *collaboration awareness* respectively.

Many applications exist which can easily be modified to run in a multi-user setting. Allowing sharing of these applications provides access for geographically dispersed participants to sophisticated tools to facilitate group work. This observation has led to the development of systems which support the *transparent* sharing of applications, often called *shared applications*.

These approaches were developed to allow the sharing of each user's screen with others. As windowing systems have developed, this *shared screen* approach has been extended to permit the sharing of individual windows. The logical structure of such a *shared window system* is shown in Figure 1.

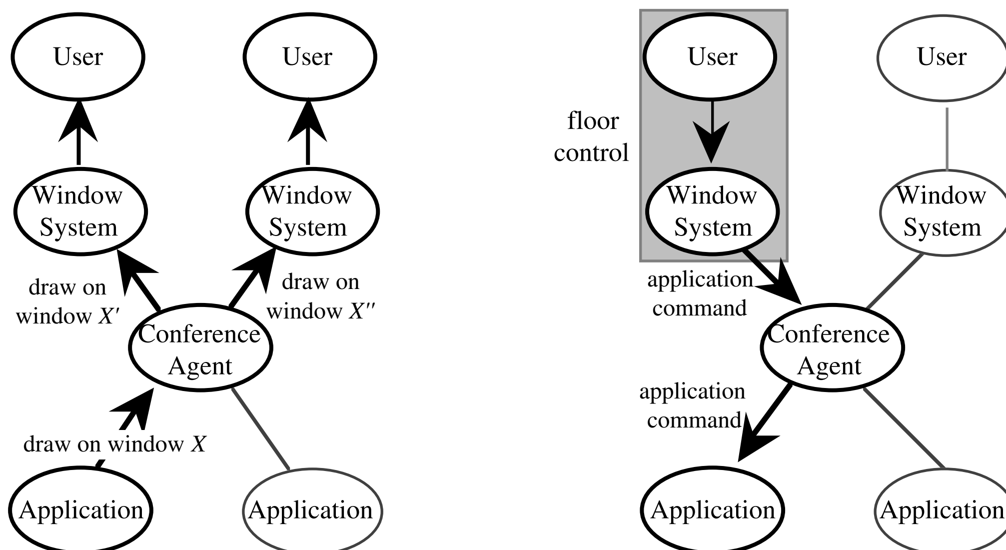

Figure 1 Logical a) output and b) input architecture of a shared window system

A central component or *conference agent* is responsible for multiplexing display output and de-multiplexing user input so that the application deals with a single stream of output and input events. The application is unaware of the presence of more than one user and only expects a single stream of input events. To avoid confusion users must take turns in interacting with the application - this is achieved by allowing only one user to interact with the application at any given time (as in figure 1b). Borrowing from the terminology of

business meetings, this user is said to have *control of the floor*. Control of the floor must be passed to other users before they can interact with the application.

Floor control is the responsibility of the central conference agent which Greenberg identifies as its *chair management role* [2]. Unsurprisingly, the focus of much of the work in supporting collaboration transparency has focused on floor control, such as the development of different turn-taking protocols and floor control policies.

In contrast to the transparent approach outlined above, collaboration aware solutions provide facilities to explicitly manage the sharing of information, allowing sharing to be presented in a variety of different ways to different users. Often the management of each user's sharing is embedded in the application itself. Applications shared in this way are referred to as *multi-user applications*.

The logical centralisation of user interface management embeds a set of decisions as to how information is presented and modified within the application. This definition is often inaccessible and this lack of visibility inhibits tailoring of the *sharing policy*. In addition, the lack of a supporting infrastructure requires most collaboration aware applications to be constructed from scratch. As a result, this approach to developing cooperative systems has tended to be less popular than the collaboration transparent approach outlined above. However, the flexibility needed by cooperative systems is such that collaboration aware arrangements are becoming more prominent.

The multi-user interfaces supported

The problems of developing multi-user interfaces were initially highlighted in the COLAB project at Xerox PARC [3], which examined the development of appropriate supporting facilities for real-time co-located meetings. As part of the project a number of applications were developed which allowed the effects of each user's actions to be shared across a number of displays. Developing on desktop publishing terminology, the COLAB project called this principle What You See Is What I See (WYSIWIS) and saw it as analogous to the use of WYSIWYG in editor systems.

Initially each user's complete display screen was shared, however this was found to be confusing and distracting for the users involved. To resolve this only portions of the display were shared and a separation was established between shared and private portions of users' displays. This arrangement is referred to as *relaxed WYSIWIS* while the initial COLAB setting is termed *strict WYSIWIS*.

