Proceedings of the Second European Conference on Computer-Supported Cooperative Work
Bannon, L., Robinson, M. & Schmidt, K. (Editors)
September 25-27, 1991, Amsterdam, The Netherlands

A Model for Real-Time Co-operation*

Flavio DePaoli
CEFRIEL-Politecnico di Milano, Italy.

Francesco Tisato
Dip. di Scienze dell'Informazione-Università di Milano, Italy

Abstract

This paper introduces a general model for both specifying and designing conferences. A major goal of the model is to be useful at both the specification and the design stage.

The model follows an object-oriented approach. It is based on the different *roles* played by *groups* of conference attendants, and describes conference behaviour in term of *role changes*. Groups are defined at different abstraction levels. Specific activities (multiplexing of data streams, floor-control for a conversation, overall conference management) are driven by *coordinators*. They encapsulate different aspects, such as: device- and media-dependencies, application-dependent behaviours and user oriented strategies. Coordinators can be combined in a hierarchical control structure.

1. Introduction

In a cooperative environment users interact via information sharing. As pointed out in Garcia-Luna-Aveces(1988), there are three basic ways of sharing information. (1) Message sharing: users interact via mailing systems. (2) Data sharing: users interact via shared data bases. (3) Application sharing: users interact with the same application program at the same time.

The two former interaction styles are basically asynchronous. Whenever a user needs to interact with another, it sends a message or updates the database. This has no immediate effect on the workspace of the second user which will later accede asynchronously to the shared information. There are no strong timing requirements;

* This work is partially supported by Olivetti Systems and Networks.

possibly just events' ordering has to be ensured. Put another way, user's workspaces are "individual" and loosely connected.

The latter interaction style is basically synchronous, since a user must perceive the effects of other user's actions in "real time", i.e. in an ordered way and within a negligible delay. This leads to the concept of "shared workspace" (Ishii, 1990).

The shared application gets input streams (e.g., voice, video and data) from some workstations, manipulates them, and delivers them to all the connected workstations. The user does not care whether the application is implemented by shared or replicated processes (see Crowley(1990) for a detailed architectural discussion). In the following we denote by *conversation* the activities performed by users while (logically) sharing an application, and by *conference* a collection of one or more related conversations.

It is worth noting that the application sharing model is general enough to include several kinds of conversations. In particular, video and voice interactions can be modeled as conversations where the shared application just plays the role of a multiplexer, without semantically processing the incoming data streams.

There are several requirements for a conference environment. First, *floor-control* must be ensured by a suitable discipline which may depend both on application's semantics and on user's interaction styles. Second, floor-control strategies must be orthogonal to device- and media- dependent issues. Third, several conversations with (possibly) different floor-control disciplines must be coordinated in a unique conference framework. Fourth, it must be possible to control properly the overall conference set-up and behaviour. Finally, a smooth transition between individual and shared workspaces must be ensured. That is conferees must be able to use the same tools (editors, spreadsheets, and so on) as in a stand-alone situation (Ishii, 1990).

The above requirements suggest that shared applications must be layered to keep the kernel of the application separated both from floor-control issues, and from multiplexing and control of single-medium data streams. Moreover, a hierarchical organization allows the definition of modular-shared applications to be used both as insulated conversations and in a coordinated multi-conversation conference.

Conference aware applications (i.e., applications which explicitly take into account that they are shared) can be designed according to these guidelines. Moreover, a clean separation between applications and floor-control mechanisms allows us to "augment" stand-alone applications with conference mechanisms without modifying the applications themselves.

This paper introduces a general model for both specifying and designing conferences. A major goal of the model is to be both understandable to the end-user and suitable as a basis for an architecture. In other terms, the same paradigm should be used both at the specification stage (where user visibility is the key issue) and at the design stage (where focus is on the architecture).

The proposed model is based on the different *roles* played by *groups* of

conference attendants and describes the conference behaviour in term of *role changes*. The role of an attendant defines the actions he/she can perform (e.g., to speak, listen, or both). Events that are significant in terms of floor-control correspond to role changes. A group collects all the attendants that play the same role. Ultimately, floor-control actions are modeled via movements of attendants between groups.

