Supporting Flexible Communication in Heterogeneous Multi-User Environments

Jian Zhao & H. Ulrich Hoppe

GMD-IPSI, 64293 Darmstadt (FRG)
e-mail: {zhao,hoppe}@darmstadt.gmd.de

Abstract

A flexible communication model for multi-user interfaces is presented. It allows for coupling arbitrary user interface objects between heterogeneous applications dynamically. When coupled, UI objects are synchronized by broadcasting events and re-executing the corresponding actions in the different application environments. To be coupled, objects have to be compatible, but not necessarily identical. The communication mechanism is based on a centralized server. In addition to multiplexing callbacks, it also handles arbitrary remote procedure calls in a standardized way, thus allowing the transfer of internal state information between applications. Our communication mechanism is implemented as a set of primitives that extend an OSF/Motif-based UI toolbox library. It can be easily used to develop multi-user interfaces in very much the same way as single-user applications, or to extend single-user applications to multi-user ones.

Keywords: *(Multi-)User Interfaces, UIMS, CSCW*

1 Introduction

There is a growing demand for graphical user interfaces to support several users distributed across computer networks and interacting simultaneously with certain applications. Much work has been focused on WYSIWIS systems [22,23], as e.g. in synchronous computer conferencing or real-time multi-user editors [6]. WYSIWIS (What You See Is What I See) characterizes interfaces through which all participants share identical displays at any time. A synchronization unit is on the level of fine-grained user actions such as keystrokes or mouse movements. The major disadvantage of WYSIWIS is its inflexibility on four dimensions as identified by Stefik et al. [23]: *display space* (participants have identical displays), *time of display* (each participant sees simultaneously the other users' modifications during an entire session), *population* (one participant has to interact with the whole group, and not e.g. with a selected subgroup), and *congruence of views* (displays have the same visual characteristics). We would like mention *application-dependency* as another important restriction: Even if certain WYSIWIS systems go beyond sharing the entire screen display, they typically only allows for sharing instances of one and the same application. There are a number of suggestions for relaxing strict WYSIWIS on several different dimensions, mostly on the dimensions of time, population and congruence. E.g., GroupDesign [2] is a time-relaxed multi-user graphics editor which enables the user to keep modifications private until commitment. Relaxations on the dimension of time are also called "loose coupling" (as opposed to "tight coupling"). rIBIS [20] and SEPIA [9] are both multi-user hypertext systems that support both loosely-coupled and tightly-coupled collaborations. Relaxing on population and congruence of view, GROVE [6] allows users to have private and public work areas, and to use different colors for certain purposes (e.g. notifications).

From a functional point of view, systems for "Computer Supported Collaborative Work" (CSCW) aim at two different kinds of multi-user support: *process-oriented* support through *communication* and *product-oriented* support for *merging* or putting together individual contributions in the form of a joint product. Of course, both aspects are interrelated since work on a joint product usually requires a certain amount of communication, and, on the other hand, communication also means sharing data in a certain sense. Product-oriented support leads to complex problems of maintaining and handling different versions or alternatives and is thus heavily application-dependent, as e.g. in the case of joint document editing [10]. However, process-oriented support is possible without defining sophisticated mechanisms for merging complex results.

Our communication mechanism has been originally designed and developed for process-oriented multi-user support in face-to-face training situations [11]. Human-human cooperation in a training setting is typically not product-oriented. In our application scenario we have the trainer or moderator on an electronic blackboard ("Xerox Liveboard" cf. [7]) networked with local student workstations. Although the trainer will also use the electronic blackboard to present interactive multimedia material, from a communication point of view our focus was on supervising and supporting group exercises including public explanations and discussions of examples. In this respect, we think training is only one example of a *guided group meeting*. Here it is

important to be able to *share objects* (e.g. between the liveboard and a local workstation, or between students' workstations) temporarily. We call this kind of partial and temporary synchronization "coupling". *Coupling* is applicable to objects that exist in different individual workspaces, even if they have not been identified or linked to each other a priori. Practically, we have implemented the coupling mechanism as an extension of a UI toolkit based on OSF/Motif [15]. This toolkit, called CENTER [13,24], is sufficiently general to support all the standard features of graphical UIs and also provides an interactive builder for users who are not experienced programmers (e.g. UI designers). Synchronization is achieved on the UI level by modifying the event-callback mechanism using a central server that receives high level events and broadcasts the callbacks to a set of coupled objects.

