Development of a Collaborative Application in CSDL

Flavio DePaoli
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci, 32
20133 Milano - Italy
depaoli@ipmel2.elet.polimi.it

Francesco Tisato
Dipartimento di Scienze dell'Informazione
Università di Milano
Via Comelico, 39
20135 Milano - Italy
tisato@hermes.mc.dsi.unimi.it

Abstract

CSDL is a specification and design language for cooperative systems. The language constructs support modular design by separation of concerns. A typical system is composed of a shared workspace, and collaboration control modules That define cooperation rules, control the underlying communication system, and provide multiuser interfaces.

This paper presents the development of a collaborative system based on a stand-alone X11 application. The key-study demonstrates the simplicity and the flexibility of the proposed approach for managing all aspects of a cooperative system design.

1. Introduction

The wide availability of networked workstations and of high-level, high-performance communication services makes computer supported cooperative works a challenging and promising research topic [21]. Cooperation is currently supported by tools that are at different evolution stages. For example, electronic mailing systems are sound and widespread products, while teleconferencing or electronic meeting rooms are still prototype or earlier releases. Most such systems are tightly associated with specific media, technologies and application areas. There is still a lack of general models and of open technological platforms supporting the design of cooperative systems and of "collaboration-aware" applications [15].

An environment supporting the design of cooperative systems should provide the designer with abstractions to model the logical coordination among activities, and to control multiple information flows by hiding system- and media- dependent issues. It must also emphasize modularization to enforce integration and open-endness. Therefore information hiding and separation of concerns [9] are basic issues.

This work has been partially supported by CNR - Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo", and CRAI - Olivetti - Progetto MADE.

Cooperative systems programming deals with four topics: multiuser interfaces, coordination, shared workspace, and networking control. The goal of CSDL (Cooperative Systems Design Language) is to cover all these aspects. The basic idea is that existing software, hardware or design techniques should be integrated to generate new systems. Integration is the crucial issue in the design of real cooperative systems, since they often rely on a network of heterogeneous workstations [7]. Moreover, the capability of integration of existing software permits to use the same tools (editors, spreadsheets, and so on) in stand-alone situations, as well as in cooperation [11].

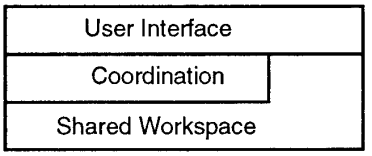

Figure 1. Cooperative systems' architecture.

In a cooperative system can be decomposed in three levels, as sketched in Figure 1. Users interact through a *shared workspace* [12] which can be formed of stand-alone applications or ad hoc applications. The coordination layer is the core of the system, since it supports the definition of cooperation rules and their implementation in terms of access control to the shared workspace. CSDL [6] defines the coordination layer through constructs for cooperation policy definition.

This paper presents the development of a system in CSDL. The system allows a group of physical distributed users to edit a document concurrently. It permits to share the single-user editor xedit by multiplexing the application's outputs to each participant, while inputs come from one user at a time. A simple floor control policy allows participants to designate who has that right. The paper provides a detailed presentation of the coordination layer, and a discussion of system architecture.

Examples of multiuser editors can be found in [13] [8] [16], and [17]. The DistEdit system [14] is related to our work since it provide a toolkit for building and supporting multiple group editors. CSDL is more general since it is not limited to collaborative editors and is open to

0-8186-3770-6/93 $3.00 © 1993 IEEE

multimedia applications. Similar goals are pursued by LIZA [10], MMConf [3], Rendezvous [18], and GroupKit [19].

Section 2 presents the general model for a cooperative system. Section 3 illustrates how CSDL allows a modular design of the coordination layer. Section 4 illustrates the multiuser interface of the system. Section 5 discusses possible implementations exploiting shared processes or replicated processes. Section 6 draws some conclusion, and presents future developments.

2. The coordination model

Cooperating people are characterized by the *role* they play inside a cooperative environment, and cooperation policies define the *role changes* [4]. The partition of users according to their role leads to the definition of *groups of users that play the same role*. This abstraction makes the role definition independent from user identities, and from particular cooperation policies. Moreover, no assumptions are required on shared applications, on communication media, nor on user interfaces.

CSDL's basic unit is the *coordinator* [5]. It is defined as a collection of user groups corresponding to roles. The cooperation control part of a system is composed of one or more coordinators. Coordinators form a hierarchy in which each module uses other modules.

A coordinator is composed of a *specification*, a *body* and a *context*. A specification defines roles and policies in terms of *groups* and *requests* that groups' members can issue; in turn, a request is defined by the *actions* it triggers. A body defines the visibility of the shared workspace according to roles. A context defines which coordinators are connected to a coordinator when it is a component of a complex and modular system.

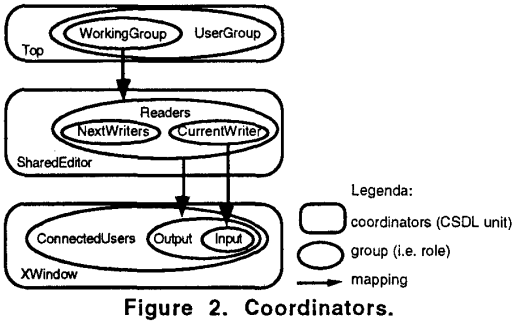

Figure 2. Coordinators.

Figure 2 is a graphical representation of the coordination part of the system we present in this paper. It is composed of three coordinators, each one expressing an abstraction of users' roles. Since the scope of the paper is to present the language features, and the deriving system architecture, we adopted a very simple cooperation policy. We suppose that a group of people may share the same text document, and that one user at a time can write the document. That user is selected through a booking policy.

