SOL: A Shared Object Toolkit For Cooperative Interfaces

Gareth Smith and Tom Rodden

Cooperative Systems Engineering Group

Technical Report : CSEG/7/1995

CSEG, Computing Department, Lancaster University, LANCASTER, LA1 4YR, UK.
Phone: +44-524-593041; Fax: +44-524-593608; E-Mail: cseg-info@comp.lancs.ac.uk

SOL: A shared object toolkit for cooperative interfaces

GARETH SMITH AND TOM RODDEN

Computing Department, Lancaster University, Lancaster LA1 4YR, UK.

E-mail: {gbs, tam}@comp.lancs.ac.uk

The paper presents a user interface toolkit to support the construction of cooperative multi-user interfaces. The toolkit is based on the configuration of shared interface objects to construct cooperative interfaces. A principal focus of the toolkit is the provision of accessible facilities to manage interface configuration and tailoring. Most existing facilities to manage multi-user interfaces tend to be application specific and provide only limited tailorability for purpose built cooperative applications.

In addition, the current structure of most cooperative applications fails to separate the semantics of applications from the cooperation specific semantics. In this paper we present a multi-user interface toolkit that provides management facilities in a manner which separates appropriate features of cooperative use from application semantics. This is achieved by allowing multi-user interfaces to be derived from a common shared interface constructed from shared interface objects. We would suggest that the separation of semantics in this form represents an initial identification of the re-usable cooperative interface components.

KEYWORDS: Cooperative Interfaces, Interface Toolkits, Interface Coupling.

INTRODUCTION

The development of applications that support a number of interfaces across a community of users has played a prominent role in CSCW. Different researchers have investigated techniques and architectures to support the real time management of these interfaces. Research has focused on the development of facilities to coordinate and manage interaction across the different interfaces. In general, one of two alternative strategies have been adopted in the construction of multi-user applications, *collaboration transparent* or *collaboration aware* (Lauwers 1990).

Collaboration transparent systems focus on allowing the large body of existing single-user applications to be used by a group of users without modification. This offers significant advantages in migrating applications to a cooperative setting. However, this is achieved at the cost of limiting the amount of control users have in managing the cooperative properties of the interface. In contrast, collaboration aware applications provide extensive facilities to allow users to both configure and control interfaces. The cost of this increased management is the need for the cooperative applications to be directly responsible for the facilities provided. This incurs a significant development cost and facilities developed tend to be specific to an application with little consideration is given to providing management and tailoring facilities across a range of applications. This is somewhat in contrast to the dominant trend in single user interface development which focuses on the identification of generally applicable user interface components (or widgets) which can provide common interface facilities across a range of applications.

In this paper we present an approach to multi-user interface construction within a toolkit called SOL that allows the rapid construction of application interfaces while also providing facilities to manage the cooperative aspects of the interface. The approach we have adopted is to consider the different interfaces presented to members of the user community by a cooperative application as being projected from a common shared application interface. This allows the management of the cooperative features to be achieved by providing facilities that control the derivation of the projected interfaces (Smith 1993). As these cooperative facilities are external to detailed application behaviour we encourage the adoption of general facilities across a range of applications. We have realised SOL through a set of interface libraries, an associated run time environment and a user interface toolkit that augments the existing X-window system.

BACKGROUND AND MOTIVATION

Our aim is to provide a set of facilities that allow the realisation of effective cooperative interfaces. The need to develop systems which sit easily with the cooperative work of users ensures that an iterative model of development is essential. The provision of appropriate development and management facilities is central to supporting this form of development model. Support for iteration in the development process is closely associated with minimising the commitment involved in development of interfaces. We seek to reduce commitment by providing facilities which allow cooperative interfaces to be :

- Rapidly constructed at a initial low cost to developers and users;
- Readily changed and reconfigured at low cost and in such a manner that promotes direct feedback to the prototyping cycle.

In addition, these facilities need to be provided to developers in a manner which is sufficiently familiar to allow the principles of development and reconfiguration to be understood. Further, given the situated nature of cooperative work these facilities need to provide the ability to alter systems behaviour in "real-time" and to allow behaviour to be amended to respond to demands initially unanticipated by developers.

Central to our provision of interface development facilities is the provision of appropriate mechanisms for the management and control of multi-user interfaces. Our approach is strongly influenced by the need to find a compromise between what is currently viewed as two disparate approaches to the provision of multi-user interfaces. The development of special purpose applications that are *collaboration aware*; and the adapting of existing single user applications to provide *collaboration transparent* shared applications.

Collaboration transparency offers the significant advantage of allowing a wide range of already developed single-user applications to be used without modification. The majority of transparent systems use an agent external to the application to multiplex output from applications onto a number of display screens and to route input from each display to the appropriate applications. In this approach an application does not know that it is being shared between users. Consequently each user is presented with identical views of the interface and only strict WYSIWIS is supported, prohibiting end-user tailorability (Greenberg 1991). Systems that exploit collaboration transparency include Rapport (Ahuja 1990), SharedX (Gust 1988), and MMConf (Crowley 1990).

This approach also relies on an assumed model of cooperation based on many readers but only one writer. The responsibility for managing the interaction involved lies solely with an external agent that uses a floor control policy to co-ordinate interaction. This control policy normally applies to the application interface as a whole with each user having permission to interact with the application in turn by being given the floor (Figure 1).

The combined result of the simple replication of output and the assumed model of interaction is that it is difficult for users to effectively manage collaboration transparent systems. Any facilities that exist have concentrated on the development of techniques to support different forms of turn taking protocols (Roseman 1993; Greenberg 1991).

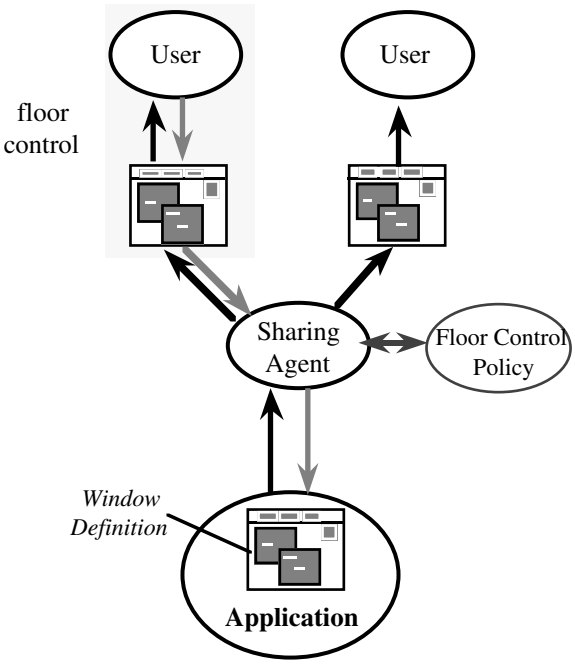

FIGURE 1. A Collaboration Transparent Arrangement

In contrast to the transparent approach outlined above, collaboration aware solutions provide facilities to explicitly manage the sharing of displays between different users. This approach has been adopted by a range of applications including Cognoter (Stefik 1987), Grove (Ellis 1988) and rIBIS (1991). Under this arrangement it is the responsibility of the application to manage the different user interfaces presented, (figure 2). Consequently, users are only supplied with the set of management facilities deemed necessary by the developer. This results in each application adopting a particular set of cooperation facilities unique to that application.

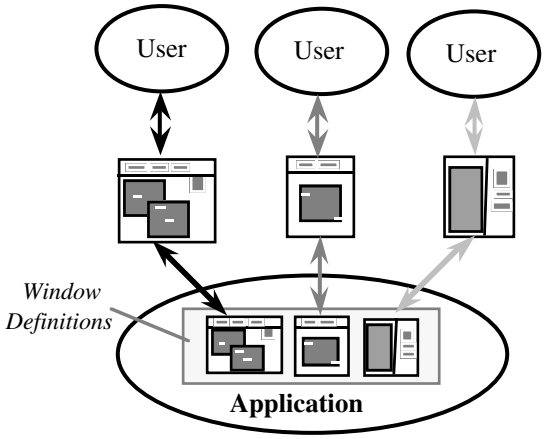

FIGURE 2. A Collaboration Aware Arrangement

Often the facilities predicted by developers of collaborative applications are inappropriate for users (Grudin 1989) and a significant proportion of reconfiguration and tailoring is needed to meet the actual demands of the application. A number of more recent collaboration aware

toolkits have considered how this should be provided by the development of architectures that explicitly support tailorability (Patterson 1990, Bentley 1992). While increasing the availability to users of cooperative management facilities, these toolkits require applications to be developed within their particular programming model rather than within a general framework. For example, MEAD (Bentley 1992) requires the application developer to provide detailed specification of complex objects in order to derive views from them.

