An Implemention Model for Collaborative Applications

Mauricio Cortés
Prateek Mishra
State University of New York at Stony Brook
Department of Computer Science
Stony Brook, NY 11794-4400
email: {mcortes, mishra}@cs.sunysb.edu
+1-516-632-7678

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 in the development of 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 computational objects and its collaborative environment specification.

INTRODUCTION

A computer program is an abstract model of a given problem. In particular, collaborative programs model the interaction between two or more persons 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 scenarios seem completely unrelated, the interaction between 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 [17] 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 individual/group goals can also affect the team working environment. Furthermore, it has been found that these group settings can vary from meeting to meeting, and even within an ongoing collaborative session [13]. Therefore, customizable applications need to be developed in order to cope with individual [10] 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.

Lets consider the following example to illustrate the need of building flexible and adaptable groupware applications. Simulated screen dumps of this browser are depicted in Figure 1. Participants name with their corresponding color, drawn as background, are shown in the lower right corner of the screen. 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 byusing a shared pencil.

At this point in the session, user A is examining sixth image, as shown in upper left corner of Figure 1.a., while users B and C are looking at the same X-Ray image (see Figure 1.b.). User B owns the telepointer that is being displayed at C's screen, meanwhile user A is waiting for the next image to be displayed. User A drew an annotation on image number 5, while an annotation was added by B on the sixth image. Note that user A has customized her/his screen by changing fonts and numeric

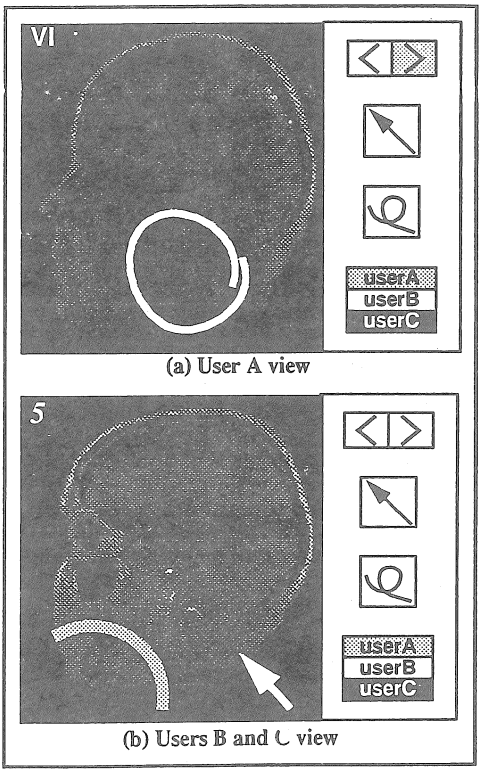

Figure 1. CT/MRI Collaborative browser

data presentation, while users B and C have exactly the same view at this point.

Such collaborative imaging browsing tool 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). 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 setting, the physician might only allow the patient to observe (read-only) his/her own medical images. In contrast, a group consultation scenario might require that physicians have equal rights over every image.Besides these access control issues, each group of users can follow their own interaction rules. Recall that user actions can be applied to shared resources which can affect other users view. Navigating through a set of images and creating image annotations are plausible user actions that can interfere with each other if applied simultaneously in a WYSIWIS (what-you-see-is-what-I-see) environment. In order to coordinate these user actions, some floor control mechanism can be defined, reflecting the way each group of participants wants to run their own meeting.

We believe that it is virtually impossible for programmers to foresee all possible mechanisms that users might need at any given instance. Instead, programmers should give users tools to define their own agenda. This implies that groupware applications should include a customizable coordination component. In the same way single user applications allow users to personalize its own visual presentation (e.g. background colors), groupware systems should allow the specification of human-human interaction rules.

Furthermore, these medical scenarios allow users to collaborate both in synchronous (same time) and asynchronous (different time) mode. For instance, group consultation can take place in real time where physicians interact with each other at the same time or they can interact through email messages at different times. These modes reflect the way participants collaborate with each other which should be captured by the coordination component mentioned earlier. Although many researchers have suggested a clear separation between these two modes, from a programmer's perspective the core functionality of both modes remains the same. 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 if we clearly separate their computational issues (e.g. visual presentation) from group coordination issues.

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++), and that no control or coordination information has been included on those objects. 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. Since we have maintained the clear separation between visual and underlying objects, multiple views can be deployed for a given set of underlying data component. Furthermore, multiple interaction rules can exist for a given pair of data and visual objects. In this way, working groups could use the appropriate interaction rules to meet the functionality required by their collaborative environment.

The rest of this paper is organized as follows. In section 2, we give an overview of our

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 basic 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 than the rest of his/her peers.

Now suppose that the session participants decide to keep a record of this meeting, forcing us to define a new role (i.e. recordkeeper). If the arbitrator is selected for this new role, researchers would argue that this participant has been assigned two roles. Actually, the former arbitrator has been authorized to perform recordkeeping and arbitration functions, forming a new role by taking the union of 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 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 information 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 arise.