Specific activities (multiplexing of data streams, floor-control for a conversation, overall conference management) are driven by *coordinators*. Coordinators encapsulate device- and media-dependencies, application dependent behaviours and user-oriented strategies. They can be combined in a hierarchical control structure. An object-based approach is possibly the most suitable for achieving modularity and reusability (Korson, 1990) (Meyer, 1988). A coordinator is defined as an object whose interface provides a set of groups and primitives to modify group memberships.

Section 2 gives an overall description of the model. Section 3 introduces the logical structure of a coordinator and discusses how coordinators can be put together into a hierarchy. Section 4 sketches the current implementation status and future developments.

2. The cooperation model

2.1. The user view

From the user's point of view the basic difference between a stand-alone activity and a conversation is that in the former case there are no distinct roles (for example, the same person is both the application manager and the application user at the same time), while in the latter case each user may play different roles according to floor-control strategies.

Let us consider a single-user word processor: it gets commands (such as *cut* and *paste*) as well as input data (typed characters) from the unique user, and delivers the output data to the same user. If the same word processor is shared, we may want to introduce some floor-control rules. For example, to state that when a user is writing the other users cannot write, but they see the typed text on the screen. Users must be able to issue, in addition to the standard commands of the word processor, extra commands that allow the control of the floor according to specific strategies.

In the general case of a conference commands for the overall conference set-up and control are also needed. It is worth noting that (1) application commands deal with the behaviour of the application as "stand-alone", (2) conversation commands deal with the cooperative strategies related to a single conversation and (3) conference commands deal with overall conference management. This "separation of concerns" seems to be useful both because it corresponds to the user's model of

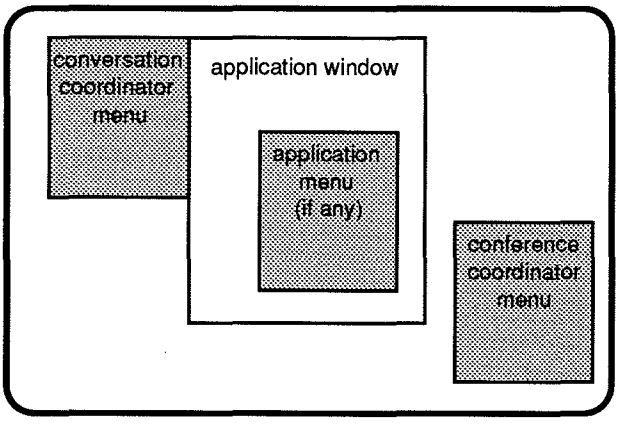

Figure 1. An example of user interface

the conference and because it can be easily mapped into the internal architecture.

Such a scheme is reflected in the menu based interface sketched in Figure 1. It shows a conference that includes a single conversation. Three menus are displayed. First, there is the conference menu. Examples of commands dealing with the overall behaviour of the conference are: join (or leave) the conference, start an application, terminate the conference.

A second menu provides commands related to conversation management. Some of them depend on the floor-control policy for the conversation. Other commands may be more general. Examples of such commands are: I want to write, I have finished writing, suspend the application, iconize the window.

The application menu (if any) belongs to the application and is affected neither by the conference nor by the conversation policy. In the case of a word processor, we may think in terms of the usual commands, such as cut, copy, paste.

Each conference attendant may have different menus enabled at different times according to his/her current role. For example we may state that only the currently writing user can use the application menu, and that not-writing users cannot execute the command *I have finished writing*.

2.2. Roles and Groups

A conference, viewed as a human activity, is characterized by the roles played by the conferees. The abstract concept of role is modeled via groups. The key idea is to introduce a group for every role and to describe the conference evolution by describing how users move from one group to another. This allows us to model the structure and the behaviour of a conference without dealing with the actual number and identity of the conference attendants.