In the next section we discuss implementation models of multi-user applications. Section 3 describes our flexible communication/collaboration model. Section 4 and the final section discuss the current implementation and applications as well as future directions of our work.

2 Implementing multi-user systems

2.1 General architecture

The main feature distinguishing between different architectures for distributed multi-user systems is the degree of centralization vs. replication of functionality at different levels (roughly: I/O, dialogue, application). In the following, we adopt the basic terminology introduced by Greenberg et al. [8], but we will refine the global notion of "program" by distinguishing between user interface and functionality (of the application).

Multiplex architecture

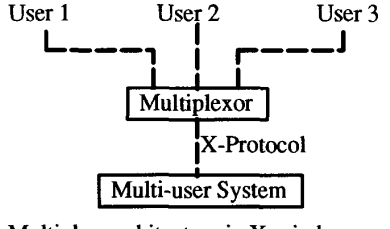

Figure 1. Multiplex architecture in X-window environment

A first type of multi-user systems employs a *single-instance* architecture (also called "multiplex architecture") in which several users interact simultaneously with a single centralized application instance from several workstations. These systems, such as the clients in *shared* X Windows Systems [1,14,21] as shown in Figure 1, are based on the assumption that collaboration among a limited number of users is not long-distance, not strictly synchronous and does not contain many time-consuming tasks. The shared window system multiplexes the application's output to each participant's display and dispatches user events sequentially in turn. In this sense, only the I/O level of the user interface is replicated. No local management is required for each participant. This architecture does not fit in with the requirements of highly parallel processing and real-time response.

Replicated architecture

To overcome the problems of the multiplex architecture, *replicated architectures* [2,3,6,8,14] have been widely used to implement synchronous multi-user systems. In the replicated architecture, display areas, data objects and (partial or full) operations of an application are replicated for each user. Thus, many operations can be performed locally. Synchronization among various copies of data objects and among the different user's display areas is accomplished by re-executing the other user's actions. To guarantee that a user's actions arrive in the same order at all sites, several approaches are adopted. In centralized control mode, users send their requests for operations to the controller, and then the controller broadcasts these operations to all users. In timestamp (or dependency-detection) approach [6], each user action is timestamped in order to detect conflicting actions.

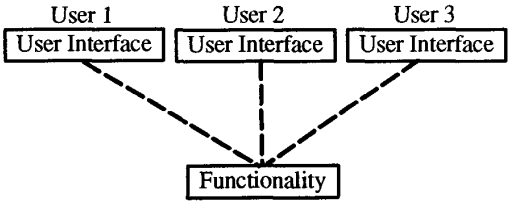

Figure 2. UI-replicated architecture
(Partially replicated architecture)

From a UIMS point of view, a multi-user application system is divided into the user interface and the application functionality in very much the same way as a single-user system [19]. Depending on which components are replicated for each user, the replicated scheme is divided further into *partially* and *fully replicated architectures*. In the partially replicated architecture (cf. Figure 2), only the shared user interface is copied for each participant, while in the fully replicated architecture as shown in Figure 3 both the user interface and semantic operations are copied for each user.

In the partially replicated multi-user applications (e.g. a multi-user card game [17,18]), the unique semantic component and the individual user interfaces run in separate processes. The Suite system [4,5] is a general tool that supports the construction of UI-replicated applications (on both I/O

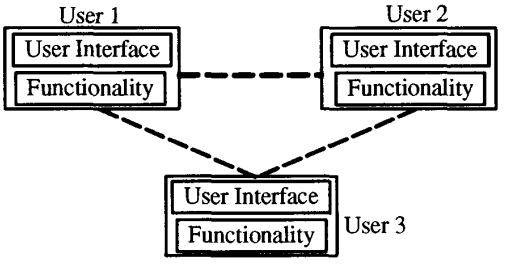

Figure 3. Fully replicated architecture

and dialogue levels). The advantage of this approach is that it avoids the complications of concurrency control for a distributed application. Concurrency on the user interface level is gained through buffering and sequential execution of those user actions that affect the semantics of the application. If such a semantic action is time-consuming, it may of course block the execution of other user's actions for an unacceptably long period of time. If such cases are frequent, the UI-replicated architecture is not appropriate. A fully replicated architecture as shown in Figure 3 avoids this run-time problem, and additionally, it facilitates the design of multi-user programs. For example, a multi-user program can be easily derived from a single-user version built with existing standard tools.