Our Approach

We have developed an environment, which allows user interfaces to cooperative applications to be defined, constructed and managed. The aim of our work is to provide a set of cooperative interface facilities which promote user involvement in the tailoring and management of multi-user interfaces. We also wish to provide these facilities in a manner which promotes the establishment of generally applicable cooperative user interface components which can be used across a range of applications. Consequently, we wish to examine the provision of cooperative interface facilities which exist outside the application and are independent of its detailed semantics. Our intent is to provide facilities which:

- Allow multi-user interfaces to be constructed from a set of generally applicable cooperative interface components;
- Maintain a clear separation between the behavioural characteristics of the application and the cooperative aspects of the user interface;
- Provide users with a means of configuring the cooperative behaviour of applications to meet their needs;
- Provide a set of simple and consistent management facilities which can be applied across a range of multi-user applications.

The basis of our approach is to consider the user interface as a collection of different shared user interface objects and to focus on managing this sharing. Rather than manage each object and its associated views independently, we view each user interface as being derived from a common application interface constructed from a collection of shared interface objects. The use of collections of objects to construct user interfaces is not in itself novel and has been widely applied by a number of researchers. However, our user interface objects (termed SOLOs, derived from *SOL O*bjects) have been specifically developed to be shared between users and represent an initial set of cooperative interface widgets. Each of these interface objects is itself collaboration aware and can present itself to a number of users' displays in different ways. Each object also provides external facilities to manage cooperative interaction independent of the application. A selection of these objects have already been defined and constructed which

mimics the facilities provided by the Motif widget set. The choice of starting with the Motif widget set allows us to port a range of existing applications with minimum changes to the source code.

Managing the user interface is achieved through the manipulation of access to these common interface objects. One reason for choosing access as the basis for managing shared user interfaces is the familiarity and simplicity of the concepts associated with access. Access models are both widespread and well understood within multi-user computing systems. Access models have been successfully used for many years to “tailor” a user's control over information in a number of multi-user systems, such as operating systems and databases.

Our general approach is evolutionary in nature, applications are constructed in a similar manner to client programs in the X Window system. Structurally the application is similar to a normal X client, with the addition of a *shared interface layer* (figure 3). As in the case of single user applications a separation is maintained between the deeper semantic behaviour of the application of the interaction properties of the interface objects. The shared layer is used to set up the different user interfaces, and contains an abstract definition of a common interface provided by the application. Three principle concepts are used to manage the cooperative features of the interface:

- The common abstract interface definition is used by an access client to derive alternative interfaces for different users;
- A set of mappings are provided between the shared interaction objects and the callback routines within the application by the shared layer;
- An interaction criteria is used to control the *extent* of collaboration required to invoke these callback routines.

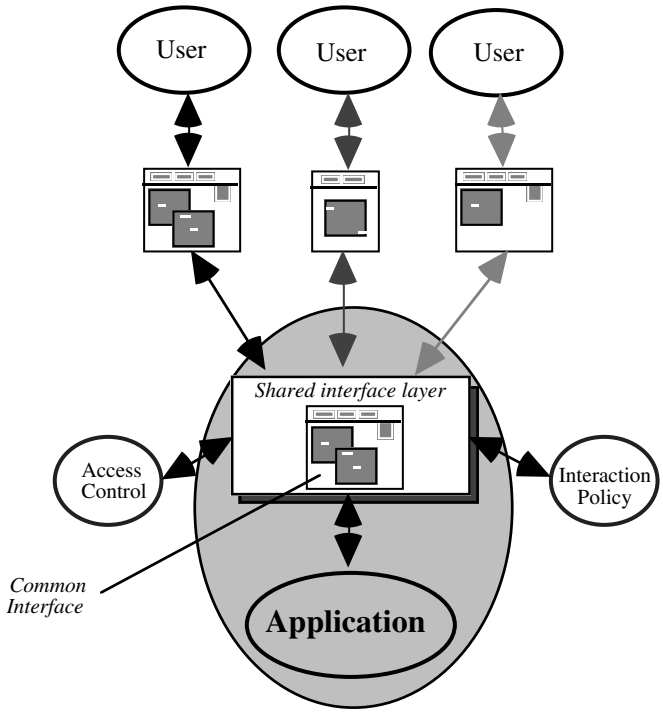

FIGURE 3. The use of a common interface definition

The use of different views of shared elements has emerged as a definite theme within real time cooperative interfaces. Private or personal views are supported in a number of systems such as Suite (Dewan 1990), Liza (Gibbs 1989) and CES (Seliger 1985). A number of architectures also exist (Patterson 1990; Bentley 1992) which support these views through the use of an underlying object model. The model maintains a clear logical separation between a shared object and its views, and separate individual views are derived from the underlying object. Provision of such views permits a control over sharing of information. In this respect we agree with Patterson (1991) when he states "*The essence of multi-user applications is sharing and control over sharing*". However, in contrast to the constraint based approach exploited within Rendezvous (Patterson 1990) we have chosen to extend existing multi-user control techniques to incorporate user interfaces.

The following sections consider the provision of facilities to manage the cooperative aspects of the user interface provided by our shared interface objects. We begin by considering the use of an access model to control both the presentation and crude interaction features of interface objects. This is followed by a consideration of how callback mappings can be used to provide finer grained control of the effects of interaction with these cooperative interfaces.

CONFIGURING PRESENTATION USING ACCESS

The configuration of individual user interfaces within a multi-user application are specified in the SOL environment using an access control system. Each shared interface object controls a

number of derived images of that object and any associated interaction. Shared interface objects may exhibit a core set of configuration and interaction properties; that is, they may be *used*, *moved* or *resized*. Interface objects know how to present themselves to particular users by referring to an access model within the system. Interface objects have a varying number of permissions allowing users access to different interaction properties particular to the object.

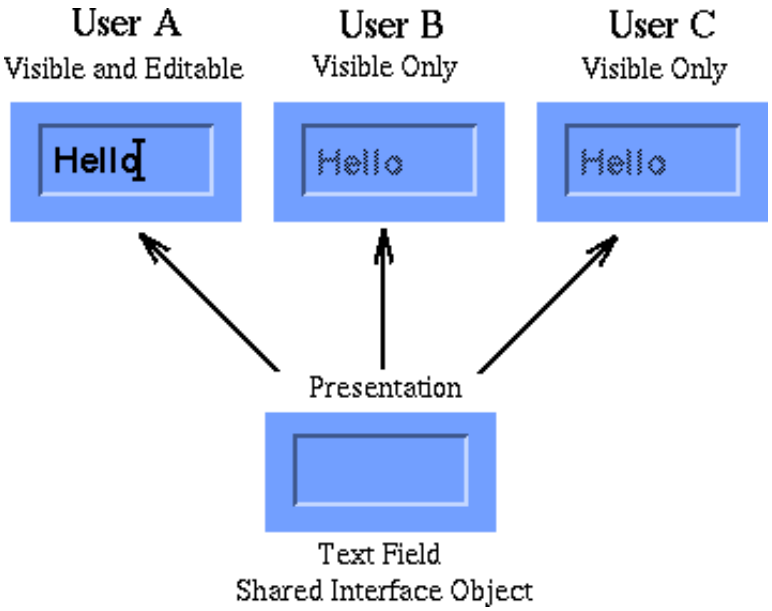

FIGURE 4. A text field shared interface object and three representations

For example, a text field user interface object may show itself to three users in varying ways. A user of the text field shared interface object could have “Visible” and “Editable” permissions allowing the user to see and alter the text object. One user may see and edit the text, while the other two may *only* view it, as in figure 4.

ACCESS MODELS

Access control is widely used within computing systems and many access models exist (Goskinski 1991), most of which are derived from the Access Matrix Model (Graham 1972). In this general model there exists a set of passive Objects which are the resources to be protected, and a set of active Subjects who wish to access these objects. Each Subject, Object pair (S, O) is associated with an access right(s). An access right is a privilege that a subject (S) has over an object (O). A Subject may access an object in a way specified by the access right(s) held by the pair (S, O). A number of more specific models are derived from the Access Matrix including:

- *Capabilities*, which divide the matrix into columns. Each user holds a list of accessible objects. The addition of a new object requires each user’s access list to be updated;

- *Access control lists*, where the matrix is divided into rows. Each object holds a list of users who may interact with it. It is difficult for a user to tell which objects they may use. Addition of a new user requires all objects to update their lists, and the list of users able to access an object may become large.

Although the access matrix defines the principal model for the control over a user's (or subject's) access to an object it has been used in only a few collaborative systems, such as RTCAL (Greif 1987), Suite (Dewan 1990) and Grove (Ellis 1988). In fact, Shen (Shen 1992) makes the point that current access models were not designed for collaborative information, and are inappropriate for direct inclusion in collaborative systems. He highlights the need for access control in collaborative systems in general and argues that "most collaborative systems give all collaborators the same rights to all objects and expect access to be controlled by some social protocol". Shen also highlights a number of specific problems with the existing models derived from the access matrix:

- They do not facilitate the use of roles or groups; individual users must be used;
- They do not facilitate easy specification of access rights for users;
- They often do not offer a general mechanism for specification of the access matrix, or for access checking.