Let us examine a simple example. In a word processor-based conversation there are basically two roles: writers and readers. A writer can both issue data and commands to the shared application, and see screen changes. A reader can just observe the changes on the screen. Let us assume a "one-writer-many-readers"

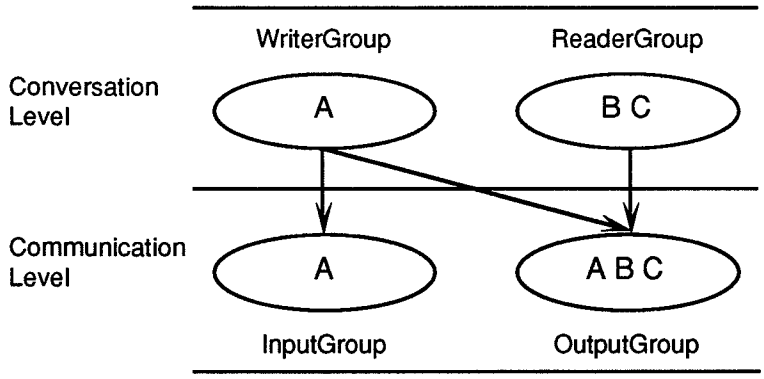

Figure 2. Example of group description

policy: this means that at any given time only one user can play the role of writer. On the other hand, all users except the writer are readers.

This situation is modeled by defining a WriterGroup and a ReaderGroup. The writer belongs to the WriterGroup, and all the readers belong to the ReaderGroup. If a reader becomes the writer, the old writer leaves the writer's group and becomes a reader.

The above is an abstraction representing the logical behaviour of the users. In general, a cooperative activity can be modeled at different levels of abstraction. In particular, if we look at the physical interactions, the users may play two more concrete roles: provide the inputs or get the outputs. Therefore we introduce a new level (we call it "communication level") characterized by two groups: the InputGroup, i.e., the group of users whose input channels are enabled, and the OutpuGroup, i.e., the group of users whose output channels are enabled (note that groups are, in general, not disjoint).

The mapping between the two levels is ensured by a (partial) mapping from conversation level groups into communication level groups, as shown in Figure 2. The meaning is that whenever a user is member of WriterGroup at the conversation level, he/she must be member of both InputGroup and OutputGroup at the communication level. In the same way, when a user is a member of ReaderGroup at the conversation level, he/she must be member of OutputGroup at the communication level. Thus floor-control actions (i.e., group changes) at the conversation level can be mapped into control actions at the communication level.

Figure 2 illustates the above example. At the conversation level the group membership of users A, B and C is driven by the floor-control strategy. At the communication level the group membership of users A, B and C is defined by the group mapping rules. Thus, A is a member of both InputGroup and OutputGroup while the other users are members only of OutputGroup. This means that the writer can type characters on the keyboard and read what he/she types on the screen, while the readers can only read what the writer is typing.

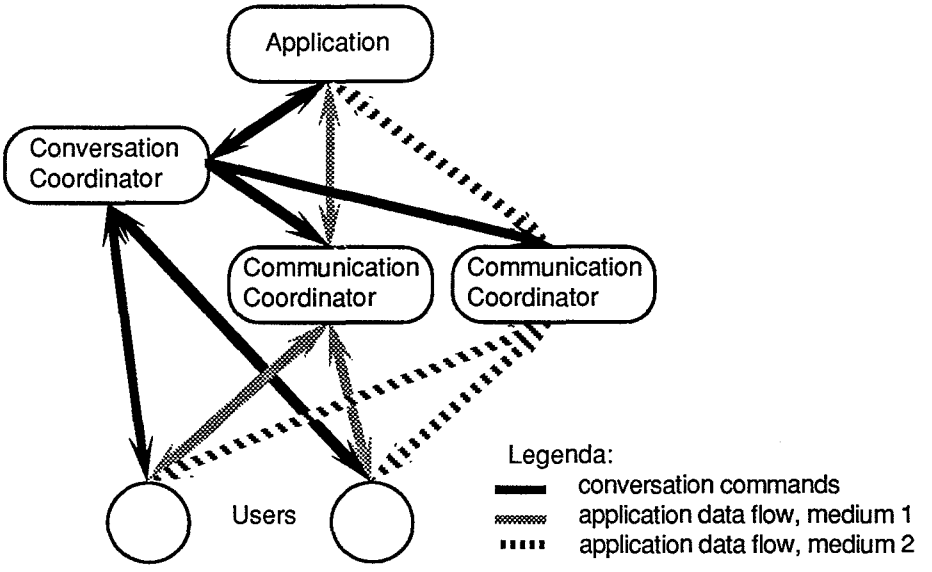

Figure 3. Object-based model of a conversation

2.3. An object-based model

The concepts above describe a cooperative activity from the user's point of view. In the following they are mapped into the architecture of the system supporting the cooperative activity.