The COSOFT architecture

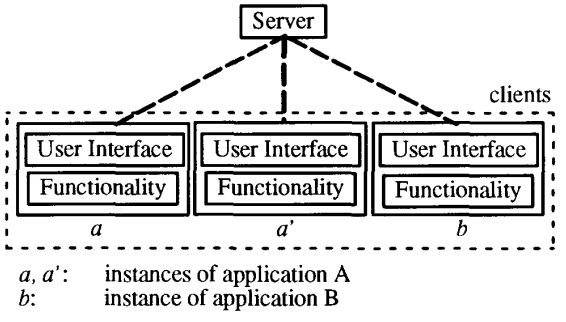

Figure 4. COSOFT architecture

Our model has been implemented as a server-client architecture as shown in Figure 4. It is a fully replicated architecture since every application instance can communicate with any other. The "normal" way of communication is through UI objects. A central controller (the server) coordinates the communication and access control. A centralized database residing on the server consists of four categories of data: the access permissions, the registration records, the historical UI states, and the lock table. Access permissions are three-valued tuples with user ID, UI state identifier, and access right category. Registration records store the application instance as well as participant information such as application instance identifier, host name, and user name, etc. The historical UI states backup the UI states which have been overwritten when *synchronizing by state* (cf. subsection 3.1) was applied, and provide the possibility of undoing/redoing user's actions. The lock table guarantees that actions occur serially within each group of coupled objects.

2.2 Flexible communication and collaboration

Dimensions of flexibility:

Complete vs. partial coupling. In most multi-user systems, different users share the content of their display and work on consistent data objects using equivalent operations. *Complete coupling* implies that the complete view of the application is coupled with the views of other participants after a new user joins a work group. This feature restricts the collaboration to scenarios in which a common task is being performed by all group members using the same application program, e.g. a drawing tool or a multi-user document editor. This excludes many real-world situations in which different users may use a variety of programs to collaboratively accomplish their (sub-)tasks (as e.g. in a collaborative teaching/learning environment). Here, it is not necessary to couple complete views, but only *partial* displays may be synchronized among several participants.

Continuous vs. periodical synchronization. To guarantee consistency, participants in most CSCW systems are usually *coupled* continuously, either in loosely-coupled or in tightly-coupled mode (e.g. rIBIS [20] and SEPIA [9]). Participants are not allowed to *decouple* from others, work alone for some time, and then join the work group again, since continuous *synchronization-by-action* synchronization is required to maintain consistency.

In both the multiplex and the replicated architectures, synchronization of data and shared views is achieved by broadcasting and multiply executing user actions: either low-level events (such as keystrokes or mouse motions) or aggregated actions. The principle of such *synchronization-by-action* allows each user to see the other's actions. However, we believe that there is another type of environments in which participants may work in parallel more independently, also not necessarily with same tool. Here, collaboration can be based on *periodical* updates, e.g. by passing *snapshots* of parts of one user's work to another user. Simple techniques for supporting such kinds of communication-oriented collaboration have recently been suggested by Patel and Kalter [16]. This kind of flexible communication and information exchange is essentially based on a *partial synchronization-by-state* approach. To share information with another user, some sort of remote copy operation is used. Often, it is not even necessary to perform an automatic merge operation on the copy and the corresponding unit in the receiver environ-

ment. Also, the potentially costly re-execution of actions is avoided. However, such interactions typically have to be *negotiated* since the transmitted information has to be requested and the transfer has to be confirmed by the owner. This is not appropriate for communications with high frequency of information exchange. Therefore, we believe that both *synchronization-by-state* and *synchronization-by-action* have to be supported in a flexible way.

Homogenous vs. heterogeneous application instances. Collaboration among participants in a CSCW system is usually only supported for a set of instances of one application. In other words, there is no support for collaboration between instances of different applications. The major reason is the *application-dependent* communication protocol, i.e. messages (types, formats) that can be exchanged are specific to the application. Assuming that we want to distribute also different types work between several participants in a multi-user environment, or at least that users may have somewhat different working environments, it is indeed desirable to support (partial) synchronization between functionally different applications. In our training scenario coupling between *heterogeneous applications* (i.e. instances of different types of applications) is frequently needed since often the trainer's application may differ significantly from the students' version, even in the same general domain.