Shen and Dewan forward an access control model that addresses these problems (Shen 1992). Their access model is motivated by Suite (Dewan 1990), which is a multi-user system that allows a number of users to cooperatively edit a text based data structure, held within an application. Shen and Dewan's access model is "*based on a generalised editing model of collaboration, which assumes that users interact with a collaborative application by concurrently editing its data structure*".

The access model used within Suite embodies a list of (approximately 50) access attributes regarding a user's level of access and coupling. These access attributes form an access right over each of the entries within the data structure. Consequently these attributes, such as *AppendR*, *PasredListListenR* and *TitleR*, are closely related to the semantics of the interfaces presented by Suite. That is, the attributes of the Suite access model embody the semantics of Suite. In a cooperative graphical user interface system the access rights such as *WriteR* and *PasredListListenR* are meaningless to user interface objects such as push buttons and containers. This fact greatly restricts the portability of that access model. Shen and Dewan note a limitation of their model is that "*its set of protected objects includes only active variables*" (Shen 1992).

Therefore a general access framework for group collaborative systems is required that applies to all cooperative objects. This access model must address those problems of existing access models, as highlighted by Shen and Dewan, without embedding any semantics of the system using it. This model must also separate a user's access and coupling properties, as they are independent characteristics of cooperative objects.

Our approach is to view each object as one with an associated number of methods which invoke some functionality. We apply an access model to these *methods* and control a user's access to them. Additionally we control the coupling (or sharing) among users though these objects *orthogonally* to their access. This arrangement permits us to control a user's level of access and sharing without knowing the semantics of the object itself.

We have implemented our access model within SOL to control individual user interfaces. The access model is designed to be applied orthogonally, and is "simple" as we wish it to be used quickly by end users. The following sections discuss our model in detail and explain its use in configuring cooperative user interfaces.

OUR ACCESS MODEL

The access model we have defined incorporates many of the features found in existing access models, including the access model used within Suite (Shen 1992). Our approach however is not to produce a matrix of users against objects, and store access rights within this matrix. Rather, we select the *methods* exported by an object and relate them to any number of orthogonal *invocational-domains*. An invocational-domain defines the context in which the method is to act. Each of these matrices is comparable to a user's access right within many of the existing access models. Thus an individual access "right" in our access model assumes the general form of the access matrix in figure 5.

Methods	**Use**	**Sharing**
...	...	...
...	...	...

FIGURE 5. A general permission matrix

The domains we have chosen for use within SOL are :

- **use**, to define whether a user may invoke a method, and;
- **sharing**, to allow the result of a user's invocation of a method to be shared. This controls the level of coupling between users.

Thus a single permission matrix defines how one user may use an objects, and how that user wishes their invocations to be coupled or shared between other users (likewise, how other

user's invocations are shared with them). In our case, the methods relate to user interface objects. For example, a push button user interface object may include the methods :

- Visible. Can it be viewed ?
- Usable. Can it be pressed ?
- Moveable. Can it be moved ?
- Resizable. Can it be resized ?

We have generalised the *usable* method to cover the main interactional property offered by the user interface object. In the case of push buttons, usable defines the method of selecting it, in text fields it describes the method of editing the text within it.

To specify the relations between the methods and domains within a permission matrix, we employ an extendible list of simple flags. Each entry within the matrix may contain a single flag. Our aim was to include the information on *who* may edit the access permissions within any access specification as well as the original specification itself. The original specification in this case is a simple Boolean flag denoting the effect of a user invoking a method, with respect to a particular domain. Our needs located two levels of access, the end user and a system administrator. The access mechanism is designed to be used by end users, but cases exist where the ability for end users to modify their access permissions, from those given, is not desired.

The value each flag may assume, in our implementation of the model, is one of four possible states. Each state denotes whether the access is granted or denied, *and* whether this setting may be altered by the end-user. The four states are represented by the letters {T, F, t, f}, as denoted in figure 6:

	Access granted	Access denied
Not end-user tailorable	**T**	**F**
End-user tailorable	**t**	**f**

FIGURE 6. Possible values each flag entry may assume

So a returned permission for a push-button SOLO will contain at least eight flags: four specifying if a user may see the button, use the button, move the button and resize the button, and four specifying whether the effect of seeing, pressing, moving or resizing the button should be broadcast to other interested parties. When the effects of methods are broadcast, all users who have also configured their interface object to broadcast that method, see the effect of that method. If a user has chosen not to broadcast a method, then any effect is local to that

user's display. This allows users to control the degree of coupling between screens. To this end, we have a 4 by 2 *permission matrix* for the button (figure 7).

Methods	**Use**	**Sharing**
Visible		
Usable or 'Pressable'		
Moveable		
Resizable		

FIGURE 7. A permission Matrix for a Button

This model allows an object to extend the number of available methods. For example, the Suite access model offers the attribute *FontR* as an ability to alter the font of an object. So, a user with the *FontR* rite may alter the font. In our model we would have a *method* to change the font of an object, such as *Changefont*. Thus the matrix above gains an additional row to include this method, which may then have an associated level of access and coupling. This defines who may change the font of an object, and who wishes to share a common font. Consequently all our methods for all interactions are potentially collaborative.

Each permission matrix specifies one user's access definition for one object. These matrices are held within an access list associated with each object. This approach was taken as an object requires the access definition of each user to configure the levels of coupling between them, whereas each user is merely concerned with the visual and interactional properties of the interface and not the access data itself.

Permission Resolution and Permission Inheritance

The basis of our model is an object with an associated list of permission matrices. The specification of permissions for all the users of an object may be an arduous management task. Consider for example, a complex user interface containing one hundred user interface objects, used in a domain of one hundred users, in this case ten thousand specifications must be entered. To ease this problem we adopt the use of *roles* to refer to a number of users. A role may be given an access permission to represent the permission for all the users in that role. This may greatly reduce the number of entries within the list held by an object. The permission matrix for a user who is a member of a role is simply that of the role of which they are a member.

As each user may be a member of a number of different roles, the situation arises where two or more permission matrices exist for a single user. To combat this we have defined a permission resolution mechanism (Smith 94) which derives a single permission matrix from

any number of others. This mechanism includes rules to govern: the end user's own preferences; end user tailorable and non-end user tailorable flags; and position sensitive lists.

To simplify the process of SOLO permission specification for an entire user interface of objects we use the inherent user interface hierarchy to group user interface objects. For example, a menu may contain a number of buttons. Rather than specify the access permissions for each button, we can hold a single copy of the required access permission within the container, which in this case is the menu. Again, a number of rules exist to derive a single permission matrix from those held by the child and parent.

Group access specification is also supported across unrelated user interface objects. SOLOs are given a symbolic name, and it is this name that is used within the access mechanism. This symbolic identification and assignment of access rights allows unrelated objects to share an initial set of access permissions if they are given the same name. Full details of the access model are not considered in this paper and interested readers are referred to (Smith 94) for a more detailed description of these facilities.

THE ACCESS SYSTEM IN USE

The access model is realised as a central part of our developed system. To illustrate the access model in use, consider a simple organisational tool. Named D-sign, the tool facilitates simple brainstorming and structuring tasks, by adding, deleting moving and joining simple articles. An article represents some arbitrary object and consists of a title and a body of text. The user interface consists of a pallet of buttons and a canvas onto which articles are added, see figure 8.

