An Implementation Model for Collaborative Applications

Mauricio Cortés[1]
Bell Labs, Lucent Technologies
Department of Multimedia Applications
101 Crawfords Corner Rd. Room 4F-625
Holmdel, NJ 07733
email: mcortes@bell-labs.com

Prateek Mishra
State University of New York at Stony Brook
Department of Computer Science
Stony Brook, NY 11794-4400
email: mishra@cs.sunysb.edu

ABSTRACT

A major challenge in building groupware systems is to provide support for control and coordination of users actions on shared resources. This support includes the maintenance of the current state of the collaborative multi-user environment such as control of group interaction rules and coordination of users actions or tasks.

We propose an extension of the visual presentation/underlying data model currently followed when developing interactive single user applications. We claim that groupware systems require two additional components: user-related data and group interaction rules. The former component maintains information about active users, their roles, and privileges. While the latter keeps the state of the current collaborative environment to control and coordinate user actions. Furthermore, our approach allows developers build each system component separately, promoting the decomposition of the application's computational objects and its collaborative environment specification.

1. This work was developed while the first author was affiliated with State University of New York at Stony Brook.

INTRODUCTION

A computer program is an abstract model of a given problem. In particular, a collaborative program needs to model the interaction between two or more persons sharing information and working on a common task. A group of software engineers debugging an application or physicians examining X-ray images are examples of groups of people working together under some interaction environment. Although, these two collaborative environments seem completely unrelated, the interaction among a group of engineers might follow similar rules (etiquette) to those found in some medical settings.

In practice, groups of collaborators create their own working environment. For example, Watson et.al. [18] report that the cultural background of each team member can influence the way the entire workgroup interacts. Other factors, such as the number of participants and the individual or group goals can affect the working environment. Furthermore, it has been found that these group settings can vary from meeting to meeting, and even within an ongoing collaborative session [14]. Therefore, customizable applications need to be developed in order to cope with individual [11] and group needs.

The development of groupware systems must address additional issues not present in single user applications. For instance, the maximum number of active users at a given time, number of telepointers or remote sprites, level of support for user awareness, registration protocols, and maintenance of user roles are new aspects that need to be considered in collaborative systems. We will argue in this paper that these issues must be addressed as a separate system component, namely control and coordination component, in order to allow the construction of flexible groupware system.

The following example illustrates the need of building flexible and adaptable groupware applications. Screen dumps of an X-Ray image browser are depicted in Figure 1. The participant's name is shown in the lower right corner of the screen. Notice that each user is identified by a gray shade, shown as the background of the user's name. Users can navigate through a set of images using the next/previous buttons shown in the upper right corner. Users B and C can request a single shared telepointer by pressing the arrow button. A and B can annotate an image by using a shared pencil.

Figure 1 shows the following user activities: user A is examining the sixth image, as shown in the upper left corner of Figure 1.a., while users B and C are both looking at the fifth image (see Figure 1.b.). User B owns the telepointer that is also displayed at C's screen. Meanwhile, user A is waiting for the next image to be displayed. Notice that User A drew an annotation on the fifth image number,