Static vs. dynamic population. With respect to the "population" dimension, most CSCW systems support *global sharing* or *static grouping*. In *global sharing* mode, each participant has to couple with the rest of the work group, while in *static grouping*, participants can be coupled according to sub-groups which are specified before starting a session. In our approach, we support *dynamic grouping*, in that we allow each participant to couple selectively with other participants. These groups connections can be defined at runtime.

Comparison of approaches

Our approach is based on sharing UI objects, thus it does not require application-specific communication mechanisms. This distinguishes our model from the multi-user system for a specific application. For example, DistEdit [12] is designed for implementing multi-user document editors. GroupDesign [2] is for multi-user sketch drawing. These systems do typically not support partial synchronization and only very limited dynamic coupling (e.g. late joining but no decoupling). Some support loose coupling (i.e. delayed synchronization through version management).

The following table shows a comparison of application-independent synchronization approaches in multi-user environments:

Features / Systems	replication	dynamic coupling	heterogeneous applications	easy upgrade
SharedX	I/O	limited		✓
Rendezvous	dialogue	✓		
Suite	dialogue	✓		
GINA	full	✓		
COSOFT	full	✓	✓	✓

Our approach is based on a fully replicated architecture. In this respect it differs from both Rendezvous system [17,18] and the Suite system [4,5]: they use a UI-replicated architecture where a single application is linked to multiple user interfaces through which users communicate with each other. Also, since in our approach the application program is completely separated from the communication mechanism, it is very easy to upgrade existing single-user applications to multi-user ones.

We support communication/collaboration between *heterogeneous applications*. This is the main difference with the cooperative version of GINA [3], which concentrates on maintaining and handling different versions or alternatives from different sources (mainly for undo/redo purposes). Similar to our approach, GINA uses a replicated architecture based on shared UI objects.

Also the different implementations of shared X Windows protocols (e.g. SharedX or XTV [1,14]) in the multiplex architecture (cf. Figure 1) are an interesting reference point. Here, replication is confined to the I/O level. With shared X window systems, it is easy to turn single user applications into multi-user ones. The basic unit shared between users is a window which disappears in the personal environment when the joint session is left (if possible). In our approach of flexible coupling/decoupling of UI objects, these will not cease to exist when being decoupled so that coupling can be used to transfer information between environments.

3 Communication between heterogeneous application instances

In this section, we will introduce the technical features of our approach and its implementation. After introducing some terminology, we describe our synchronization procedures, including the copying of user interface states as well as tight coupling, then discuss issues of compatibility as requirements for coupling and copying, and finally describe a mechanism for extending the standard communication protocol.

A *primitive UI object* is an instance of a pre-defined UI object type (e.g. form, button, menu, etc.). It encapsulates low-level events and provides high-level interactive techniques. A set of attributes is defined for each type of UI objects. In an application instance, a UI object is globally represented (across application instances) as a pair:

< *instance_id*, *pathname* >, where *instance_id* is the application instance identifier, and *pathname* is the hierarchical name of the UI object. User interface objects in an application instance are organized as a tree along the *parent/child* relationship (represented in a specific attribute). A **complex UI object** is a hierarchically structured collection of primitive UI objects. In this paper, unless it is explicitly excluded, we use UI object for both primitive and complex UI objects. The **state of UI object** is the set of attribute-value pairs of this object. The set of attributes of an object only depends only on the object type.

A *couple link* is a directed arc from the source UI object to destination UI object, labeled with the application instance identifier, which creates the link. The **couple** relation C consists of all pairs of UI objects connected by a couple link. To compute the set of objects $CO(o)$ connected to or coupled with a given object o, we use the transitive closure of C.

3.1 Synchronization by UI state

To synchronize a new participant with existing participants in a group, most multi-user drawing or editing systems copy the identical views of other participants for the new participant. After joining a session, the new participant is synchronized with others by re-executing actions.

However, relaxations on the dimensions of time and space in our communication model require the synchronization not only of specific display areas, but also of the application-specific internal data "behind" certain surface objects.

Copying UI state

UI objects can be synchronized with other objects by transferring the UI state in two different modes: *active* or *passive synchronization*. With the active synchronization (implemented as a function *CopyFrom*) which represents the paradigm of *monitoring another person's activities*, an application actively requests the state of UI objects in other instances, and updates its own state with the obtained state. The passive synchronization (implemented as a function *CopyTo*) indicates a scenario in which one person *lets another person see his or her work*. An application passively receives the UI states of other application instances and uses them to update its corresponding UI state.