Collaboration transparent systems replicate display output and adopt an approach based on sharing the *presentation* of an application. Multi-user interfaces for such systems therefore support the WYSIWIS sharing of applications, which gives each user a 'common frame of

reference' [1]. The removal of this common context can cause problems when users engage in *tightly-coupled* group work.

However, although some shared window systems relax the coupling of the interface, a fundamental problem with such systems is their inability to support anything other than WYSIWIS sharing of the information itself. Where end-users have widely differing knowledge, roles and attributes, this can be overly restrictive. A view used by a technical person, for example, may have too much detail for effective viewing by (say) a manager [2].

Where different users engaged in different tasks must access and manipulate shared data, the development of multi-user interfaces cannot be based on purely WYSIWIS sharing. Different users may require different representations of the same information, or may be concerned with different information entirely. This level of flexibility cannot be provided by collaboration transparent applications as it requires the system to exploit knowledge concerning the shared task been undertaken by the users. This form of interface is provided by collaboration aware applications.

Although they can and often have supported strict WYSIWIS interaction (as with the COLAB system for example), the advantages offered by collaboration aware systems are most evident in the provision of a range of alternative application presentations across a community of users. To contrast this arrangement with the work of COLAB, Dewan [4] terms this style of sharing What You See Is Not What I See or WYSINWIS.

[1] Lauwers, J. C., Lantz, K. A., Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems, *Proceedings of CHI '90*, April 1-5, 1990, Seattle, Washington, ACM Press, pp 303-311

[2] Greenberg, S., Sharing views and interactions with single-user applications, *Proceedings of the ACM/IEEE Conference on Office Information Systems*, 1990, Cambridge, Mass., ACM Press, pp 227-237

[3] Stefik, M., Bobrow, D., Foster, G., Lanning, S., Tatar, D., WYSIWIS revised: early experiences with multi-user interfaces, *ACM Transactions on Office Information Systems*, 5(2), pp 147-167, 1987

[4] Dewan, P., Principles of designing multi-user user interface development environments, *Proceedings of the 5th IFIP Working Conference. on Engineering for HCI*, J. Larson and C. Unger (Eds), August 10-14, 1992, Ellivuori, Finland, pp 35-48

SIDEBAR 2: Representation in the user interface

In designing a user interface for an application the interface designer(s) must provide representations of application entities. These must support the user's task, allowing information to be displayed, collated and manipulated effectively. The problems of designing representations are compounded where entities are updated in real-time and are potentially shared, with different representations being used by different users to support a variety of tasks.

Two general approaches to the design of effective representations can be identified. First an approach based upon determining and encoding the various principles which underlie their effective use allows representations to be generated automatically. The alternative is an iterative approach whereby initial representation designs are refined on the basis of user feedback generated in evaluation trials. Other approaches are hybrids of these two, such as allowing direct manipulation tailoring of automatically generated designs.

The most effective representation to use for a set of entities will obviously depend on the entities themselves; for example, a pie chart of a data set will not be effective if that data set comprises several thousand categories, each requiring a slice of the pie. Many automatic generation systems, such as Mackinlay's definitive APT system [1], use an analysis of the data to be presented to generate the representations.

Besides the data itself, it is equally important, however, to take into account what the representations will be used for and who they will be used by. The effectiveness of a particular representation is therefore dependent on the task(s) being supported and the level of expertise/domain knowledge of the users, as well as the data itself. The earlier work of Mackinlay has been extended in systems such as Casner's BOZ [2], which requires a formal description of the user's task to generate a representation.

The alternative approach of iterative development does not attempt to formally model the data, tasks or users. Rather, evaluation of prototype designs by end-users engaged in real tasks allows iteration towards task- and user-specific interface representations. Such an approach, as illustrated in figure 1, has long been advocated for developing systems where understanding of the problem is poor, as in user interface design.

Representation in cooperative systems

There has been much research in the HCI community on the formal modelling of both tasks and users. Such a description is required if automatic generation approaches to representation design are to be effective. The description of individual tasks in a formal and complete manner is complex; decomposing a work activity into task units is greatly complicated when work is performed in groups rather than individually.

The process of task allocation amongst group members can be very flexible, based not on individual status or prescribed responsibilities but on levels of busy-ness and current situation. This flexibility is often responsible for the efficient manner in which work is carried out; consider for example the effects on productivity of 'working to rule' in industrial disputes. In cooperative environments, the cooperative structure changes dynamically to match changes in the current situation.

It is not therefore clear how a task performed by a group can be formally represented. In addition, the prescriptive nature of automatic approaches prevents alternative and radical interface designs being produced. For cooperative systems, rapid prototyping of interface representations is needed to determine the effectiveness of an interface design.