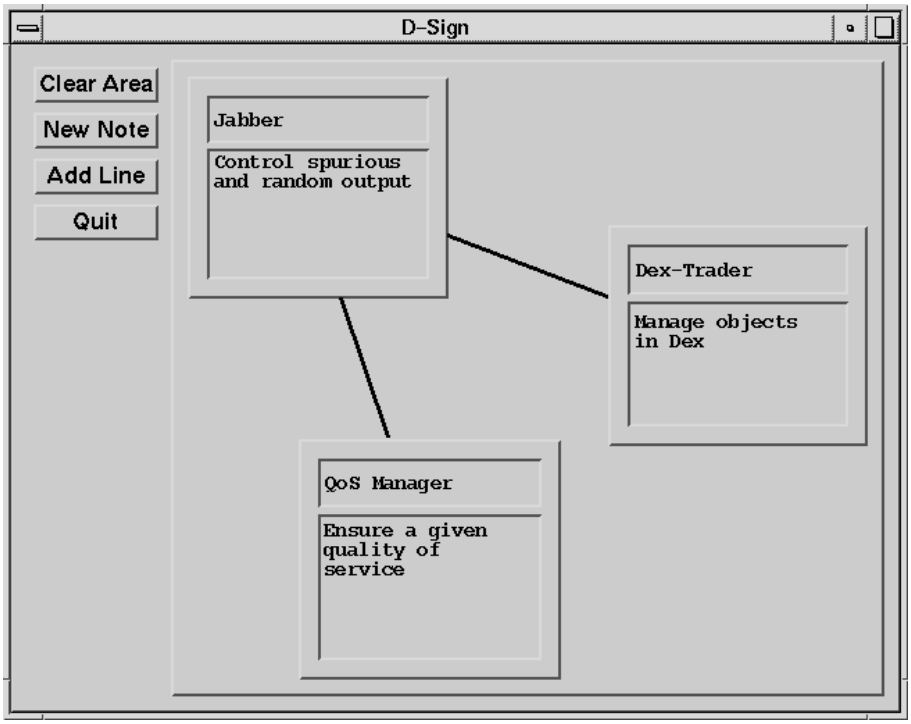

FIGURE 8. The organisational tool “D-sign”

The application allows users to add and delete the articles on the canvas. The articles may then be joined by a line. An article is deleted by selecting the “delete” entry within a pop-up menu associated with it.

The application originally existed as a single user program and was constructed using a user interface development toolkit based on the Motif Widget set for X Windows. The application was ported to our environment by amending the source code to replace the Motif widget set with our developed interface objects. This allowed us to reason about the tool being used cooperatively.

We set the access permissions for the application using a number of roles, these roles were taken from positions at our university, thus the roles we use include *students*, *lecturers*, *administrators* and *professors*. We wished the tool to support maximum functionality for the more authoritative roles such as *professors*, and less for roles such as *students*. Thus a professor may add and delete articles, then reposition them at will, they may also edit the text within any of the fields within articles. Students, however may not add, delete or move articles, or edit the text contained within them.

We may continue to tailor the user interface of the application by adding a further role "*assistants*", which is based on the students’ access specification, but users who are a member of this role may edit the text fields. Such users may refine or edit the notes, thus assisting any members of the lecturer role, but not edit the design layout.

Each user interface object has a management interface which shows the access permissions for it, figure 9. This management interface may or may not allow that user to alter these permissions at run-time. Providing members of the lecturer role with the permission to alter the sharable aspects of the text fields within the articles, allows them to have a common view of the design, with common titles of the object, but have private entries within the body of text in some of the articles, figure 10. This is possible as the results of editing the text are not broadcast, so the effect of editing is local, allowing the user a private view of the text field. Figure 10 also shows that user A has decoupled the position of one of their notes, and moved it independently of user B. The use of the run-time, pop-up, management interfaces allows users to tailor the fine grain coupling of their interfaces dynamically and rapidly.

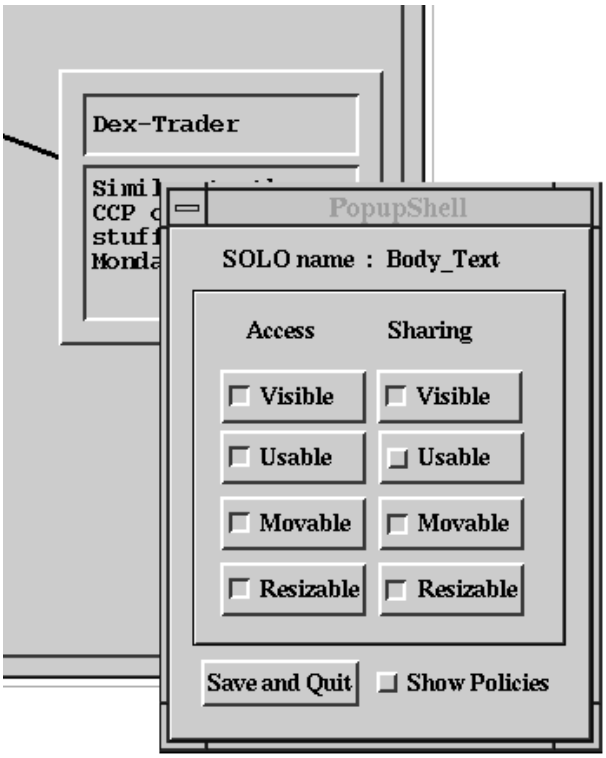

FIGURE 9. Management interface for a text field.

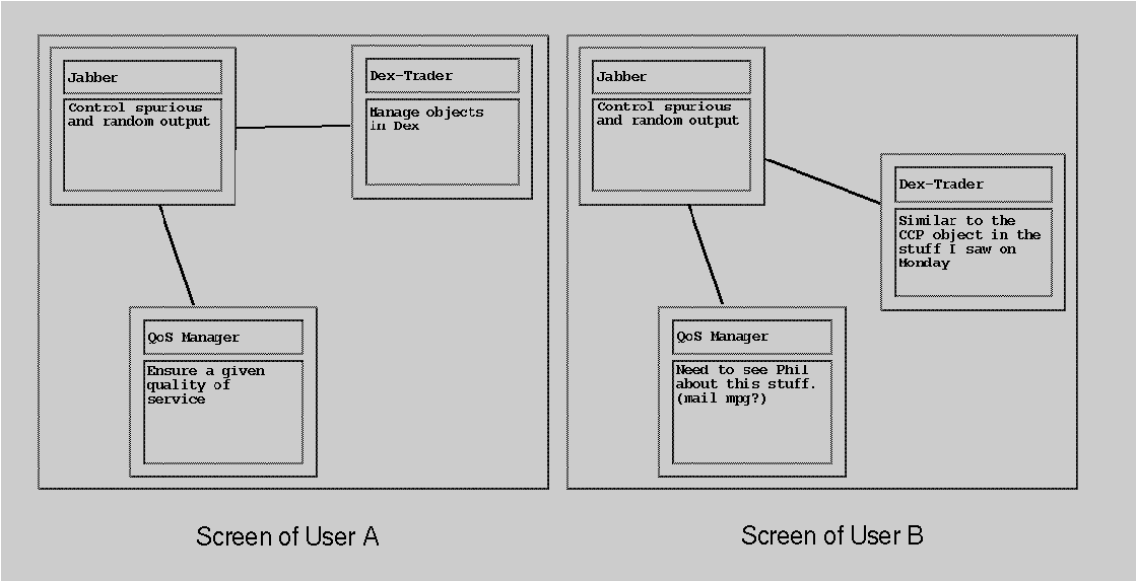

FIGURE 10. fine grain sharing tailorability.

CONTROLLING APPLICATION BEHAVIOUR

The use of an access model to define presentation and sharing of user interface objects allows multiple users to interact with the application through differing user interfaces. For example, three users may see and use a button although the position and size of this button may be

different for each user. Similarly, each user may or may not wish to know when anyone else has pressed it. The use of the access model within our system allows us to specify this behaviour externally to the application. Essentially, our access model makes the application's interface aware of its cooperative setting by allowing users to :

- Specify the presentation and layout of interface objects on different users' displays;
- Interact with different interface objects and for propagation of interaction across different users.

However, *input* from each user is delivered to the shared application without reference to the cooperative setting. For example, the effect of any user pressing a button is identical, in terms of the semantic behaviour of the application. This results from the way in which user interaction is handled by event driven windowing systems (such as X windows, Macintosh, Microsoft Windows). These share a common model where a procedure is called on the occurrence of a particular event. This procedure is known as a "callback routine" in the X windowing system. We use the term more generally to represent the portion of the application called from an interface object. If we choose to use the access model alone to manage shared interface objects, then interaction from any user is handled by executing the same piece of callback code (this arrangement is shown in figure 11).

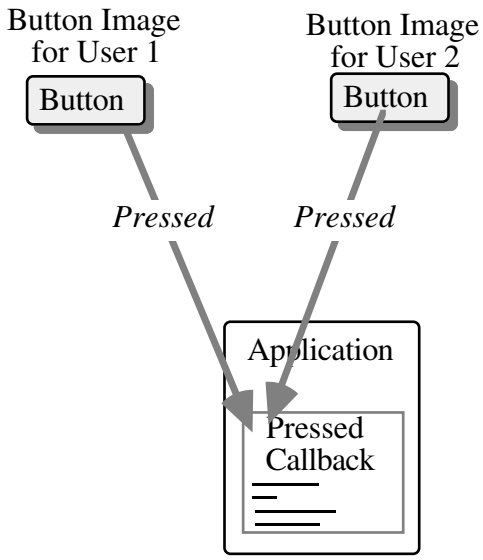

FIGURE 11. Two buttons invoking a callback independently

Often cooperative applications wish to behave differently depending on who is interacting with them. This requires the cooperative setting to have deeper affects on the application. The simple approach outlined in figure 11 severely limits the possibility of providing facilities to manage the deeper semantic behaviour of the application.