Each UI object type has its characteristic set of attributes. To synchronize two objects not all of these attributes have to have identical values (e.g. two text input fields may have with different size and fonts, but just share the same content). In our system, a set of relevant attributes is predefined for any type of couplable UI objects. *Relevant attributes* are those that have to be shared (i.e. made identical) when instances of these types are coupled. So far, we did not feel a practical need for handling these relevance definitions flexibly as e.g. supported by Suite [4,5].

An additional primitive (*RemoteCopy*) is provided for facilitating to remotely copy complex UI objects from the first application instance to the second instance into a third application instance.

Synchronizing semantic state

Copying a complex UI object's state only guarantees the consistency on the UI level. It neither synchronizes the semantic data (i.e. internal application data) associated with these complex UI objects, nor updates the related UI objects – these related UI objects have been modified by the actions occurred on the *dominating (i.e. copying)* or *dominated (i.e. copied)* complex objects during their period of being decoupled. Either of the inconsistencies is difficult to solve. One approach is to record all actions occurring on the (copied and copying) complex objects while they are decoupled, and then re-execute these actions when they coupled. Another approach is to copy not only the complex UI object's state, but also the associated semantic data, as well as the modifications on related UI objects. The first approach is expensive, especially for long period of decoupling. A simplified version of the second approach has been employed in our model.

To keep UI and semantic states consistent, application programmers have to define two functions for each semantic data structure to *store* and *load* application data. They are automatically invoked in the dominating and dominated application instances respectively when the state of a UI object is copied. Fortunately, by following some programming conventions (e.g. by not defining application data that can be accessed from various UI objects, or by attaching all relevant application data to UI objects), application programmers can reduce the complication of these functions, or even avoid them completely.

3.2 Synchronization by multiple execution

After two complex UI objects are initially synchronized by copying the UI state, synchronization among coupled UI objects is accomplished by re-executing actions on these objects. Whenever an event occurs on one of the coupled objects, this event packed with some parameters is sent to the server. Then the server broadcasts this message to the application instances where it is unpacked and re-executed.

To serialize the actions occurring on the objects in a coupling group, we adopt a simple *floor control* strategy. Whenever an event occurs on one of the objects in a group of coupled objects, the rest of other objects in this group are *locked*. They are *unlocked* when the processing of this event is completed. Actions on locked objects are disabled. Such a locking mechanism might become costly if the events were fine-grained, such as cursor movements or the typing of

single characters. However, in our model, most events are high-level callback events of UI objects.

Algorithm: Multiple execution.

Assume event *e* to occur on UI object *o*. Let *CO(o)* be the set of the UI objects that have been coupled with *o*.

```
o = < my_id, my_pathname >;
/*lock control*/
LCO = { } stores the objects which have been locked.
for each UI object < id, pathname > ∈ CO(o)
  if < id, pathname > has been locked in the server
        LockFailed = True;
        break;
  else  disable object < id, pathname>;
        lock < id, pathname > in the sever;
        LCO = LCO ∪ {< id, pathname >};
if LockFailed is True
        /*undo locking*/
        for each < id, pathname > ∈ LCO
          unlock < id, pathname > in the server;
        undo syntactic built-in feedback of the event e;
else    for each o' = < id, pathname > with o' ∈ CO(o)
          simulate the feedback of e;
          execute callbacks of the event e on object o';
        /*release lock*/
        for each <id, pathname > ∈ CO(o)
          unlock on < id, pathname > in the server;
          enable UI object < id, pathname >.
```

In a group of coupled objects, the coupling information is replicated for each object (to be completely available locally). For creating a new couple link from a source UI object o_1 to a target object o_2 in a destination application instance, a "link generator" sends this couple request to the server, who then broadcasts it to the destination environment where the link from o_2 to o_1 is created. Now, objects already connected to o_2 are added to the list of targets, and objects already connected to o_1 are added to the source, thus computing the complete transitive closure of the couple relation.

When a couple link is removed (decoupling), it is deleted from all connected objects that belong to the same group. The decoupling algorithm is applied automatically when a UI object is destroyed or an application instance terminates.

Furthermore, to facilitate the use of the couple/decouple mechanism between different applications for the programmer, we provide *RemoteCouple* and *RemoteDecouple* functions which allow a third application instance to couple objects in remote instances.