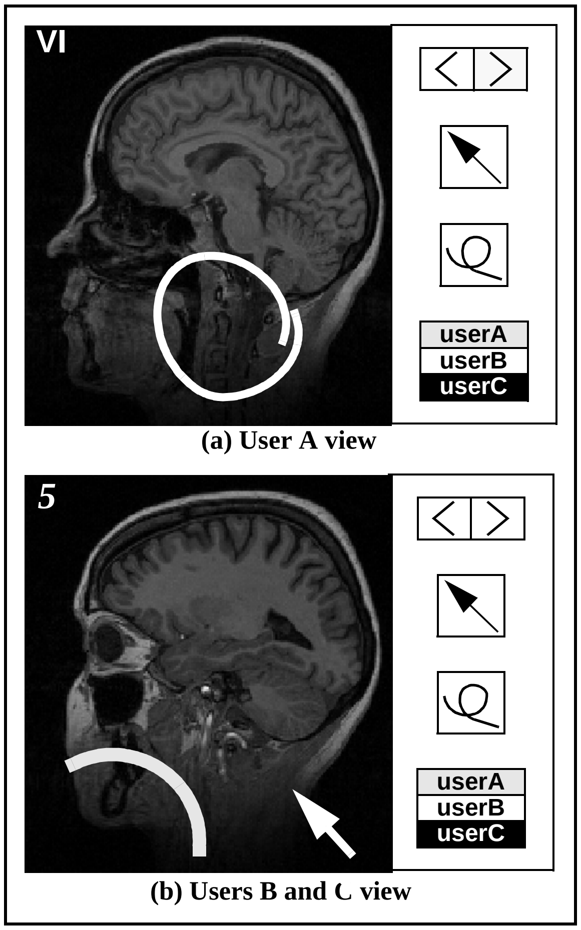

Figure 1. CT/MRI Collaborative browser

while User B added an annotation on the sixth image. User A has customized her/his screen by changing fonts and numeric data presentation, while users B and C are sharing the same view at this point in the session.

Such collaborative image browsing tools can be used in several scenarios such as virtual lecture halls, patient consultation (e.g. radiologist/patient), group consultation between two or more physicians (e.g. surgical procedure), or a radiology conference. Formally, a collaborative scenario is an instance of the collaborative environments mentioned earlier. These scenarios differ from each other on issues such as users' data accessibility, group interaction rules, and users visual presentation. For instance, in a patient consultation environment, a privileged user (i.e. physician) only allows the other user (i.e. patient) to observe his/her own medical images. In contrast, a group consultation scenario would require the same access rights for each user (i.e. physicians).

In addition to access control issues, the group of users should be able to define the way they want or need to interact with each other through the set of shared resources. Notice that in groupware applications, user actions (e.g. navigating through the set images) are applied to a shared resource. These user actions can potentially update the application's visual presentation shown in one or more user screens. For example, navigating through a set of images and creating image annotations are plausible user actions. As such, a user action can interfere with another user action if they are applied simultaneously to the same shared resource. In order to coordinate these user actions, some coordination mechanisms must be defined, reflecting the way the group of participants wants to run their meeting.

We believe that it is virtually impossible for programmers to foresee all possible mechanisms that users might need at any given time. Instead, programmers should provide a set of tools so that users can define their own agenda. This implies that groupware applications should include a customizable coordination component, such that users working together can tailor the application to their needs. In the same way that single user applications allow users to personalize the systems's visual presentation (e.g. background colors), groupware systems should allow the specification of human-human interaction rules describing how the group of users want to work.

Notice that the medical scenarios mentioned earlier allow users to collaborate both in synchronous (same time) and asynchronous (different time) mode. For instance, group consultation can take place in real time when physicians interact with each other at the same time or they can interact through electronic mail messages at different times. These modes reflect the way participants collaborate with each other and should also be captured by the proposed coordination component.

Although many researchers have suggested a clear separation between synchronous and asynchronous collaboration modes, the core functionality of these modes remains the same from a programmer's perspective. For example, the actual functions that display medical images and allow users to navigate through the patient's medical records can be used for both modes. Thus, programmers should not have to develop the same functionality for each mode. If there is a clear separation between these computational functions (e.g. show_image()) and group interaction issues (i.e. coordination mechanisms), developers should be able to reuse their programs

In this paper, we propose an extension of the visual presentation/underlying data architecture currently followed in the development of interactive single user applications. We claim that groupware systems require two additional components: user-related data and group interaction rules or controller component.

We will assume that the application's visual presentation and underlying data manipulation functions can be developed in some traditional programming language (Pascal, C++). These functions will not include control or coordination information. On the other hand, the new components will include user information to control their collaborative actions and the interaction rules that should be followed by the working group.

We claim that each of these components can be implemented separately. This approach has several advantages from the programmer's and user's standpoints. Maintaining the clear separation between visual and underlying objects allow multiple views to be defined for a given set of underlying computational objects. Furthermore, multiple interaction rules can be defined for a given pair of computational and visual objects. In this way, working groups could use the appropriate interaction rules to meet the coordination functionality required by their collaborative environment.

The rest of this paper is organized as follows. In section 2, we give an overview of our groupware implementation model. Sections 3 and 4 gives a brief description of user and coordination components, respectively. The next section briefly describes a coordination programming language that allow developers to specify the coordination component. Finally, in section 6 we present future work and some concluding remarks.

IMPLEMENTATION MODEL

A system is recursively defined as a group of entities and the interaction between them, where each entity can be a system in itself. The interaction among these entities must specify, among others, control and coordination rules between its macro entities and within each complex entity or subsystem. The external behavior or user functionality of a system is given by the visible state of its components and interaction capabilities between them.