Policy Node

In addition to our access model we provide a means of managing collaborative aware interaction through the use of a *policy node* attached to user interface objects. The policy node intercepts input from users and depending on its particular policy, may undertake some cooperative behaviour and then decide whether the callback should be executed (figure 12).

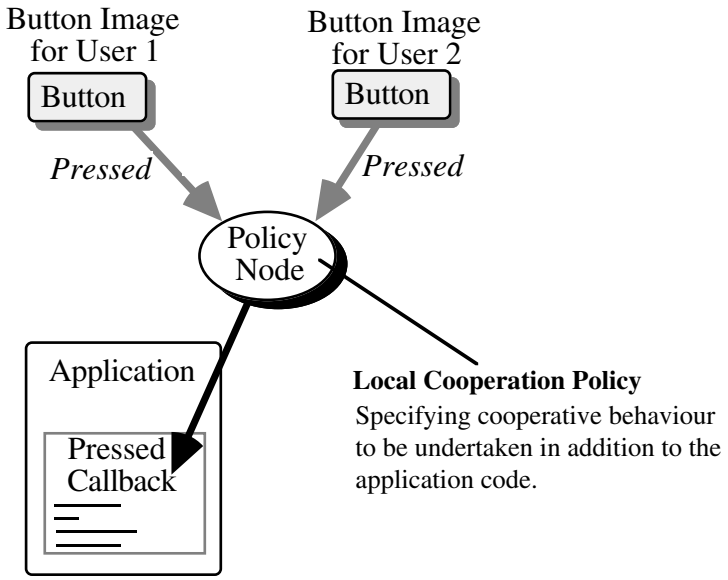

FIGURE 12. Location of the Policy Node

The policy node allows cooperative behaviour to be reasoned about independently of the behaviour of the application and amended dynamically. For example, in the arrangement shown in figure 12 we could state that the callback code should only be called when both users have pressed their button. In this way we provide a means of specifying user coupling (Dewan 1991), allowing users to collaboratively communicate with the application. The use of policy nodes ensures a clear separation between the behavioural semantics of the application and the cooperative setting within which it is placed. This separation promotes the provision of external management facilities and the migration of single user applications to a multi-user setting.

The policy node associated with each SOLO receives inputs from each user when they interact with their representation of the SOLO. The policy node recognises and reacts to particular forms of input to undertake different cooperative behaviour prior to the invocation of the callback code. The policy node receives all input events from the different images of the SOLO and produces an output token. The node contains a set of criteria that determine which output

token will be produced for a particular set of input events. This arrangement is shown in figure 13.

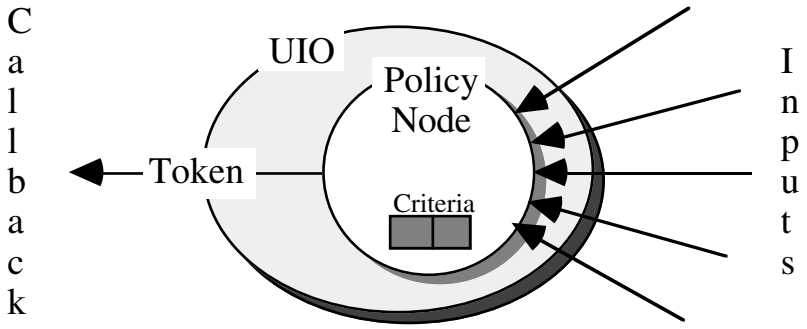

FIGURE 13. The policy node model

The critera in the policy node allows the SOLO to react to the different patterns of input in different ways. Rather than simply executing the callback code, the use of criteria tells the SOLO *or* the callback code that a certain condition has been met. For each recognised criteria, the policy object produces a different output "token" (figure 13). The token may be either:

- *SOLO specific*, in which case it commands the SOLO to do something, or;
- *Application dependant*, where the token is passed to the callback code, and the *application* decides the nature of the cooperation.

Tokens are simply strings, SOLO specific tokens are prefixed with “SOLO”. For example, *SOLO_LOCK_WORD* will cause a text widget to lock the word under the user's cursor; it is not passed to any callback routine. Most SOLO specific tokens facilitate semantic feedback to other users. For example other users may see that the locked word has actually been locked. Different types of SOLOs have an associated set of output tokens defined for them. In developing our set of SOLOs we have defined an initial set of output tokens for a selection of SOLOs. Some of these and their associated behaviours are described in figure 14. In contrast, application dependant tokens are ignored by the SOLO and passed straight to the callback routine. These application specific tokens may be any string and are interpreted by the application.

SOLO type	SOLO specific token	SOLO Behaviour
Text Field	SOLO_LOCK_WORD	Lock the word that the user's cursor is within, (when he/she starts typing); free when the cursor moves away from that word.
	SOLO_LOCK_SENTENCE	Lock the entire sencence containing the user's cursor; free when cursor moves away from the sentence.
	SOLO_LOCK_ALL	Lock all the text within the text field; free when user's pointer moves out of SOLO.
	SOLO_OVERRIDE	Ignore all locks.
Scrollbars (Slider)	SOLO_FORCE_MOVE_ALL	Override the access constraints and synchronise all representations of this scrollbar.

FIGURE 14. A selection of output tokens associated with SOLOs.

The criteria determining the policy are represented in the form of a table, with one side listing the required context (type of input), and the other the token (or tokens) to be sent. A simple *if* <*input = context*> *then* <*token*> rule is used to interpret the policy. For example, "IF username = gbs THEN SOLO_OVERRIDE".

A rule based approach was preferred, primarily, as it is simple. The specification of cooperative behaviour via the policy node is to be tailored at run-time by end-users. Thus the information presented by the policy node must be easily and quickly modifiable by non-programmers. A code based approach would require end user's to comprehend basic procedural programming and have the ability to understand parsing errors produced when the code was translated or compiled. Single line rules facilitate the use of a simple form-based user interface, whereas a code based approach would require either a text editor (which may be daunting to non-programmer end users) or a powerful structured form editor, which modifies itself with respect to the code (this is a large overhead to provide for and requires the user interface to be consistent with the language used, reducing the extendibility of the language).

We have two types of context; ***selective*** and ***consensus***. *Selective* behaviour is available for all interaction SOLOs (buttons, text fields, etc.). This type of behaviour queries the status of the user interacting with the SOLO. *Consensus* behaviour is exhibited within a specialised type of button. In these buttons (which may also exploit selective behaviour), the number of users who have selected their button is used as the criteria. We have two types of consensus buttons, *accumulated* and *total*. These compute the number of selected buttons over a specified sampling period.

Accumulated buttons count the number of users who are currently holding down their buttons. The sampling period is temporal (specified by the end-user, using the policy node). That is, the SOLO waits for a specified time after the first user pressed their button. After this time the

SOLO counts the number of buttons *currently* held down. This value is then used to determine the correct token to be passed.

Total buttons operate in a similar fashion, but this time the sampling period is controlled by a user. For each button of this type, two buttons may appear; one is the consensus button itself, the other is the controlling button to *end* the sampling period. The access control system is used to constrain the availability of this second button. This type of button does not count the number of buttons *currently* held; instead, it counts the number of users who have clicked their button *at any time* during the sampling period (one per user). The sampling period is terminated when the control button is selected.

We have two types of expressions (or *criteria*) associated with the number of users who have pressed their buttons:

i) An explicit number and;

ii) A percentage value.

For example, a specification of when to execute the callback code may be "TOTAL> 5" or "TOTAL> 50%". The former executes the callback when over five users have pressed the buttons, the latter when over half the users have pressed their buttons.

As a simple example, we may wish to reflect a button's cooperative setting by using some form of consensus to determine if the behaviour associated with the button is to be undertaken. An appropriate criteria for the SOLO is shown in figure 15. This criteria ensures that if over fifty per cent have pressed the button the, callback code is executed and the “marginal” token is passed; if over sixty per cent of the users have selected it the token, “overwhelming” is passed.

If the latter case is true, for example if 75% of the users have their buttons pressed, then the former case is also true. But only the token “overwhelming” is passed, because the policy node automatically places lower bounds on each context entry. Thus in figure 15, we have the ranges 0 to 50; 51 to 60 and 61 to 100.

Any explicit values in a policy node, for example “ACCUMULATED > 10” (more than ten users), are regarded as percentage values (at run time) by the policy node. These percentage values are treated as normal percentage values, and thus used to calculate lower bounds. The two tokens here are purely application specific, and are passed to the callback code. The callback code may choose to ignore these tokens, or use them to process some information in a certain way.