3.3 Synchronization between different UI objects

Not only UI objects of the same type can be coupled or copied, already a certain compatibility (to be defined) of objects types is sufficient. Several "coupling modes" are supported in our model:

- UI objects of the same type in homogenous application instances, including the case of two objects coupled within the same application instance;
- UI objects of the same type in heterogeneous instances;
- different types of UI objects in homogenous or heterogeneous application instances;

Primitive objects are **compatible** if they are of the same type or if a correspondence relation is declared for their relevant attributes (i.e. each relevant attribute of o_1 has a corresponding attribute o_2 that can be used for copying or coupling).

Let O_1 and O_2 be two complex UI objects, each represented as the set of its direct components (i.e. root object and direct children). Then O_1 and O_2 are said to be **structurally compatible** (*s-compatible*) iff there is a *one-to-one mapping* σ between O_1 and O_2 so that:

For any o in O_1, $\sigma(o)$ is either directly *compatible* with o (in case o is primitive), or $\sigma(o)$ is *s-compatible* with o.

Of course, calculating σ over several levels of nesting may be costly in practice. Sometimes it can be pre-defined, or certain heuristics have to be used to avoid combinatorial explosion.

The above definition for the compatibility between complex objects is restricted to objects with *identical structure*. We introduce *destructive merging* and *flexible matching* as approaches to copying or coupling two complex objects that are not structurally compatible.

Destructive merging is applied when two complex objects with non-identical structures are synchronized by copying UI state. Not only the attribute values, but also the structure of the dominating complex object is copied to the dominated object. Copying structure includes *destroying objects* of the dominated complex object if they conflict with the dominating complex object, and *creating objects* if they do not exist in the dominated complex object.

The *flexible matching* approach identifies identical substructures between two complex objects when they are coupled or synchronized by copying. Differing substructures are conserved by merging.

3.4 Extension of the communication protocol

The unit of coupling in our model is a UI object, rather than the application context as in most CSCW systems. Thus, we have a application-independent communication protocol which can support collaboration between heterogeneous application instances. However, this common protocol restricts the scope of sharing information since it is only defined for UI objects and actions associated with these (such as pressing of push button object, entering and deleting of characters) but not for other semantic units or functions.

To define application-specific communication protocol, we provide a primitive (*CoSendCommand*) which enables pro-

grammers to define their own protocols. An application can call this primitive to send a command (i.e. a symbolic name of a function) together with a packed message to other instances. In the receiver instances, a function (corresponding to the command) is defined to unpack and interpret the message. From a programmer's point of view, it is important that these messages are directly handled by our communication server.

4 Practical applications and experience

So far, we have practically used the described communication model, implemented as an extension of the CENTER Toolbox, in two application areas: The application we had in mind when designing the mechanism was face-to-face classroom communication focusing on flexible coupling between the teacher's presentation screen and student workstations. The name of our project and system is derived from this application, called COSOFT ("COmputer Support for Face-to-face Teaching") [11]. Here, a first version has been completed. Additionally, we have extended an interface for database retrieval called TORI ("Task-Oriented database Retrieval Interface") [24] to be *cooperative*. Due to the limited space, we can only state some essential "lessons learned".

Making the TORI retrieval interface cooperative was an interesting engineering exercise, since TORI itself is a rather complex piece of software. The UI objects that we chose for coupling were query and result forms that TORI generates from high-level descriptions. Synchronization takes place on menus for selecting comparison operators (e.g. "substring", "like-one-of", etc.), on text input fields associated with attributes, and on menus for selecting a certain view (i.e. a set of query attributes). Result forms have a similar attribute-value structure, but offer different operations, e.g. for using result data to partially instantiate new query forms. Also these operations are synchronized. We also synchronize the invocation of queries, which implies that a query will be potentially re-executed several times. From a performance point of view, one might argue that it would be preferable to evaluate the query once and share the results. But this goes beyond a simple sharing of UI objects. This shows the inherent limitations of the UI-based approach. On the other hand, multiple evaluation is more flexible in that it allows queries to be different; e.g. only some query attributes may be shared between users. Also, queries can be sent to different databases. The main amount of work went into the provision of an interactive interface to coordinate a joint retrieval session between several users. The whole conversion of TORI was done in a few days during one week by one person.