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

An artifact has at least the following attributes that is required to control the degree of sharing of a given set of objects:

- **Operations:** set of operations that can be applied to the set of objects that the artifact controls.
- **Space Granularity:** basic or complex artifact.
- **Number of Components (NC):** number of shareable subcomponents
- **Degree of Ownership:** maximum number of concurrent users.

The following constraints should be preserved at all times within an artifact:

1. Granularity = basic $\Rightarrow$ NC = 1
2. DOO $\leq$ NC

As an example, lets examine the representation of relational database implementation under this definition. A relation can be viewed as an instance of an artifact, where the NC attribute will depend on its granularity attribute. If the table granularity is set to basic, the entire table is considered as the only shareable component, implying that only one user can own the relation at any one time. On the other hand, if we define this table artifact as complex, its NC attribute can be set up to the number of tuples in the relation, allowing multiple concurrent accesses to the table. In

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

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 the characterization of an abstraction that shared the same structure and operations or privileges. Thus, a role is formed by a set of role attributes, such as type identifier (e.g. moderator), and role privileges specifying the set of authorized operations.

Role Attributes

The attributes of a data type can be summarized as 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. In contrast to the fixed space allocation scheme present in most 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 for in special cases this number should 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 restricts 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, seems natural to allow the construction of new roles from previously defined one.by applying set operations such as union and intersection. Furthermore, we define a new negation operation which creates a new role that cannot execute any of functions included in the original set of authorized functions. Note that user roles, as defined in this section, have a flat structure opposed to the hierarchical topologies described in DCPL [4]. 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 which is being used
- Location, e.g. IP, e-mail, or geographic addresses
- Role, e.g. moderator, peer, observer

groupware implementation model. Sections 3 and 4 gives a brief description of user and coordination components, respectively. Finally, in section 5 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 system components must specified, 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 data objects, group of active users, and group interaction rules. Visual (e.g. pushbutton) and underlying data objects (e.g. raw medical images) model physical/virtual resources, which can be manipulated by some set of program functions. We must keep user information such as its address (e.g. email address), privileges, role within this session. 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
Data 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

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) has some effect on other users' state. Notice that the widely accepted implementation model to develop single user applications is a special case in our approach, letting U and C be a singleton and an empty set, respectively.

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 into our model. Thus, the notion of session history can be defined as a sequence of S-tuples $[S_{t0}, S_{t1}, \ldots, S_{tn}]$, 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.[6] 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:

this case each tuple is an artifact in itself that also requires a granularity specification. Again, the artifact tuple can be defined as a basic object, in which case we arrive to traditional database implementations, or as complex object, allowing multiple users access the same tuple simultaneously. Finally, the inclusion of relational operator (e.g projection), in the set of shareable operations completes the definition of the database artifact.

Artifacts could be assembled and decomposed in arbitrary ways, as the grouping and ungrouping functions present in drawing tools that encapsulate two or more objects in a complex object. Any artifact control system should let programmers specify these resource 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.

Lets examine each of these artifact attributes in more detail.

1. Operations

We believe that a main goal to successfully develop groupware applications is to provide programmers and/or users some means to specify the necessary group condition for a shareable operation (or task) to be executed. The control information that can be collected for a given operation or task must include its *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 operations.

Tentatively, we divide artifact operations under the following type classification: real-time, multi-user interface, underlying data, and control/coordination functions. Table 2. below shows examples for each type of operation. Real-time operations needs to meet timing constraints, either because data is invalid after some time interval or task must meet some 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. Types of shared tasks

Multi-user interface operations capture the session state, for example the feedback given to users on same session (i.e. user awareness). Note that user interface operations can change the look-and-feel of the 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.

2. 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 maximum between the session size and NC. The nature of the artifact and the environment in which it is used, also constrains this maximum value.

One goal for groupware applications is to promote group ownership. After all, users should be able to manipulated shared objects as much as possible. Users will have a closer perception of actual object sharing, if they can manipulated this 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 telepointer, that cannot be decomposed into simpler artifacts. However in collaborative environments, concurrent access should be fully supported for complex arti-

facts taking into account that data and user interaction constraints are preserved.

3. Granularity

Granularity is intrinsically determined by each shared artifacts and the application functionality. It has been found that adopting one granularity unit (e.g. characters, paragraph, page) in groupware applications can be too restrictive or computational 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 tools. Lines drawn with this tool can be decomposed in simpler artifacts such as pairs of points in space (e.g. magnitude or slope). If two users manipulate the same line, the application can allow one of these users to change the line's magnitude and the second user to update line's slope, simultaneously. However, at a later time, some user might need to apply some operations 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 environment. Update operations applied to shared artifacts can be sent: after some predetermined time interval or at commit point. For instance, in a shared editor, text can be propagated either after the user finishes certain 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 these artifact.

4. General remarks

In collaborative environments, DOO and space granularity are closely related. Coarse-grain resources (e.g. complex artifacts) can handle simultaneously multiple resource owners, while fine-grain or basic resources can be handled by 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 [5] must be provided.