In particular, our groupware system model contains the following major components: visual and underlying computational objects, group of active users, and group interaction rules. Visual (e.g. pushbutton) and underlying computational objects (e.g. raw medical images) model physical or virtual resources, which can be manipulated by some set of program functions. Information such as the user address (e.g. email address), and access privileges should be kept separated from the visual and data components. Finally, the interaction rules must deal with possible conflicts that can arise between any combination of these major components, and within each component. Table 1 shows an example of a real-time shared text editor modeling a synchronous peer environment with few interaction constraints.

Component	**Data**	**Functionality**
Computational Objects	document	insert, delete
Visual Objects	cursor position, fonts	cursor functions (up, down)
Users	name, userid	add_user, delete_user
Control and Coordination Rules	number of users number of telepointers	Asymmetric view, any user can join, updates must be notified immediately

Table 1. Groupware Components

Formally, the state of a collaborative applications can be represented by a 4-tuple S = (D,V,U,C), where D, V, U, and C are sets of data objects, visual objects, users, and interaction rules, respectively. From a multi-user perspective, D and V should only include shareable objects, i.e. resource instances that can be manipulated directly or indirectly by some group of users. For example, push-button should be added to the set of visual objects only if its attached function (e.g. callback function) has some effect on other users' state.

It is important to point out that the widely accepted implementation model that decouples computational objects and visual objects for single user applications is a special case of our approach. Let U be a singleton that only includes the single user running the application. Since no conflicts can exist with a single user, we assign an empty set to C, the set of interaction rules. Thus, single user applications can be implemented using our implementation model.

Collaborative environments evolve during its lifetime. This implies that S-tuple is a snapshot of the collaborative state at any one time. In order to reflect these environmental changes, we introduce a time variable t into our model where S_t represents the collaborative state at time t. Thus, the notion of session history H can be defined as a sequence of S-tuples $H=[S_{t0},S_{t1}, \dots ,S_{tn}]$, capturing the dynamic nature of real world meetings ranging from its creation time (t_0) to session's termination time (t_N). Any time t between session creation and termination is referred here as session time. Additionally, we define *session size* as the cardinality of the set of users.

Recall that Ellis et.al.[7] defined a groupware session as the time interval where participants can interact with each other manipulating some shared objects. Notice that we have extended Ellis' definition since under our approach a session is formed by an actual group of interacting users that can apply some common program functions to actual shared data following a set of coordination rules over a period of time.

On the other hand, sessions have been classified by Szyperski[19] in terms of its participant's work (role) as follows:

- **Democratic:** Participants have the same rights. In general, group interaction is determined by the team members.
- **Conference/Panel:** This type of sessions include one or more moderators (e.g. professor) and a group of session attendees. Normally, the moderator specifies the session's group interaction rules.
- **Hierarchical:** In stratified environments, session interaction rules can vary widely, from boss-subordinates relation to many hierarchical structures working in group, collaborating in some related tasks.

The main difference between these session types is the interaction rules that govern how users can interact with each other. That is, we need to define a different C component for each session type.

USER COMPONENT

In this section, we will introduce the notion of user roles drawing an analogy from data types present in programming languages. We conclude this section relating the actual control information needed for any given user with its corresponding role.

Roles

Ellis et.al. [7] define a user role as a set of privileges assigned to a group of users. Similar to data types in programming language theory, a role can be viewed as a user type abstraction that shares the same set of operations or privileges. Thus, a role definition includes role attributes, such as type identifier (e.g. moderator), and role privileges that specify the set of authorized operations.

Role Attributes

The attributes of a data type can be summarized as a type identifier and its internal representation. For example, an integer type can be identified by its name and space that a variable of this type will occupy. Similarly, we need to give each role a unique identifier, and include control attributes such as maximum number of users that can be registered at any one time with this role. In contrast to the fixed space allocation scheme present in many programming languages, the maximum number of users can eventually vary during any given session, since more users than initially expected can join an ongoing session. Note that in special cases, this number can be left undetermined.

Role Privileges

Similar to the operations attached to data types, we will associate a set of functions to each user role. These privileged or authorized operations will characterize the behavior of the group of users, defining the objects that they can access. Clearly, the authorized operations must be drawn from the set of shared functions defined for any public object. Thus, any arbitrary set of artifact operations can be specified for a given role. The set of functions defined for each role constraints the operations that any given role instance (i.e. actual user) can invoke while these restrictions are in effect.