The model introduced so far describes a cooperative activity at different levels of abstractions in terms of groups, movements of users between groups, and group mappings between levels. Since concepts and activities are similar across the levels, the system architecture is based on a general class of objects: the *coordinators*. A coordinator is defined as an object whose interface provides a set of groups and primitives for modifying group memberships. Coordinators encapsulate device- and media-dependencies, application-dependent behaviours and user-oriented strategies. They can be combined in a hierarchical control structure.

A conversation is managed by means of three object classes, as shown in Figure 3: the application, a *conversation coordinator* and some *communication coordinators*. The conversation coordinator implements the cooperation rules by interpreting commands generated either by the users or by the application. When the application is not conference aware, there is no flow of information between coordinator and application. The conversation coordinator controls the communication coordinators, which in turn control the actual exchange of information (data and commands) between users and application.

The presence of several communication coordinators reflects the requirement of dealing with different media. A communication coordinator works as a multiplexer that provides a device independent interface for the control of a single-medium data

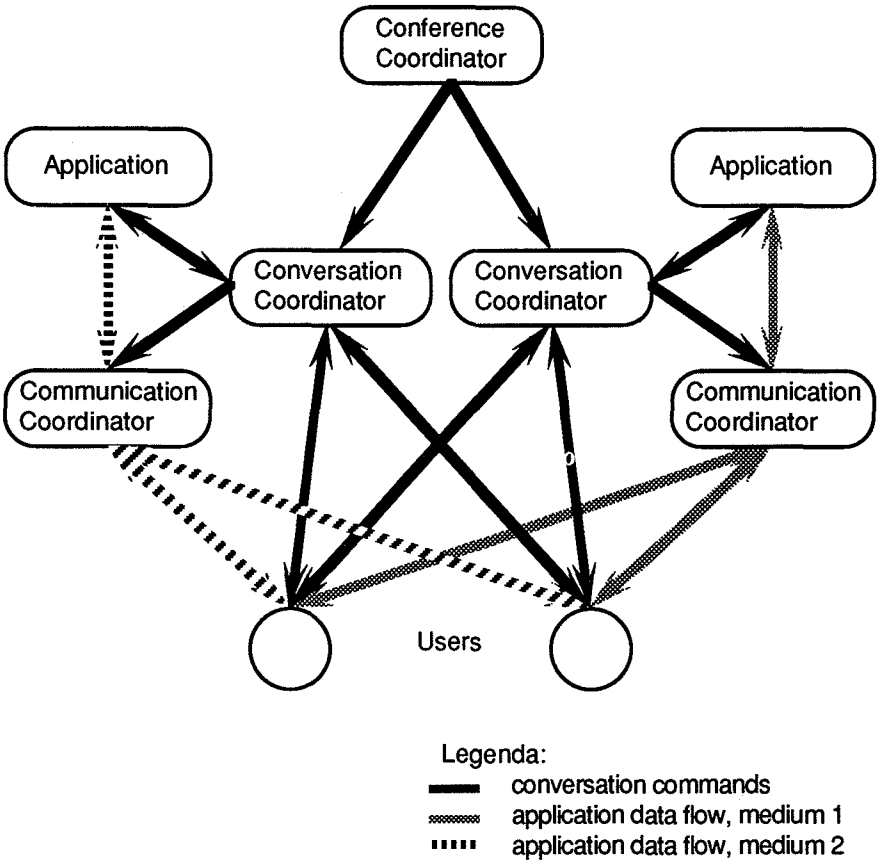

Figure 4. Object-based model of a conference

flow. Communication coordinators are generic with respect to the physical environments: networks with different speed or supporting different media, workstations with different hardware, operating systems, UIMSs or devices.

The model is based on "policy/mechanism separation": in the example above the communication coordinators provide media-dependent mechanisms for multiplexing and demultiplexing data streams, whereas the conversation coordinator provides media-independent floor-control policies.

The above scheme can be extended to overall conference management, possibly with several conversations, by introducing a higher level *conference coordinator* as shown in Figure 4. It is typically in charge of setting up the conference and controlling users when joining and leaving the conference. It controls the conversation coordinators by issuing proper commands. As we shall discuss later, this implies that some conversation commands are no more available to the users.