Context	Output Token
ACCUMULATED > 50%	Marginal
ACCUMULATED > 60 %	Overwhelming

FIGURE 15. A simple set of criteria

Simple policy nodes of this form can modify the behaviour of interaction objects within a multi-user interface in a limited way. A simple consensus button fits this model, however we are severely limited if the number of users is our only input type. To overcome this limitation we use the *selective* criteria within policy nodes. Selections can be made on information held within a *user's profile*. This allows us to also exploit information about a particular user to determine policy. For example, we can specify that we only want the button to execute the callback code if more than five users have pressed it **or** i*f a member of staff has pressed it*.

The User Profile

Multi-user applications which use shared interface objects are executed in a specialised run time environment (Smith 1993). This environment includes a central server which all the applications communicate with. This server maintains a *user profile* for each user permitted to use the system. The profile is represented as an extendible set of resources associated with each user. These may be general in nature or application specific. Application specific resources only make sense to policy nodes associated with a particular application. More general resources such as user name, role and title may be used across a number of applications.

The policy node may use any of the resources held within a user's profile in the input context of a criteria. As an example, consider a policy node associated with a button in a shared application used within an academic department. The application is run within an environment which records user activity. A criteria for the policy node using profile information is shown in figure 16.

Input Context	Output Token
PROFILE.Activity=Examiner	examiner
PROFILE.Role=Staff	staff_override

FIGURE 16. A set of criteria using a user profile

This criteria will execute the callback code when a user in the *Examiner* activity presses the button, which passes the token “examiner” to the callback code. The tokens used in this example are not SOLO specific; that is, they are ignored by the SOLO. The second entry in the above example operates in a similar manner, but this time passes the token "staff_override" to the callback code. The policy node using this criteria makes the shared application sensitive to activities external to the application.

The use of application specific tokens provide a powerful way to tailor cooperative interaction. It provides us with a single point of amendment to allow the migration of single user applications to cooperation sensitive ones. The application callback code can be written to undertake a specific action if the token "staff_override" is passed to it and another when

"examiner" is passed. The external statement of policy allows us to tailor the criteria table at run time to allow other users to emulate particular actions. For example, we can allow a particular user to undertake the behaviour associated with "staff_override" by adding the line:

PROFILE.Name="Gareth Smith"	staff_override

FIGURE 17. Amending use through an additional criteria line.

We may also remove this line when it is no longer needed. This use of application tokens relies on amending the callback code within the shared application. However, when migrating applications, access to the callback code will often be limited. To deal with this situation we use SOLO tokens to amend the behaviour of shared interface objects. As mentioned earlier, the SOLO interprets these tokens and may alter any user's representation. In this way we allow collaboration awareness independently of the application. That is, the semantics of the user interface defined within the application are unchanged, but the behaviour of the collaboration is defined externally, within each policy node.

To illustrate the use of SOLO tokens, consider a text field within the D-sign application introduced in our consideration of access. We may wish a member of staff to lock all the text, and students just one word at a time. We define the criteria of a text SOLO as shown in figure 18. In this case, students lock only the words they are editing (one at a time), whereas members of staff lock the entire sentence they are working on. If a user is acting as a spell checker they are unrestricted by any locks, and may change any word.

Context	Output Token
PROFILE.Role=Student	SOLO_LOCK_WORD
PROFILE.Role=Staff	SOLO_LOCK_ALL
PROFILE.Activity=SpellChecker	SOLO_OVERRIDE

FIGURE 18. The criteria for a text SOLO

We can further modify the collaboration semantics for this application. For example, we wish students to see and use the "Clear Area" button (which deletes all the articles within the shared canvas), but for it only to fire if the majority of buttons are pressed. We also want a *member of staff* to have unrestricted access to this button, allowing them to clear the area immediately. In this case we can use the criteria shown in figure 19.

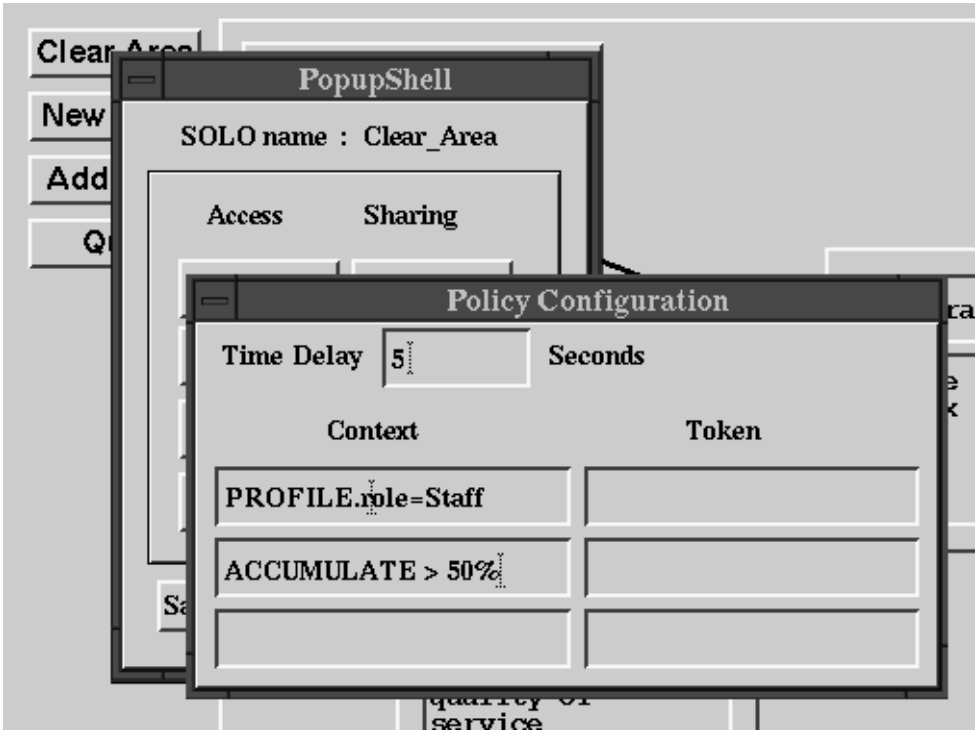

FIGURE 19. The use of policy in the slide viewer application

When we mix consensus and selective contexts, each press of a user's button is matched (during the sampling period) to any selective contexts. If a selective context is matched, then the correct token is passed (or executed), the sampling period then terminates and all the buttons are reset. If none of the inputs match any selective contexts during the sampling period, the consensus contexts are used as normal.

In this case, as buttons are pressed, the node checks if their origin is from a member of staff, and if so, the callback is executed. Otherwise the node counts the number of buttons held down *after* five seconds of the first user selecting it. If this result if greater than half the current number of users, then the callback will also fire.

The policy for the button displayed in figure 19 uses the management interface associated with the shared interface object. These management interfaces are available to user during run-time to allow detailed tailoring of the shared user interface.

The policy configuration window, as in figure 19, displays the rules in a number of text fields. Each of the text fields are constructed from SOLOs. Each of these SOLOs are fully shared, thus a number of users may edit the policy information simultaneously. This also allows the visible and usable methods to be constrained for some users, so that some users may not see the policy information, some may see it but not edit it, while others may see and edit the policy criteria.

Node Linking

Multi-user interfaces in our environment are all constructed from objects. We can use the policy node to not only send a token to the callback code or the SOLO itself, but also to another SOLO. This allows us to reflect cooperative use externally from the application. For example, we may require a title of a label to be modified at run time, depending on whether a member of staff or over half of the users have selected it. A suitable criteria is shown in figure 20.

Context	Output Token
PROFILE.Role=Staff	call “MainTitle” <Label ‘Staff ‘>
ACCUMULATED >50%	call “MainTitle” <Label ‘Most’>

FIGURE 20. Passing tokens to other SOLOs

In the table above, we can see the button reacts to either a member of staff pressing it or at least half the users pressing it. In the former case the label of the “MainTitle” object will change to “Staff”. Likewise its label will change to “Most” when over 50% of the users have pressed the button.

When we wish to communicate with other SOLOs in the user interface, the format of the output token is:

[**Token**] [**call** “*SOLOname*” < *Attribute Value*>]

A call is made to a SOLO of the name “SOLOname” and passed a value attribute pair for the SOLO. In this case the title of the SOLO is altered however we could amend other properties of the user interface by sending different attribute value pairs (e.g. <Size 20x20> or <Coords 200x20>).

Updating of other SOLOs via the call with a token is restricted by the access mechanism. That is, a call made by a token to update another SOLO’s size would only take place if that user has the resizable permission for *that* SOLO. Further, if the user has specified that the resizable method was shared, this change may also be reflected in other users’ displays.

THE TOOLKIT

A central aim of our work is the provision of facilities which allow cooperative user interfaces to be rapidly constructed and supported within the context of iterative development. Our principal focus has been on the realisation of mechanisms which allow this by augment existing models of the interface. We have developed mechanisms which allow the sharing of user interfaces to be controlled, and support more detailed specification of the cooperative

aspects of interaction. These mechanisms are intended to support real time amendment of interface properties as part of a dynamic development process. Central to this development is the provision of a toolkit which provides high level facilities for the specification and tailoring of user interfaces.