Since role privileges are defined as a set of authorized functions, it seems natural to allow the construction of new roles from previously defined one. By applying basic set operations, such as union and intersection to the set of functions, we can create new roles from existing role definitions.

Furthermore, we define a negation operation (complement) which creates a new role prohibitin the execution of any function included in the original set of functions. Note that user roles, as defined in this section, have a flat structure, as opposed to the hierarchical topologies described in DCPL [5]. In our view, user hierarchies are a special case of coordination dependencies that relate two or more groups of users.

User Representation

The U component in the collaborative state was defined as a set of users, where each element contains user control information. The following list is an outline of the relevant information needed to represent the user state:

- **User Identification**: e.g. user name, social security number, color, or any combination of user identifiers, depending on the context that is being used
- **Location**: For example, the IP, e-mail, or geographic addresses
- **Role**, e.g. moderator, peer, observer

Researchers have argued that session participants can assume several roles simultaneously, however we defined user role as a single-value attribute for any given user in a session. A single-value role can always be constructed, using roles operators and previously defined roles. For instance, in a democratic session every participant (peer role) should be able to contribute to the problem domain. However, if two or more peers get involved in a conflicting situation, an arbitrator (distinguished peer) can help solve their problem. Note that in this scenario, an arbitrator has at least the same privileges as the rest of his/her peers.

Now suppose that the session participants decide to keep a record of their meeting, forcing them to define a new role (i.e. record-keeper). If the arbitrator is selected for this new role, researchers would claim that this participant has been assigned two roles. In our framework, we say that the former arbitrator has been authorized to perform both record-keeping and arbitration functions, forming a new role by applying the union operator to the sets of operations assigned to each of these previously defined roles.

CONTROL COMPONENT

We have subdivided the control component in two main areas, namely artifacts and coordination rules. The former captures the required control information of any shareable object, while the latter allow programmers establish plausible user interaction rules.

Data and Visual Artifacts

We defined an *artifact* as the unit of control information to model physical entities (e.g. pencil) and visual entities (e.g. sprite) characteristics. Clearly, artifacts are related to one or more data (D) and visual (V) objects described in the previous section. It is important to point out that an artifact should not hold actual data drawn from the object problem domain. Instead it includes the information needed to control any object manipulation in a shared environment. For instance, an artifact representing a rectangular figure might contain the maximum number of users that can access this object at any one time and a protocol to handle conflicts if one arises.

We classify artifacts either as basic (e.g. integer, push-button) that cannot be decomposed into simpler objects or as complex (e.g. spreadsheet, line) artifacts that combine two or more (basic/complex) objects. While a complex artifact might allow multiple users to manipulate it, simple artifacts can only allow at most one user to update its content at any one time.

For example, a line can be modeled as a complex artifact, where the underlying basic artifacts are its two endpoints. In this case, the maximum number of users that can grab the line simultaneously is two, each manipulating one of its endpoints.

The information that needs to be kept for an artifact should at least include the following attributes to control the way users can share a given set of objects:

- **Operations:** A set of operations that can be applied to the set of objects being shared.
- **Space Granularity:** Basic or complex artifact.
- **Degree of Ownership (DOO):** Maximum number of concurrent users.

Artifacts could be assembled and decomposed in arbitrary ways, much like the grouping and ungrouping functions available in drawing tools that encapsulate two or more objects in a complex object. Any artifact control system should allow the specification of these coordination attributes at any given time. Recall from section 1, that groupware sessions can vary dynamically, depending on users actions and the current interaction rule that constraints user actions.

1. Operations

We believe that an important goal when developing successful groupware applications is to allow programmers and users have the means to specify the necessary group condition for the execution of a shareable operation (or task). The control information that can be collected for a given operation or task must include *execution state*, *artifacts accessibility*, and *operation type*, among others. This information can be collected either at function- or artifact-level. The latter case can be viewed as a collection of control attributes shared by every element of the set of artifact operations.

Tentatively, we divide these artifact operations under the following classification: real-time, multi-user interface, underlying data, and control/coordination functions. Table 2 shows examples for each of these types of operations. Real-time operations need to meet timing constraints, either because data becomes outdated or the task must meet a strict deadline.