2.4. Media dependency: communication coordinators

Media dependency is encapsulated by communication coordinators, which behave as multiplexers with the task of providing the physical connections between users

and applications. A communication coordinator is a software module based on specific hardware and software platforms. For example, if both graphics and voice are managed, there could be a graphics coordinator using sockets on top of Ethernet and a voice coordinator exploiting a telephone switching system.

Note that different coordinators provide the same interface even if they are based on different platforms. In fact, all multiplexers have the same description in term of roles and groups. Only three groups are needed to describe a generic communication coordinator: input group, output group and connected group. A user belonging to the input group is able to issue data to the shared application, while a user belonging to the output group is able to receive data from the shared application. The connected group collects all the users that are connected to the communication channel even if they are temporarily not active. This is what happens when a telephone user diverts a phone call: for a while he/she is temporarily disconnected from the audio channel, but he/she is still connected to the switching system.

3. Specification of coordinator behaviour

3.1. Groups and commands

The external interface of a coordinator is specified by means of two basic elements: (1) the *groups* it manages, and (2) the *commands* it is able to execute. As will become clear later, it would be more correct to talk of *public commands*. In the following, whenever no ambiguities arise, we shall use "commands" as a shortcoming for "public commands".

The actions a coordinator performs are ultimately movements of users among groups. It seems reasonable to introduce three basic commands which allow us to control the dynamic evolution of a generic coordinator, i.e., all the possible role changes of a user:

join (U,G);
leave (U,G);
select (G);

where U is a generic user and G is a generic group. Join(U,G) makes the user U member of group G; leave(U,G) removes U from the group G.

Select(G) returns a member of the specified group G. It is a deferred function in on object-oriented terminology, as it is not defined once and for all. In fact, it may have different definitions in order to support different strategies for user selection.

3.2. Invariants

Though the above three general commands are supported by any coordinator, their execution is bound to fulfil some *invariants*, i.e., consistency rules that must always be satisfied to ensure that the cooperative activity managed by the coordinator does not produce incorrect situations.

We introduce a semi-formal notation to specify the invariants.

U **is_in** G;
U **is_not_in** G;

denote that U is or is not a member of group G respectively. The operator *implies* (=>) can be used to impose constraints on user's memberships. The expression:

U **is_in** G1 => U **is_in** G2;

means that if U is a member of group G1, then U *must be* also a member of G2. More information about membership rules can be done by defining the cardinality of a group G with an expression of the form:

G *operator expression*;

where *operator* is a logical operator (=, ≠, >, ≥, <, ≤) and *expression* is an integer expression. Of course, for every group G holds the rule: # G ≥ 0.

For example, let us specify the invariants for a conversation coordinator whose groups are:

In_room: the group of users in the (virtual) conference room;
Speakers: the group of users who are enabled to speak;
Listeners: the group of users who can listen.

The coordinator invariants are

U **is_in** Listeners => U **is_in** In_room;
U **is_in** Speakers => U **is_in** In_room;
U **is_in** Listeners => U **is_not_in** Speakers;
Speakers < 2;

meaning that (1) a user must be in the room in order both to speak and listen, (2) Speakers and Listeners are group representing disjoint roles and (3) there may be no more than one speaker.

3.3. Commands execution

A command triggers the execution of a sequence of actions that modify the internal status of the coordinator. Actions are expressed in terms of basic *private commands*

which implement elementary operations on groups and users. Since the invariants cannot be violated, the invocation of a (public) command may lead to different situations: (1) the command can be executed in a straightforward way by a private command, (2) the command execution implies some extra actions (i.e., sequences of private commands) in order to fulfil the invariants, (3) the command cannot be executed at all. Note that the execution of a public command must be atomic to avoid time-dependent errors.