We have developed a number of tools to assist with the creation of cooperative interfaces within the SOL environment. A user interface builder, written with the MOG toolkit (Colebourne 1993), allows a user interface to be constructed by "drawing" it using a number of user interface objects (see figure 21). The output of this tool is the C++ source code of the interface using SOLOs. This code may then be compiled and used immediately within our environment.

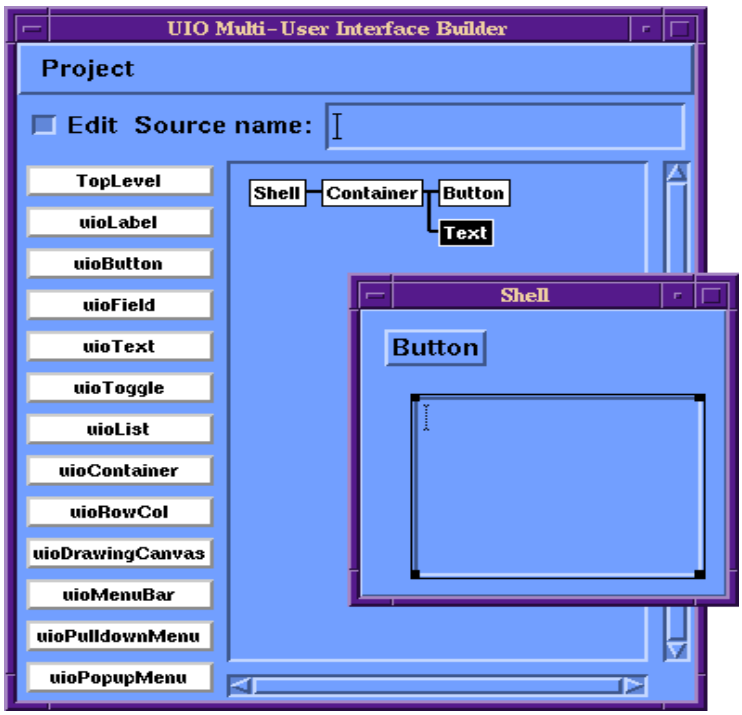

FIGURE 21. The SOL user interface constructor

The user interface builder constructs the common user user-interface for an application. The interface for the slide viewer application used in earlier examples in this paper was created in this way, using a canvas, a text field and two buttons. We have used the builder to construct a range of applications including, a shared brainstorming systems, a minute taker and a cooperative network management facility.

The access control and interaction policies are specified after the application interface is built from SOLOs. This may either be done by modifying access and policy on a per object basis through their management interfaces. Alternatively, the SOLO can be configured from an

initial set-up by an authorised user. At any time in the application's lifetime, end users may tailor their interfaces using the management menus associated with the interface object.

ASSESSING THE SYSTEM IN USE

Evaluation and assessment of a user interface construction and management system such as SOL is inherently difficult. It is unclear exactly what we are intending to assess and evaluate when we examine interface toolkits such as SOL. Three obvious choices exist for this form of toolkit (Twidale, 1994) :

- *The user interfaces presented by the toolkit.* The ease of learning and using the tools provided for different interface developers;
- *The coverage of Toolkit*. The ability to generate the kinds of interfaces required by a wide variety of end-users;
- *The different user interfaces produced by the system*. The usability of the interfaces that can be generated by the SOL interface can themselves by assessed and evaluated.

This multiplicity alone makes evaluation or assessment of SOL much more problematic than the end user evaluation of an individual tool or application. The complexities involved are compounded further by the considerable uncertainty of the nature and role of evaluation in CSCW systems design (Grudin 1989). It is still unclear that reliable techniques for the assessment of Cooperative Systems have emerged from the research community or that they are imminent without considerable methodological debate. Much of this uncertainty reflects differences between traditional experimental and cognitive approaches to HCI and CSCW and the forms of design that turn much more to input from the social sciences. Interested readers are referred to (Bannon 1993) and (Twidale 1994) for a summary of these debates.

The uncertainty surrounding assessment and evaluation places the authors as developers of novel cooperative systems and techniques in some difficulty. While we are motivated to construct systems that are usable and of benefit to cooperative users the lack of agreement as to how to undertake assessment places us in some difficult in undertaking any form of summative assessment of their utility. In response to this dilemma we have adopted an informal approach to assessment aimed at informing the construction of the system as it has emerged. This approach is in line with previous development and assessment at Lancaster and has focused on the use of informal assessment of use as a means of guiding development. A fuller discussion of this approach is given in (Twidale, 1994).

As a set of generic facilities to support a wide variety of cooperative applications we envisage SOL being used to support a wide range of cooperative applications. Given the breath of coverage of SOL it is difficult to assess the usability of all the potential interfaces to be produced by the toolkit. Consequently, our usage has focused on making the tool available across the community of developers at Lancaster to support the construction of a range of cooperative systems. These applications have included:

- The development of a cooperative network browser for displaying different resources available on the department network;
- The development of an idea generation and arrangement facility;
- The construction of set of computer based training applications;
- The development of an interface to support the cooperative browsing of on-line information sources.

The use of SOL has focused on the utility of the interfaces used to manage the system and the ability of facilities provided to support a wide range of cooperative applications. The community of users were all system developers within a research laboratory familiar with the construction of X-windows based applications. Most found the interfaces presented to them usable and other than commentary on detailed layout and wording few problems were reported. Given our forgiving community of users this result is not surprising. It is worth stressing that this community is not completely divorced from the eventual users of a system and it is reassuring to find that our proof of concept demonstrator was sufficiently useful to be used in constructing a range of applications. However, our confidence in the usability of the current system needs to be tempered by widening the current limited user base.

Perhaps more informative than a consideration of the usability of the construction tools provided as part of SOL is a reflection of the scope of the facilities provided. As a generic toolkit for the construction cooperative user interfaces SOL wishes to support interfaces for a variety of cooperative applications. Our experiences with SOL in developing applications have been a little more mixed in this regard. SOL has proven to be useful for cooperative applications where one application has a readily identifiable common abstract interface that others may be derived from. For many of the applications this was not particularly problematic. However, the reliance on a single abstract interface assumes a *single* state of sharing, per SOLO. For example, the contents of a text field may be private, or shared. There is no notion of sub-group sharing. The current implementation of SOL does not support radically differing user interfaces, such as the ability for one user to see a bar chart, while another sees a pie chart. Both these limitations were highlighted as problematic by our user

community and we are currently considering extensions to the access model and the interface toolkit to support these features.

The applications developed by the SOL prototype were drawn from unrelated projects across the research laboratory. Interfaces were built very quickly (mostly within one or two days) and the ability to rapidly construct and amend the features of cooperative interfaces were highlighted as a desirable feature of the toolkit. Experiences within the lab of using alternative toolkits for constructing applications was limited. Other than availability, one reason given for this was the reliance on specialised platforms inherent in many existing toolkits. For example, MEAD (Bentley, 1992), Rendezvous (Patterson, 1990) and OVAL (Malone, 1992) all exploited the facilities provided by specialised environments. In contrast, users felt that SOL promoted the construction of cooperative applications without advance knowledge of the semantics. However, users also felt that this required them as developers to construct more detailed cooperative facilities. Consequently, many saw SOL as a point of departure for constructing cooperative applications rather than a complete toolkit.

FUTURE ENHANCEMENTS

A desired addition to our environment is support for inter-application communication. Initially this may take the form of a messaging bus, but a shared memory approach to application interaction may be preferable, for the reasons of efficiency outlined by Koshizuka(1993). We also wish to investigate some distributed issues. Our environment is distributed, but coarsely grained. Each application, user interface and the system server may execute on different hosts, but we still have a central node of failure within the centralied server.

Moreover if we scale up our domain from local area, to wider area networks, the delays incurred may make the environment unusable. For example, delayed user interface events may render the timed *consensus* buttons useless. We would therefore desire our environment to support more fine grain distribution, including the use of a replicated federated server.

We wish to explore support for an activity model (Araujo 1988; Danielson 1988), by using the *user profile* to store activity information. We are also investigating the possibility of extending the SOLO linking call, to support an attribute value pair containing an *access specification* for the new SOLO. This would allow a great deal of flexibility in automatically manipulating multi-user interfaces at run time, as we may turn other SOLOs “off” and “on” (visible / invisible or usable / greyed out) through other interactions. Using the messaging system, we may extend the node linking feature to call SOLOs held within *another* application, to further enhance inter-application interface communication.