Operation Type	Examples
Real-time	telepointer movement, point-and-drag
Multi-user interface	changing fonts, next-page push-button.
Underlying data	textual insertion
Control and Coordination	set_policy

Table 2. Operation on Shared Data

Multi-user interface operations capture the session state, for example the feedback given to users participating in the same session (i.e. user awareness). Note that user interface operations can change the look-and-feel of a shared object but not necessarily the object itself. On the other hand, data artifact functions can transform or retrieve the state of the object, such as attaching a line annotation to an image or incrementing the image counter whenever a next-image function is executed. Finally, coordination operations can change the way users interact with each other.

2. Granularity

Granularity is intrinsically determined by each shared artifacts and the application functionality. It has been found that adopting only one granularity unit (e.g. characters, paragraph, page) in groupware applications can be too restrictive or inefficient. In general, collaborative applications require flexible granularity specification that can be changed dynamically as shown in the following example.

Suppose two or more users are working together with a shared drawing tool. Lines drawn with this tool can be decomposed in simpler artifacts such as pairs of points in space (e.g. magnitude and slope). If two users manipulate the same line, the application can allow simultaneous updates of the line's magnitude and slope done by two different users. At a later time, another user might need to apply some operation that requires the ownership of both components. In this case, the object should be treated as a basic artifact (i.e. no other user should be able to access it).

Time granularity is also present in groupware environments. For example, update operations applied to shared artifacts can be sent: after some predetermined time interval or at commit point. In a shared editor, text can be propagated either after the user finishes a document unit (e.g. paragraph) or character by character. While the former characterizes a coarse update granularity, the latter takes the finest possible granularity for this artifact.

3. Degree of Ownership

A key aspect for controlling and coordinating objects is its degree of ownership (DOO). We define the term *owner* as any user that can control an object, at any one time, thus *DOO* is the number of concurrent owners for a given object. DOO values can range from zero to the session size. DOO is also constrained by the nature of the artifact and the environment in which it is being used.

A goal for groupware applications is to promote group ownership. After all, groupware should support the interaction of two or more users manipulating shared objects. Users will have a closer perception of actual object sharing if they can manipulate these objects concurrently.

Other research areas have studied concurrent data access, where each artifact is owned by at most one user (DOO = 1) at any one time. This assumption is sound for basic artifacts, such as telepointers that cannot be decomposed into simpler artifacts. However in collaborative environments, concurrent access need to be fully supported for complex artifacts taking into account that data and user interaction constraints are preserved.

4. General remarks

In collaborative environments, DOO and space granularity are closely related. Coarse-grained resources (e.g. complex artifacts) should be able to handle multiple resource owners simultaneously, while fine-grain or basic resources can handle only a limited number of users. If several users want to update a basic artifact (e.g. one bit), it is required that either mutual exclusion or consistency guarantees must be provided [6].

Ellis et.al. [7], Greenberg [12], Dourish [6] and Cortes et.al. [2] have argued that traditional concurrency control mechanisms found in database and operating system areas that do not need to satisfy cooperative requirements.

Coordination

The Webster dictionary defines coordination as *the act of working together in a smooth concerted way*. Control, on the other hand is defined as *the act of checking, testing, regulating or verifying* [19]. From our system perspective, coordination is the set of rules that define the interaction (or working together) between system components and within each component. And control is the action of verifying that these coordination rules are not violated.

In the past, the terms protocol and policy have been used indiscriminately as equivalent to generic coordination tasks. In this section, we define these terms, in order to distinguish among several types of coordination tasks that are commonly found in workgroups.

Early groupware applications had fixed coordination rules, restricting user-user interaction. For instance, the shared editor GROVE [7] was built to support democratic sessioning scenarios, but it does not offer any means to change this setting. Similarly, early groupware programming tools offered very limited, if any, coordination task support [1]. The NYNEX toolkit [1] and Groupkit [15] provide some communication primitives and well defined architectures to build groupware applications. However, programmers using these programming tools have to carry the burden of programming every detail of the coordination tasks for each shared resource.

It was quickly recognized by the CSCW research community [4][8][16] that groupware applications must include support for several protocols and policies in order to cope with different session environments. Current applications support several coordination tasks, but these tasks cannot be redefined dynamically or coexist by attaching them to different shared resources.