In our COSOFT classroom situation, we have two types of interactive, multimedia-based working environments: the teacher's presentation environment that runs on the electronic blackboard; and the local student environments that typically offer exercises and, in some problem domains, also local context-sensitive help. Of course, materials for presentation and for exercises on a given subject are closely related to each other, even partially identical (e.g. they may use the same simulation windows or function displays). In our current scenario, communication through coupling takes place when the teacher or trainer wants to discuss in public a particular solution provided by one or several students. This is typically initiated either by a direct request sent by a student or by an automatic message generated by an intelligent demon. These messages are buffered and can be inspected by the teacher. For initiating a joint session, we provide an interactive interface for a procedure that essentially consists of *(1) selecting a student* (or group of students) with which the teacher's environment is to be coupled from a graphical menu that shows the classroom situation in stylized form, and *(2) selecting the UI objects to be coupled* from a (potentially simplified) graphical representation of the student's environment. Application-specific correspondences between elements of the student's and teacher's environments have to be declared on beforehand. (Otherwise, we would have to select *both* kinds of objects explicitly.) Dynamic coupling and decoupling is based on the remote operations *RemoteCouple/RemoteDecouple* (cf. section 3.3) since it is initiated from outside the respective applications.

To make a pair of COSOFT applications (teacher's/ student's versions) couplable, no more programming than inserting a statement to register the application with the server is needed. The application specific selection menu can be created interactively. Here, also the internal names of objects to be coupled are specified. As with TORI, the most of the work went into providing the interactive control mechanism which, in this case is even more general since it can be used for a variety of COSOFT applications.

In the COSOFT environment it has turned out that partial coupling can be very efficient since it allows for *indirect coupling*: Often is sufficient to couple UI objects that contain information (e.g. certain input fields for parameters, function terms, or other data) from which the content or behavior of other components can be *generated*. For these dependent objects (e.g. simulations or graphical displays), direct coupling might be much more costly.

5 Conclusion

The communication model presented in this paper relaxes the WYSIWIS principle in a new dimension: *application-dependency* (in addition to time, space, and population), i.e.

it allows for collaboration between *heterogeneous application instances*. To achieve this, a common, application-independent communication protocol situated on the UI level has been defined. Moreover, our model also enables application programmers to extend this protocol. Another benefit of such an application-independent communication protocol lies in supporting the development of new multi-user interfaces and as well as the easy conversion of single user interfaces into a multi-user ones.

In its pure form, the presented model only allows for *sharing information in so far as it is directly accessible through user interface objects and their attributes*. This excludes the use of internal semantic structures of the application, such as e.g. internal structures of text documents, even if they are being displayed in a window, and even moreso the use of internal data that are not directly related to any visible object on the surface. Currently, it is left to the application programmer to extend the initial synchronization-by-state between components of the user interface to include such internal states. However, this task may be supported by some standard extensions for typical applications. Also, initialization procedures for making complex, hierarchically nested UI objects compatible will have to be refined.

Acknowledgement:

Nelson Baloian and Frank Tewissen made important contributions to developing the COSOFT framework and applications. They also participated in many fruitful discussion and thus influenced our suggestions. We want to express our gratitude for these contributions.

References

[1] *Abdel-Wahab*, H.; *Jeffay*, K. (1992). Issues, Problems and Solutions in Sharing X Clients. Technical Report TR92-042, University of North Carolina, Chapel Hill, 1992.

[2] *Beaudouin-Lafon*, M.; *Karsenty*, A. (1992). Transparency and awareness in a real-time groupware system. In: Proceedings of ACM UIST '92 (Monterey, CA, November 1992), pp.171-180.

[3] *Berlage*, T. and *Genau*, A. (1993). From undo to multi-user applications. In: Proceedings of INTERCHI'93 (Amsterdam, Netherlands, April 1993), pp.213-224.

[4] *Dewan*, P. and *Choudhary*, R. (1991). Flexible user interface coupling in a collaborative system. In: Proceedings of ACM CHI '91 (New Orleans, April/May 1991), pp.41-48.

[5] *Dewan*, P. and *Choudhary*, R. (1991). Primitives for programming multi-user interfaces. In: Proceedings of ACM UIST '91 (South Carolina, November 1991), pp.69-78.

[6] *Ellis*, C.A.; *Gibbs*, S.J. and *Rein*, G.L. (1990). Groupware: Some issues and experiences. Comm. ACM, 34(1), pp.39-58.