Ellis et.al. [6], Greenberg [11], Dourish [5] and Cortes et.al. [2] have argued that traditional concurrency control mechanisms found in database and operating system areas do not satisfy cooperative requirements. This is basically due to static granularity and isolation premises found in database systems, where ACID properties are enforced and serializability is used as proof of correctness.

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* [18]. From our system perspective, coordination is the set of rules that define the interaction (or working together) between system components and within each component. Leaving control as 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, shared editor GROVE [6] 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 [14] provide some communication primitives and well defined architectures to build groupware applications. However, programmer 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 research community [3][7][15] that groupware applica-

tions must include support for several protocols and policies in order to cope with different session environments. Current applications support several coordination tasks, however these tasks cannot be redefined dynamically or coexist by attaching them to different shared resources.

Coordination tasks can be designed to access/update component dependencies, it can also make use of control information stored in any other component. For instance, suppose a team has agree that any participant should wait in line to use a shared pen. An application supporting this rule must retrieve the artifact's DOO and granularity attributes, in order to enforce correctly this group 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 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 data/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. Session access is restricted to the list of guests.
- **Open house**: no restrictions are imposed on the users registering under this protocol.
- **Democratic**: using a voting tool and setting the appropriate parameters of acceptance (e.g. 50+%).
- **Appointment**: in hierarchical settings, managers can appoint employees to participate in a given session.

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 to a democratic session, where users can only join after being invited, but once the users joins 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 general agreement on how the task must be performed.
- **Strict procedure**: user(s) must follow some predefined steps to accomplish a task.

Clearly, these coordinating tasks are not orthogonal. For example. suppose a workgroup agress in 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.
- **Request:** the user intending to leave must make a formal request, and a decision is taken either by ongoing users (e.g. voting) or boss (e.g. hierarchical environment) 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 sessionl. For instance, eject and request protocols can coexist in hierarchical session, such as the following virtual lecture hall. A professor can dismiss a student at any time, but students must request permission to leave the virtual classroom.

Policies

The term policy have been defined as: *a definite course of action selected from among alternatives and in light of given conditions to guide and determine present and future decisions* [18]. 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:

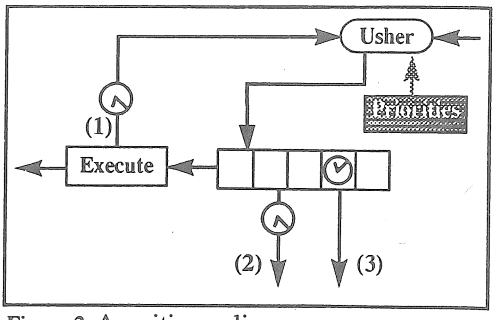

Figure 2. A waiting policy

- **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.

1. Numbers in parenthesis reference arrows in Figure 1.

allow new users join an ongoing session). Several parameters can be 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.

FUTURE WORK AND CONCLUSIONS

In this paper, we have presented a groupware model that decouples the development of coordination issues from traditional computational objects. This programming strategy leads 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 are in charge of the maintenance of the application's underlying data. As noted by Lauwers [12], this model falls short for the development of groupware systems. In our view, the lack of support for coordination issues forces programmers to include the interaction rules in the artifacts themselves.

Currently, we are developing a coordination scripting language and its runtime interpreter to allow programmers specify a set of coordination rules representing some collaborative environment. This scripting language follows very closely the ideas presented here.

In this framework, a coordination program will model group interaction found in some collaborative scenario. Therefore, programmers will be able to create multiple coordination programs for a given computational program.

REFERENCES

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

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

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

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

[5] Dourish, Paul, The Parting of the Ways: Divergence, Data Management and Collaborative Work, ECSCW '95, Stockholm, Sept. 95

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

[7] Ellis, Clarence, Gibbs, Simon J.,Groupware Some issues and experiences, CACM, Vol.34 N.1, pp.38-58, 1991.

[8] Ellis, Clarence, et.al., Goal-based models of collaboration, Collaborative Computing, Vol.1, N. 1, March, 1994.

[9] Ellis, Clarence, et.al., A Conceptual Model of Groupware, Proceedings CSCW'94, ACM, pp.79-88, 1994

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

[11] Greenberg, Saul, Concurrency control in Groupware, Proceedings of the CSCW'94, ACM, pp., 1994.

[12] Lauwers, Lantz, Collaboration awareness in support of collaboration Transparency: Requirements for the next generation of shared window systems, Proceedings of the ACMSIGCHI Conference in Human Factors in Computing, ACM, 1990

[13] Olson, Gary, Olson, Judith, Defining a metaphor for group work, IEEE Software, pp.93-95, May 1992.

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

[15] Roseman, Mark et.al, Building Flexible Groupware Through Open Protocols, Dept. Computer Science University of Calgary, Tech.report 93-518-20, 1993

[16] Szyperski, Clemens and Ventre Giorgio, A Characterization of Multi-Party Interactive Multimedia Applications, Computer Communications, September 1994.

[17] Watson, Richard, et.al., Culture:A fourth dimension of group support systems, CACM, Vol.37 N.10, pp.44-55, 1994.

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