Coordination tasks can be designed to access and update component dependencies. It can also make use of control information stored in the model component C to access the corresponding underlying or visual component. For instance, suppose a team has agreed that any participant should wait in line to use a shared pen. An application supporting this rule must verify the artifact's DOO and its granularity attributes to correctly enforce the artifact policy.

Protocols

Protocols can be viewed as a collection of steps that must be followed in order to accomplish a specific goal. These steps can be executed either sequentially or in parallel. We further divide these coordination tasks in terms of its functionality as follows: registration, working, and leaving protocols. Registration protocols are employed by working groups to specify the way new users can join ongoing sessions. Working protocols define plausible conditions to execute artifact operations requested by any member(s) of the team. Finally, leaving protocols specify the way users must exit from the collaborative environment. A detail description and several examples for each type of protocol follows.

1. Registration Protocols

Although registration protocols have been used exclusively for session registration, it is conceivable to attach registration protocols to underlying data and visual artifacts. In this case, participants should follow some registration protocols to access or update object attributes. Under this perspective, access control issues can be viewed as a special case of this protocol type.

The following list of registration protocols is by no means exhaustive, however it presents several examples currently being used in face-face meetings.

- **Invitation**: Users can join the session only after receiving an invitation, i.e. access is restricted to the list of guests.
- **Open house**: No restrictions are imposed on the users to register in advance.
- **Democratic**: Group members can use a voting tool to determine whether a newcomer can join the session. Ongoing users should specify the appropriate parameter to accept a new member.(e.g. 50+%).
- **Appointment**: Managers can appoint employees to participate in a given session in a hierarchical environment.

Notice that registration protocols and session types, described in section 2, are orthogonal concepts. Any session type can have any registration protocol attached to it. For instance, an invitation protocol can be associated with a democratic session, where users can only join after being invited, however once the user has joined he/she can interact freely with other participants.

In summary, a registration protocol specifies how users can gain access to a session or object, while session type states the way users can interact once they have gain this right.

2. Working Protocols

Working protocols specify the order in which artifact functions can be executed when groups of users interact. Although many working protocols constrain task execution to a serial order, these protocols can specify concurrent execution of collaborative tasks. Again, the following list of working protocols is by no means exhaustive, however it illustrates various working protocols:

- **Deadline**: User(s) performing a task must finish before certain time.
- **Consensus**: Users must find a common ground to agree on, such as the task to be performed or the final outcome of a shared object.
- **Strict Order:** User(s) must follow some predefined steps to accomplish a task.

Clearly, these coordinating tasks are not orthogonal. For example. suppose a workgroup agrees to comply with the following rule: a consensus must be reach within some time interval.

3. Leaving Protocols

Similar to registration issues, leaving protocols can be applied globally to a session, or it might be needed for some given resource. For instance, a privileged user can revoke access privilege to other user. The following examples of leaving protocols

- **Any time:** users can leave a session at any time, i.e. no authorization is needed.
- **Authorized:** A user intending to leave must make a formal request, and a decision is taken by one or more ongoing users (e.g. voting or boss), accepting/rejecting his/her request.
- **Ejection**: One or more users can have the authority to eject a participant from an ongoing session.

Several leaving protocols can coexist in the same session. For instance, eject and “Any time” protocols can coexist in a hierarchical session, such as a virtual lecture hall. In this case, a professor will be able to dismiss a student, and students can leave the virtual classroom at any time.

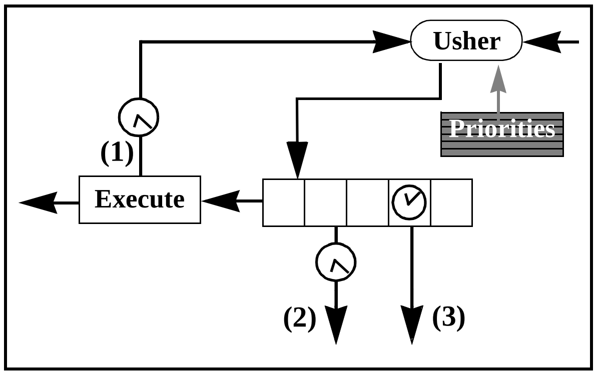

Figure 2. A waiting policy

Policies

The term policy has been defined by the Webster Dictionary as: *a definite course of action selected from among alternatives and in light of given conditions to guide and determine present and future decisions* [19]. Following this definition, we will use the term policy to indicate a decision-making algorithm that establishes some selection criteria in case a conflicting situation arises.