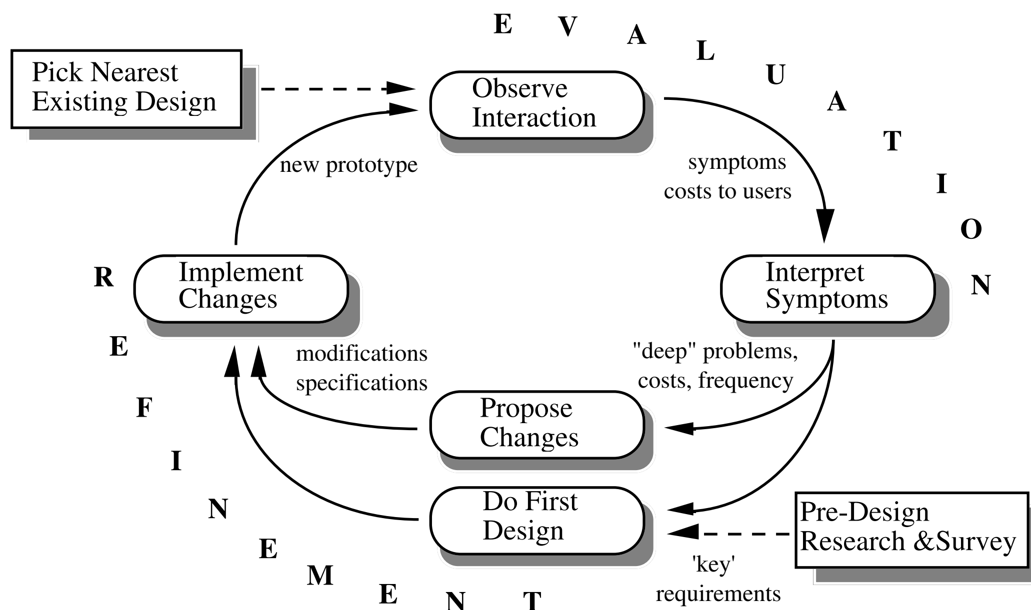

Figure 1 The user interface prototyping cycle (based on [3]: figure 1)

In addition to utilising qualitative rather than quantitative data, an approach based on rapid prototyping has the advantage that a range of different designs can be developed and evaluated. Some of these designs may initially seem unsuitable given current understanding of what makes a user interface effective, but evaluation trials may demonstrate their suitability for end-users who have extensive knowledge of their domains of expertise.

[1] MacKinlay, J., Automating the design of graphical presentations of relational information, *ACM Transactions on Graphics.*, 5(2), 1986, pp 110-141

[2] Casner, S., A task-analytic approach to the automated design of graphic presentations, *ACM Transactions on Graphics.*, 10(2), 1991, pp 111-151

[3] Draper, S., Watley, K., *Practical Methods for Measuring the Performance of Human-Computer Interfaces*, notes accompanying talk given at the JCI HCI Summer School, Queen Mary College, August 1991

SIDEBAR 3: The MEAD multi-user interface prototyping environment

MEAD† is a prototyping environment which allows the construction and refinement of cooperative displays. It makes visible the model of a User Display (UD) discussed in the main text, whilst hiding the complexity of the Update Handler, Master/Surrogate and local caching mechanisms. The tools MEAD provides are shown in plate 1†† which illustrates the definition and realisation of a Radar UD.