Three possible public commands for our example coordinator are:

join (U, In_room)
join (U, Speakers)
leave (U, In_room)

Let *Insert* (U,G)/*Extract* (U,G) be private commands that modify the coordinator status by inserting/extracting user U into/from group G, and *Select* (G) be a private command that returns the identifier of a user belonging to group G (the selection policy depends on application issues). The execution of the above commands can be specified as follows:

Command	Execution

join (U, In_room)	**begin**
	  Insert (U, In_room);
	  return true
	end;

join (U, Speakers)	**begin**
	  if U **is_in** In_room **then**
	  begin
	  if #Speakers > 0 **then**
	  begin
	  X = *Select* (Speakers);
	  Extract (X, Speakers)
	  end;
	  Insert (U, Speakers);
	  return true
	  end
	  else return false
	end;

leave (U, In_room)	**begin**
	  if U **is_in** Speakers
	  then *Extract* (U, Speakers);
	  if U **is_in** Listeners
	  then *Extract* (U, Listeners);
	  if U **is_in** In_room
	  then *Extract* (U, In_room);
	  return true
	end;

The public command **join** (U, In_room) can be executed anyway.

The public command **join**(U, Speakers) can be executed only when user U is

already "In_room". Moreover, if there is already one speaker, suitable extra actions must be performed in order to fulfil the invariant #Speakers<2. A user X must be *select*ed and *extract*ed from the Speakers group (the selection is trivial in this particular case). The extraction of X from the "Speakers" group leaves him/her in the "In_room" group. After these actions U can be *insert*ed into the "Speakers" group.

The execution of the public command **leave**(U, In_room) must ensure that user U is *extract*ed from all the groups he/she belongs to.

Commands should be presented to the user via a suitable interface (see Section 2.1) which may also provide atomic macro-commands. A macro-command could be

switch (U,V)

whose meaning is that users U and V must exchange their roles (U is speaking, gives the floor to V and becomes a listener). A possible expansion for this command is

leave (U,Speakers)
leave (V,Listeners)
join (U,Listeners)
join (V, Speakers).

It is up to the reader to look for more clever expansions of such a macro.

3.4. Coordinators control hierarchy

A coordinator is defined by its groups and its public commands. The effect of a public command is to move users between groups by performing suitable sequences of private commands. The previous discussion dealt with a "stand-alone" conversation coordinator receiving commands from the users. As shown in Figure 3, even in this simple situation there are at least two levels of coordinators: the conversation coordinator and the controlled communications coordinators. In general, as shown in Figure 4, a conference consists of a hierarchy of coordinators. A coordinator at a given level is connected via commands to the lower level ones.

Private commands (namely *Insert* and *Extract*) at a given level are implemented by invoking public commands (namely **join** and **leave**) of lower level coordinator(s) according to group mapping rules like those shown in Figure 2. Accordingly, the definition of a coordinator includes the definition of the coordinators it controls and the mapping from the groups it defines into the groups defined by the controlled coordinators.

This mechanism allows us to collect coordinators designed as "stand alone" into a hierarchy, and to map easily abstract roles supported by high level coordinators into more and more "concrete" roles supported by the lower level ones. Note that