[7] *Elrod*, S.; *Bruce*, R.; *Gold*, R.; *Goldberg*, D.; *Halasz*, F.; *Janssen*, W.; *Lee*, D.; *McCall*, K.; *Pedersen*, E.; *Pier*, K.; *Tang*, J.; *Welch*, B. (1992). Liveboard: A large interactive display supporting group meetings, presentations, and remote collaboration. In: Proceedings of the ACM CHI '92 (Monterey, CA, May 1992), pp.599-607.

[8] *Greenberg*, S.; *Roseman*, M.; *Webster*, D.; *Bohnet*, R. (1992). Human and technical factors of distributed group drawing tools. Interacting with Computers, 4(3), pp.364-392.

[9] *Haake*, J.M. and *Wilson*, B. (1992). Supporting collaborative writing of hyperdocuments in SEPIA. In: Proceedings of ACM CSCW '92 (Toronto, Canada, November 1992). pp. 138-146.

[10] *Haake*, A. and *Haake*, J.M. (1993). Take CoVer: Exploiting version support in cooperative systems. In: Proceedings of InterCHI' 93 (Amsterdam, Netherlands, April 1993), pp.406-413.

[11] *Hoppe*, H.U.; *Baloian*, N and *Zhao*, J. (1993). Computer-support for teacher-centered classroom interaction. In: Proceedings of 1993 International Conference On Computers In Education (Taipeh, Taiwan, December 1993). pp. 211-217.

[12] *Knister*, M.J. and *Prakash*, A. (1990). DistEdit: A distributed toolkit for supporting multiple group editors. In: Proceedings of ACM CSCW'90 (Los Angeles, CA, October 1990), pp.343-355.

[13] *Kostka*, B.; *Tschumakoff*, S.; *Zhao*, J. (1992). The IMIS User Interface Toolbox: Programmer's Guide. Sankt Augustin: GMD, February 1992 (GMD-Studie, No. 201).

[14] *Lauwers*, J.C. and *Lantz*, K. (1990). Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window system. In: Proceedings of ACM CHI '90 (Seattle, WA, April 1990), pp.303-312.

[15] *Open Software Foundation* (1990). OSF/Motif Programmer's Guide. Prentice Hall, Englewood Cliffs, NJ, 1990.

[16] *Patel*, D. and *Kalter* S.D. (1993). Low overhead, loosely coupled communication channels in collaboration. In: Proceedings of Third European Conference on Computer-Supported Cooperative Work (Milano, Italy, September 1993). pp.203-218.

[17] *Patterson*, J.F.; *Hill*, R.D.; *Rohall*, S.L.; *Meeks*, W.S. (1990). Rendezvous: An architecture for synchronous multi-user applications. In: Proceedings of ACM CSCW '90 (Los Angeles, CA, October 1990), pp.317-328.

[18] *Patterson*, J.F. (1991). Comparing the programming demands of single-user and multi-user applications. In: Proceedings of ACM UIST '91 (South Carolina, November 1991), pp.87-94.

[19] *Pfaff*, G.E. (Ed.) (1983). User Interface Management Systems. Berlin et al.: Springer-Verlag, 1983.

[20] *Rein*, G.L. and *Ellis*, C.A. (1991). rIBIS: A real-time group hypertext system. Int. J. of Man-Machine Studies, 34(3), pp.349-368.

[21] *Scheifler*, Robert W. and *Gettys*, Jim (1986). The X window system. ACM Transactions on Graphics, 5(2), pp.79-109.

[22] *Stefik*, M.; *Bobrow*, D.G.; *Foster*, G.; *Lanning*, S.; *Tatar*, D. (1987). WYSIWIS revised: Early experiences with multi-user interfaces. ACM Transactions on Office Information Systems, 5(2), pp.32-47.

[23] *Stefik*, M.; *Foster*, G.; *Bobrow*, D.G.; *Kahn*, K.; *Lanning*, S.; *Suchman*, L. (1987). Beyond the calkboard: Computer support for collaboration and problem solving in meetings. Comm. ACM, 30(1), pp.32-47.

[24] *Zhao*, J.; *Kostka*, B.; *Müller*, A. (1993). An integrated approach to task-oriented database retrieval interfaces. In: R. Cooper (ed.). Interfaces to Database Systems. London et al.: Springer-Verlag. pp.56-73.