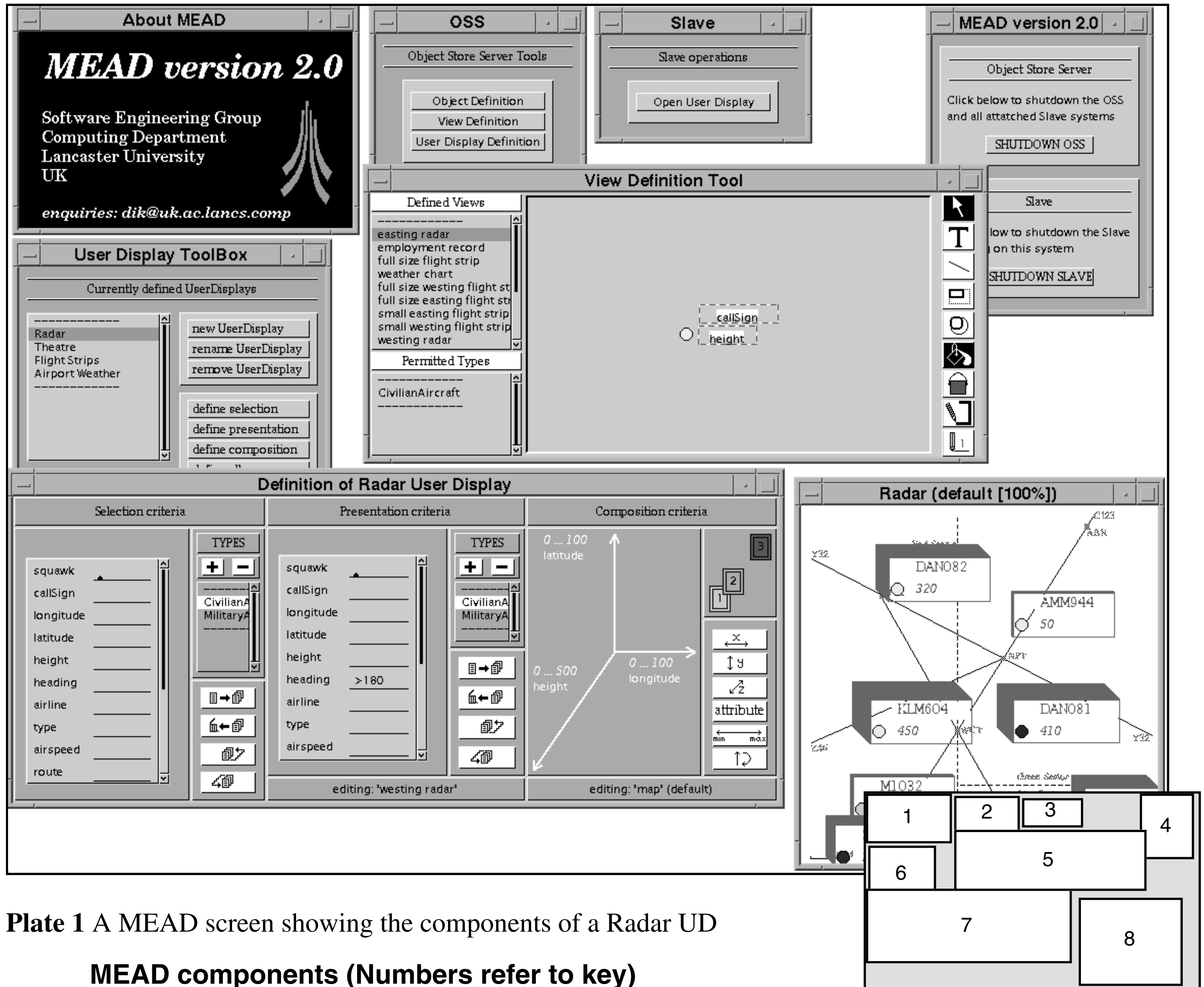

Plate 1 A MEAD screen showing the components of a Radar UD

MEAD components (Numbers refer to key)

1. MEAD information window

2. Object Store Server tools

From this toolbox, the user interface developer can open tools on the object store (not illustrated), the View definitions (5) and the Master UD agents (6). These definitions are all held in a central component called the Object Store Server (OSS).

---

† MEAD version 2.0 is developed in Objectworks\Smalltalk r4.1 for Sun workstations

†† Note to editor: A colour version of this plate is available

3. Slave toolbox

To open a Window such as (8) which presents a UD, a Slave must be started which registers itself with the OSS described above. The Slave toolbox is then accessible which allows UDs to be opened on the local workstation.

4. MEAD launcher

OSS and Slave modules can be started and shutdown using this panel. Typically, one machine will run the OSS in any one session with Slaves being started on a number of machines (which can include the machine running the OSS).

5. View definition tool

This supports the definition of different entity representations called *Views*. Using a set of primitives, the user interface developer constructs the representation, indicating how entity attribute values are to be positioned, how they are to be presented and how the user can interact with them to change the state of the represented entity.

6. User Display browser

UDs can be created, re-named and removed using this tool. In addition, definition tools can be opened on the Selection, Presentation and Composition criteria components of the selected UD (7).

7. User Display definition tools

These capture the definition of the Selection, Presentation and Composition criteria for a UD. Each set of criteria are created and modified using a separate 'form'. The *Selection* and *Presentation* criteria are specified by editing a condition template. A single entity must pass all the conditions on a template to pass the *guard* which the template defines.

Composition Axes are defined to specify the layout of the representations in the UD. This tool allows a number of different arrangements to be defined to allow the user to change layouts, as well as supporting the association of different backdrops, depth effects etc. The layout shown here is a 3-D arrangement, with aircraft longitude and latitude mapped on to x,y position and height being used to calculate the depth.

8. User Display window

This presents the UD defined in (7). Aircraft entities are taken from a small example information store created from actual flight plan data. The attribute values shown in each View can be edited to update the underlying entities, with all changes propagated to all other representations on all Slaves. In addition, the user can change the View representation being used for each entity and can change the layout they wish to use if alternatives have been defined). Any changes made to the View or UD definitions are immediately propagated to all open Windows which present effected UDs.