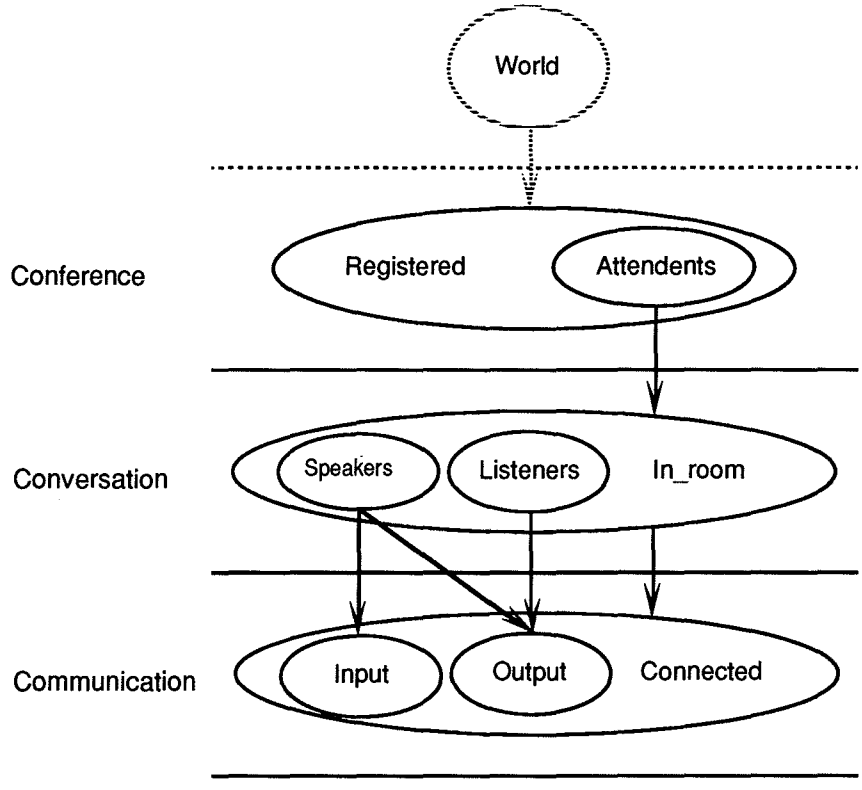

Figure 5. A membership table

groups, invariants, commands and mappings are abstract concepts expressed in a system- and media-independent way. Of course, the iteration stops at the bottom level, that of communication coordinators, whose *Insert* and *Extract* private commands are implemented in terms of system- and media-dependent features. The model is highly modular: a communication coordinator does not need to know what kind of floor-control is supported by the conversation coordinator it is serving, a conversation coordinator does not need to know anything about the conference coordinator and so on.

3.5. Hierarchy and user commands

As soon as coordinators are connected in a hierarchy, a question arises: which commands are visible to the users? It appears that, if a coordinator, say C1, is under control of a higher level one, say C2, then commands of C1 are used by C2 to implement its internal commands (i.e., group changes) according to the group mapping rules. Commands issued by a user directly to C1 cannot violate such rules, otherwise C2 looses the actual status of the system.

A straightforward solution is to impose the rule that a user can issue commands to the topmost coordinator only. A more flexible solution can be devised by looking at the example of Figure 5. The table describes a conference where there are some

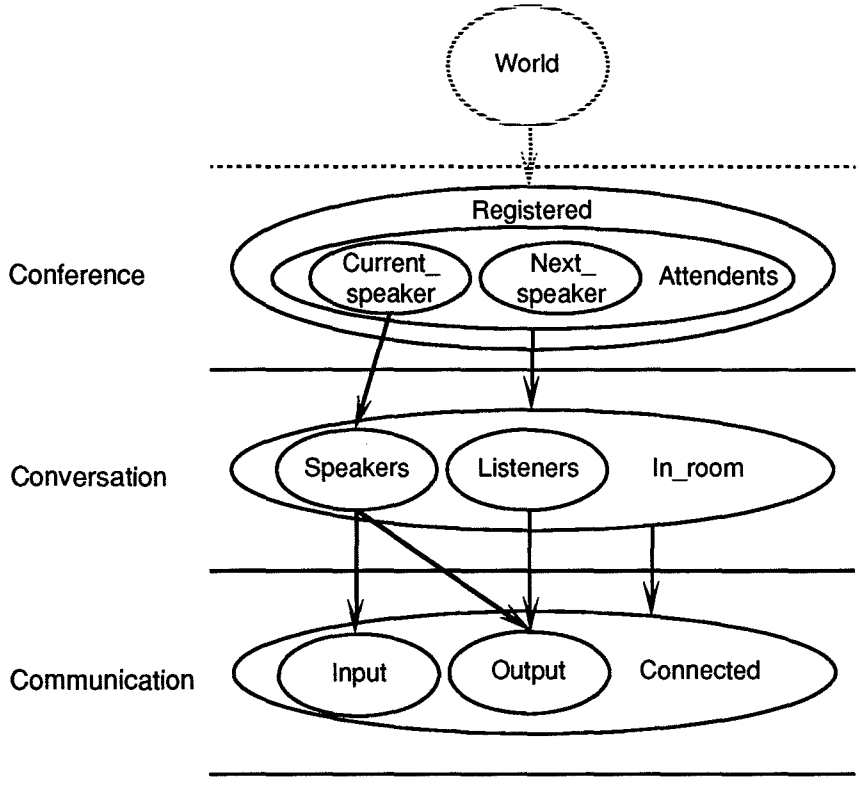

Figure 6. Another membership table

people that have registered (registered group), but not all part of them are actually attending the conference (attendance group). All people that are attending the conference are in the conference room (in_room group). One of them is the speaker, while all the others can be listeners (we consider the possibility that an attendant is not listening, maybe reading the proceedings). The lowest level illustrates the actual communication among attendances: all people are connected to the communication system (connected group), but only the speaker can actually speak (input group) while all the listeners and the speaker can listen. It physically means that the speaker's input channel is open, while the output channel is open for both speaker and listeners.