Programmers should be able to define groupware policies that can be associated to multiple sets of shareable objects. This software flexibility allows users to have several objects under the one policy, and several policies for a given object. In the latter case, only one policy should be present at any one time, however depending on the collaborative scenario, users will be able to choose the policy that best fits their needs.

In the scope of this study, we considered the following types of policies:

- **Queueing systems:** user requests are added to a queue only if another user is in control of the artifact. Depending on the artifact, designer, or group of users, many policies can be implemented such as priority queues and multi-level queues. The former is suitable, for example in a hierarchical environment where boss-employee should be taken into account. The latter is suitable for multi-hierarchical environments, for example a very large programming projects involving technical and commercial groups. Figure 2. is a snapshot of the state of a queueing policy that can be associated to some basic shared resource.This policy specifies a bounded priority queue, where a user with the usher role can place new requests into the queue according to a priority table. This protocol also includes time constraints such as[1]: (1) users cannot keep control of the artifact for extended periods of time; (2) users must leave the queue after some time interval even if the request was not executed. And finally, users can drop a request (3) at any time.

- **Master/slave**: a special user polls for or receives requests from other users. For example, a talk show host polls its audience or a lecturer waits questions from the students.
- **Voting tool**: session participants cast votes to take a decision on a given issue (e.g. allow new users join an ongoing session). Several parameters can set the acceptance criteria, such as limited/unlimited voting time, public/private vote, and number of votes needed to approve or reject a request.

The queueing systems gives a broad range of possibilities, such as round-robin, grabbing, priority queues, timed requests, multiqueue systems, etc. For example, a timed-queue policy can be either implemented with round-robin or discard policies. In the former, the user is added again to the queue whenever the timer expires. In the latter, the owner (whose time has elapsed) needs to do an explicit request to be included again in the queue. In either case, a second timer can be used to remove old user requests (e.g. the user needs to leave at certain time).

These basic policies can be combined, forming complex policies, modelling actual policies found in face-face meetings.

1. Numbers in parenthesis refer to arrows in Figure 2.

COORDINATION LANGUAGE

Describing Cooperative Work Programming Language (**DCWPL**) is a textual programming language designed to ease the development of collaborative applications. This programming language is aimed to assist programmers in the construction and maintenance of collaborative applications to support multiple cooperative scenarios given a computational application. Each cooperative scenario can be specified by a DCWPL program, which establishes a set of group interaction rules to be followed by ongoing session participants.

The group interaction rules that can be specified in DCWPL include artifact access controls, waiting policies, session management protocols, and working protocols. In order to establish these rules, programmers must describe from a coordination viewpoint the artifacts that can be shared by session participants. In our framework, an artifact is the coordination specification counterpart of a computational class. An artifact describes the way class instances (i.e. objects) can be shared by a group of participants. Similar to a class definition in an object oriented programming language, an artifact definition comprises attributes and functions. However, artifact attributes represent control information about its computational class counterpart, while artifact functions specify the runtime conditions that must be met to execute a computational function. Furthermore, programmers can include in the definition of an artifact function, the execution of any action, computational or otherwise. Using artifact attributes and functions, programmers can write access control privileges, policies, and protocols.

It must be pointed out that programmers do not need to define an artifact for each class supported in the computational program. That is, no one-one mapping is needed between an artifact and a class definition. Moreover, it is not required that the computational program be developed in an object-oriented programming language or methodology. In our framework, a computational class is an abstraction of a group of entities, identical or otherwise, that need to be controlled under the same control specification. Later in this chapter, we present an example where an artifact is defined as an abstraction of an application which includes many different computational classes.

Our goal was to design and implement a coordination programming language that allow programmers build flexible groupware applications. DCWPL [3] enables programmers to specify how users can share information, as opposed to a computational language that allows programmers define the information that is being shared among a group of users. Moreover, several coordination programs can be developed for a given computational application. For instance, we should be able to develop two or more coordination programs for a shared whiteboard application. Each coordination program represents the rules that govern how the group of users will interact, such as a virtual lecture hall or study room.

CONCLUSIONS

In this paper we have presented a groupware model that decouples the development of coordination mechanisms from traditional computational objects. This programming strategy may lead to the construction of flexible groupware systems. In particular, programmers can develop several group interaction environments for a given set of visual and underlying objects.