So far, we have implemented little to support telepointing. Current telepointers, in systems such as GroupSketch (Greenberg 1991), are used in WYSIWIS areas of each user's display. In a non-WYSIWIS environment, we cannot directly map the coordinates of one telepointer to another, because the item being pointed to may not be at the same coordinates for all user interfaces. We therefore wish to experiment with techniques to support non-WYSIWIS telepointing in our toolkit.

CONCLUSIONS AND SUMMARY

We have provided a means to tailor collaborative aware user interfaces within our environment using an access control system. Using this system we can accurately control any particular user's access to any user interface object, and specify how that object is shared. It allows us to use tightly coupled collaborative aware user interfaces and (at the same time) collaboration transparent user interfaces.

The use of a policy node provides collaboration aware or collaboration transparent *application input.* That is, the application may receive input from users individually, or from a group of users acting cooperatively. The node provides us with context sensitive input. For example, one user's interaction with an interface object may result in a different action to that of another user. Also, the action of a group interacting cooperatively with an interface object may differ depending on the number of users.

We have developed a number of cooperative application using the SOL environment and toolset. These include a simple brainstorming generator tool, a minute taking facility and a network administration tool. In addition, we have also used our environment to produce cooperative aware multi-user tools from existing single user applications. An example of this is the D-sign tool; this simple application was converted into a powerful cooperative tool with minimal effort.

The mechanisms that support these collaborative features are external to the application code. Separating application behaviour and cooperative use in this way allows us to outline an initial set of generally applicable cooperative features. We believe that the development of cooperative interface objects in SOL and their associated access and policy nodes represents an initial outlining of a set of cooperative interface widgets. We believe that the development of general purpose building blocks of this form represents a significant research goal for cooperative application developers. In addition, our use of shared objects allows existing single user applications to be readily ported to collaborative settings. In essence we require the use of existing motif widgets to be replaced by SOL's interface objects.

The collaborative mechanisms of SOL interface objects are dynamic in nature. They allow the collaborative behaviour of an entire, or any part of, a user interface to be changed at run-time (by any subset of users). These mechanisms within interface objects may also be used to provide "automatic" features, such as changing another SOLO's label, size, etc. when a specific input is received. Since these mechanisms are associated with general user objects rather than particular applications, shared user interface objects also promote orthogonality of cooperative facilities across different applications.

ACKNOWLEDGEMENT

This work was partially funded by the COMIC project (ESPRIT project no 6225), the Science and Engineering Research Council and IBM Laboratories UK. Our thanks are due to our partners in the project, and our colleagues within the department at Lancaster.

REFERENCES

AHUJA, S. R., ENSOR, J. R., & LUCCO, S. E. (1990). A comparison of applications sharing mechanisms in real-time desktop conferencing systems. *Proceedings of the Conference on Office Information Systems*. Boston 238-248.

ARAUJO, R. B., COULOURIS, F., ONIONS, J. P., SMITH, H. T. (1988). *The Architecture of the Prototype COSMOS Messaging System*. Elsevier Science Publishers B.V., North-Holland.

BANNON, L. (1993). Use, Design and Evaluation: steps towards an integration. *Proceedings of the 12th Schaerding International Workshop on Design Of Computer Supported Cooperative Work And Groupware Systems*. June 1-3 1993. D.Shapiro, M. Tauber and R. Trunmueller (Eds.), Elsevier Science, Amsterdam.

BENTLEY, R., RODDEN T., SAWYER P., SOMMERVILLE I. (1992). An Architecture for Tailoring Cooperative Multi-User Displays. *Proceedings CSCW'92* . ACM Press. 187-194.

COLEBOURNE A., SAWYER P., SOMMERVILLE I. (1993). MOG user interface builder: a mechanism for integrating application and user interface. *Interacting with Computers*. Volume 5 No. 3. 315-331.

CROWLEY, T., MILAZZO, P., BAKER, E., FORSDICK H., TOMLINSON R. (1990). MMConf: An infrastructure for building shared multimedia applications, *Proceedings of CSCW '90*. October 7-10, Los Angeles, Ca. ACM Press. 329-342.

DANIELSON, T., PANKOKE-BABATZ, U. (1988). The Amigo Activity Model. *Research into Networks and Distributed Applications*. R.Speth (Ed.). Elsevier Science Publishers B.V., North Holland. 227-241.

DEWAN P. (1990). A tour of the Suite User Interface Software. *Proc. of the 3rd ACM SIGGRAPH Symposium on User Interface Software and Technology*. October 1990. 57-65.

DEWAN P., CHOUDHARY R. (1991). Flexible User Interface Coupling in a Collaborative System. *Proc. Computer Human Interactions* . 41-48.

ELLIS, C., GIBBS, S. J., REIN, G. (1988). Design and use of a group editor. *Technical report STP-263-88*. MCC, Austin, Texas, September 1988.

GIBBS, S.J. (1989). LIZA: An Extensible Groupware Toolkit. *CHI '89 Proceedings*, ACM Press. 29-35.

GOSKINSKI , ANDRZEJ. (1991). *Distributed Operating Systems - The Logical Design*. Addison-Wesley 1991 ISBN 0 201 41704 9. Chapter 11 Resource Protection. 585-647.

GRAHAM G. S., DENNING P. J. (1972). Protection: Principles and Practices. *Proc of the AFIPS Spring Joint Computer Conference 1972*. 417-429.

GREENBERG, S. (1991). Personalisable Groupware: Accommodating Individual Roles and Group Differences. *Proceedings of ECSCW '91*, Bannon, L., Robinson, M., Schmidt, K. (Eds), Sept 25-27, Kluwer. 17-31.

GREENBERG, S., BOHNET, R. (1991). GroupSketch: A multi-user sketchpad for geographically-distributed small groups. *Proc. Graphics Interface 1991*. 207-215

GREIF, I., SARIN, S. (1987). Data sharing in group work. *ACM Transactions on Office Information Systems*, 5(2). 187-211.

GRUDIN, J. (1989). Why groupware applications fail: Problems in design and evaluation. *Office: Technology and People* Vol. 4 No 3. 245-264.

GUST, P. (1988). Shared X: X in a distributed group work environment. *2nd Annual X conference*. MIT, Boston, January 1988.

KOSHIZUKA N., SAKAMURA K. (1993). Window Real-Objects: A distributed shared memory for distributed implementation of GUI applications. *Proc. of ACM Symposium on User Interface Software and Technology* , Atlanta, Georgia, USA, November 1993. 237-247.

LAUWERS, J.C., LANTZ, K.A. (1990). Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. *Proc. CHI 1990.* 303-310.

MALONE TW, LAI K-Y, FRY C. (1992). Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. *Proceedings of ACM CSCW'92*. ACM Press. Toronto, November 1992, ACM Press. 289-297.

PATTERSON J. (1991). Comparing the programming demands of single user and multi-user applications. *Proc. of ACM Symposium on User Interface Software and Technology.* November 1991. 87-94

PATTERSON, J. F., HILL, R. D., ROHALL, S. L., MEEKS, W. S. (1990). Rendezvous: An architecture for synchronous multi-user applications, *Proceedings of CSCW '90*, October 7-10, Los Angeles, Ca., ACM, 1990. 317-328.

REIN, G. L., ELLIS, C. A. (1991). rIBIS: A real-time group hypertext system, *International Journal of Man Machine Studies*, 34(3), March 1991. 349-368.

ROSEMAN M., GREENBERG, S. (1993). Building flexible groupware through open protocols. *Proceedings of COOCS'93 ACM international conference on Organisational Computing Systems*, San Jose, November 1993, ACM Press. 279-288.

SELIGER, R. (1985). The design and implementation of a distributed program for collaborative editing. *Lab. for Comp. Sci. Tech. Rep. TR-350*, MIT, Cambridge, MASS., Sept 1985.

SHEN, H., DEWAN, P. (1992). Access Control for Collaborative Environments. *Proceedings of CSCW '92*, Toronto, Canada, ACM Press. 51-58.

SMITH. G., RODDEN T. (1993). Using an Access model to configure multi-user interfaces *Proceedings of COOCS'93 ACM international conference on Organisational Computing Systems*, San Jose, November 1993, ACM press. 289-298.

SMITH. G., RODDEN T. (1994). An access model for shared interfaces. *Collaborative Computing*. Vol 1, No. 2 Chapman & Hall. 109-127.

STEFIK, M., FOSTER, G., BOBROW, D. G., KAHN, K., LANNING, S., SUCHMAN, L. (1987). Beyond the chalkboard: computer support for collaboration and problem solving in Meetings, *Communications of the ACM*, 30(1), January 1987. 32-47.

TWIDALE, M. RANDAL, D., BENTLEY, R. (1994), Situated Evaluation for Cooperative Systems. *Proceedings of CSCW'94*, 22-26 October 1994, North Carolina, ACM Press. 441-452.