The arrows illustrate the mapping between groups of two different levels. Let us examine the conversation level (the conversation coordinator has already been discussed in a previous section). The membership of the "In_room" group is controlled by the conference coordinator: every user that belongs to the "Attendants" group is also a member of the "In_room" group. As a consequence, the application commands issued by the users cannot modify the memberships of the "In_room" group. Instead, the "Listeners" and "Speakers" groups are not controlled by the conference coordinator. This means that the users can freely move into and from such groups. In other terms, at the conversation level the only commands available to the users are **join**(U,Speaker), **leave**(U,Speaker),

join(U,Listener), and **leave**(U,Listener), while commands like **join**(U,In_room) and **leave**(U,In_room) can be issued only by the conference coordinator.

Figure 6 illustrates a different situation: the conference coordinator controls the floor taking by imposing that the member of its "current_speaker" group is the member of the "Speakers" group of the conversation coordinator. In this case, the commands **join**(U,Speaker) and **Leave**(u,speakers) are no longer available at the conversation level.

The above examples illustrate that when a coordinator is included into a hierarchy, some of its commands are issued by the higher level coordinator, while the remaining commands are still available to the users. More precisely the only commands available to the user at a given level are those that do not affect the membership to groups controlled by the higher level on the basis of group mapping (i.e., groups with incoming arrows in the figures).

4. Conclusions and future work

The model has been devised as a generalization of Conference Toolkit, an experimental conference system presented in (Bonfiglio, 1988). It provided the basic mechanisms for the implementation of a communication coordinator. In this paper the basic ideas of Conference Toolkit are extended in two ways. First, we define a general model for the description of a conferencing activity by introducing the concepts of role and group. Second, we generalize the concept of coordinator and define how coordinators can be combined into a hierarchical structure in order to design the architecture of a conference control system. The model allows us to define complex cooperative environments by composing basic building blocks and by introducing specific user-oriented policies.

The paper deals basically with specification issues. A set of prototype coordinators has been implemented and has been tested to build several conference control systems implementing different strategies. The prototypes, designed in an object-oriented style, are currently running exploiting X-Windows in a Unix environment.

The experimental activity allows us to conclude that the proposed approach is worthwhile. We have been able to build with a limited effort conferencing systems supporting different cooperation strategies, and to integrate standard X-Windows applications into them.

On-going work is related to the implementation of a set of communication coordinators for non-standard media (voice and video). We are also implementing a set of reusable standard coordinators supporting widely used floor-control strategies and overall conference management strategies. Such coordinators will be collected into a library managed by a semi-automatic tool supporting their composition.

References

Bonfiglio, A., Malatesta, G. and Tisato, F. (1988): "Conference Toolkit: A framework for real-time conferencing", Proc. EC-CSCW 89, September 88, pp 303-316.

Crowley, T., Milazzo, P., Baker, E., Forsdick, H. and Tomlinson, R. (1990): "MMConf: An Infrastructure for Building shared multimedia application, Proc. Conference on Computer-Supported Cooperative Work", ACM SIGCHI & SIGOIS, Los Angeles, October 1990, pp 329-342.

Garcia-Luna-Aveces, J.J., Craighill, E.J. and Lang, R. (1988): "An Open-system Model for Computer-Supported Collaboration", In Proc. 2nd International Conference on Computer Workstations, IEEE, March 1988, pp. 40-51.

Hishii, H. (1990): "TeamWorkStation: Towards a seamless shared workspace", Proc. Conference on Computer-Supported Cooperative Work, ACM SIGCHI & SIGOIS, Los Angeles, October 1990, pp 13-26.

Korson, T. and McGregor, J.D. (1990): "Understanding Object-oriented: a Unifying Paradigm", Communications of the ACM, 33,9, September 1990, pp. 40-60

Meyer, B. (1988): *Object Oriented Software Construction*, Prentice-Hall.

Scheifler, R.W.and Gettys, J. (1986): "The X Window System", ACM Transaction on Graphics, 5, 2, April 1986, pp. 79-109.

Stroustrup, B.(1986): *The C++ Programming Language*, Addison-Wesley.