In previous programming models, front-end or display servers have been used to display the visual artifacts while client applications have been in charge of maintaining the application's underlying data. As noted by Lauwers [13], this model falls short for the development of groupware systems. In our view, the lack of support for coordination mechanisms forces programmers to include the interaction rules in the artifacts themselves.

Several years ago, researchers argued that the separation of underlying data and the application's visual presentation in two different components was unfeasible. These researchers raised the following issues against the separation: significant increase in user response time and increase in the complexity of developing visual interfaces separated from its underlying data. At the present time, this model is widely accepted by the research community and every major computer environment typically has a visual presentation engine or GUI server.

Similarly, researchers have argued that our implementation model is impractical. At first sight, the new coordination component introduces an additional burden to the programmer who needs to build yet another program, i.e. coordination program. However, the benefits of separating coordination from computational issues outweigh this additional work, because new collaborative scenarios can be supported by developing new coordination programs.

We developed a coordination language and its runtime interpreter to allow programmers specify sets of coordination rules representing some collaborative environment.

In this framework, a coordination program models the group interaction rules found in everyday collaborative scenarios. This allows programmers to create multiple coordination programs for a given computational program. This enables end-users to pick the coordination program that best suits their needs.

REFERENCES

[1] Cortes, M., CSCW Survey: Concepts, Applications, and Programming Tools, Tech. Report 94-006, SUNY at Stony Brook, 1994.

[2] Cortes, M., Mishra, P., Replicated Servers for On-line Groupware, Second International Concurrent Engineering Conference, Washington, Aug.1995.

[3] Cortes, M., DCWPL: A Coordination Language for Developing Collaborative Applications, Ph.D. Thesis, State University of New York at Stony Brook, 1997.

[4] Crowley, T., Forsdick H., MMConf: An infrastructure for building shared applications, Proceedings of CSCW'90, ACM, pp. 329-342.

[5] De Paoli, F., Tisato, F., CSDL: A language for Cooperative System Design, IEEE Transactions on Software Engineering, Vol.20, No.8, pp. 606-616, Aug. 1994.

[6] Dourish, P., The Parting of the Ways: Divergence, Data Management and Collaborative Work, ECSCW ‘95, Stockholm, pp. 215-230, Sept. 95

[7] Ellis, C., and Gibbs S.J. Concurrency Control in Groupware Systems, Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, pp. 399-407, 1989.

[8] Ellis, C., Gibbs, S.J., Rein, G.L., Groupware Some issues and experiences, Comm. of the ACM, Vol.34 N.1, pp.38-58, 1991.

[9] Ellis, C., Wainer J., Goal-based models of collaboration, Collaborative Computing, Vol.1, N. 1, March, 1994.

[10] Ellis, C., Wainer J., A Conceptual Model of Groupware, Proceedings CSCW’94, ACM, pp.79-88, 1994

[11] Greenberg, S., Personalizable groupware: accommodating individual roles and group differences, Proceedings of 2nd ECSCW pp.17-31, 1991.

[12] Greenberg, S., Marwood, D., Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface, Proceedings of the CSCW’94, ACM, pp.207-217, 1994.

[13] Lauwers, J.C., Lantz, K.A., Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems, Proceedings of the ACM SIGCHI Conference in Human Factors in Computing, ACM, 1990

[14] Olson, G., Olson, J,, Defining a metaphor for group work, IEEE Software, pp.93-95, May 1992.

[15] Roseman, M., Greenberg, S., Groupkit: A Groupware Toolkit for Building Real-time Conferencing Applications, Dept. Computer Science-University of Calgary, Proceedings CSCW '92, pp. 43-50, 1992.

[16] Roseman, M. Greenberg, S., Building Flexible Groupware Through Open Protocols, Dept. Computer Science University of Calgary, Tech.report 93-518-20, 1993

[17] Szyperski, C., Ventre G., A Characterization of Multi-Party Interactive Multimedia Applications, Computer Communications, September 1994.

[18] Watson, R., Hua Ho, T., Raman, S., Culture: A fourth dimension of group support systems, CACM, Vol.37 N.10, pp.44-55, 1994.

[19] Webster Dictionary, http://c.gp.cs.cmu.edu:5103/prog/webster.