The Top coordinator partitions the users in two groups: a group formed of potential collaborators, i.e., people that are granted to join a work session, and a group formed of actually working people, i.e., people that are connected to the system and may access the edited document. What these users can do is stated by *mapping* their group to a group of the second abstraction level. It means that the members of WorkingGroup are also members of the Readers group. The Readers group collects people that can read the document. The role of writer is assumed by the member of the CurrentWriter group. This group is *nested* in the Readers group stating that who plays the role of writer, also plays the role of reader. Readers may ask to become the writer by joining the NextWriters group. Note that members of this group are still members of the Readers group.

The last coordinator, XWindow, is the bridge between the coordination layer, and the shared workspace layer. It translates the roles defined in the upper coordinator into access grants to the shared workspace. Thus, members of the Reader groups are members of the Output group, i.e., they can receive data from the shared workspace, and the member of the CurrentWriter group is member of the Input group, i.e., he can send data to the shared workspace.

3. Coordinators

The coordination part of the system formalizes what described in the previous section. For space limits, we present only the specification of coordinator SharedEditor:

```
coordinator SharedEditor {
 invariant #CurrentWriter ≤ 1;
 group Readers {
  type Set;
  requests
   join NextWriters myself {
    actions: insert NextWriters myself; } }
 group CurrentWriter {
  type Set;
  nestedIn Readers;
  requests
   leave CurrentWriter myself {
    actions: extract CurrentWriter myself;}
   join CurrentWriter {
    requires: #NextWriters ≠ 0;
    actions:
     extract CurrentWriter myself;
     insert CurrentWriter remove NextWriters;}
   join CurrentWriter other {
    requires: other in NextWriters;
    actions:
     extract CurrentWriter myself;
     extract NextWriters other;
     insert CurrentWriter other;} }
```

```
group NextWriters {
  type Queue;
  nestedIn Readers;
  requests
   join CurrentWriter myself {
    requires:
     #NextWriters ≠ 0 and
     myself = select NextWriters;
    actions: insert CurrentWriter myself;}
   leave NextWriters myself {
    actions: extract NextWriters myself;} } }
```

A coordinator specification includes an optional **invariant**, and a list of group specifications. The invariant is a logic expression for consistency verification. It is expressed in terms of group cardinality, since members are anonymous. Coordinator SharedEditor has an invariant stating that CurrentWriter has at most one user.

A group specification includes a type (set, queue, stack, ...) that defines in which way members are stored and retrieved (this is needed because users are anonymous), a nesting that defines the dependency from another role, and a set of available *requests*. When two groups are nested (e.g., Readers and CurrentWriter), the set of available requests to a member of the inner group (CurrentWriter) are those associated with both groups. The inner group may hide a request by redefining it.

The cooperation model is based on roles played by users. Thus, requests that users of a system can issue are related to groups' membership. For instance, a member of the group CurrentWriter can issue the following requests

leave CurrentWriter **myself**
join CurrentWriter **other**
join CurrentWriter

plus those available as member of Readers, WorkingGroup, and Input groups. **Myself** means the sender, and **other** means any other but the sender. The last request, that does not mention any user, is an *anonymous request.*

Effects of a request are actions expressed by means of the *internal operators* **insert** and **extract** that change groups' membership. Cooperation policies can be defined through proper combinations of them. For example, the *actions* associated with the anonymous request **join** CurrentWriter make a new writer by removing the sender (i.e., the current writer) from the group, and adding a new writer *removed* from the NextWriters group. Since NextWriters has been defined of type queue, the *first member* is removed. The **select** and **remove** functions return a user selected according to the type of the group. Remove also extracts the user from the group.

A request may include a *pre-condition* that defines when it is acceptable. It is expressed in term of group membership and group cardinality.

3.1. Coordinators' context

Coordinators in Figure 2 are connected by *mapping rules* stating that members of WorkingGroup are also members of Readers group, members of Readers group are member of Output group, and the member of CurrentWriter is member of Input group.

Mapping is expressed by the *context* subunit of a coordinator. A context may include the name of the *controlled* coordinators (one or more) and the group mapping. An example is:

```
coordinator context SharedEditor {
 controls XWindow;
 group Readers {
  controlled;
  mappedTo XWindow.Output;}
 group CurrentWriter {
  not controlled;
  mappedTo XWindow.Input;}
 group NextWriters {
  not controlled;} }
```

When a coordinator is controlled, its set of available requests is reduced according to the mapping. In other terms, requests affecting controlled groups are no longer available to users. This is to assure system consistency: only the controller can send requests that modify controlled groups. Such requests are unrejectable.

3.2. Coordinators' body

So far, we have considered roles (and consequently groups) as abstract description of what a user can or cannot do in terms of requests for coordination control. For instance, we have defined that the current writer controls who will be the next speaker, and when he or she actually becomes the next writer. But roles should also define how users interact with the shared workspace. For instance, the role of writer requires the capability of sending data to the shared editor. The coordinator body defines the visibility of the shared workspace granted to users playing a role.

Communication happens through physical channels, e.g., Unix sockets, ISDN connections, high-speed video channels, etc. Physical channels are modeled in CSDL as *virtual channels*. They model the data exchange between a coordinator and system entities (i.e., users and applications). Virtual channels are controlled by *virtual switchers* that model multiplexing and demultiplexing of data streams in terms of connection matrixes.

A matrix controls a unidirectional flow of data. Hence, bidirectional flow of data are represented by two matrixes even if they are supported by the same physical channel (Figure 3). We made this choice since it models the most general situation we can have. In fact, a virtual bidirectional channel could be implemented by two

physical channels. Moreover, a virtual data flow could be implemented by more than one physical channel (e.g., synchronized audio and video channels). This peculiarity is crucial when dealing with multimedia applications. In our example the two matrixes are just a better graphical representation of physical channels, that are actually bidirectional Unix sockets.

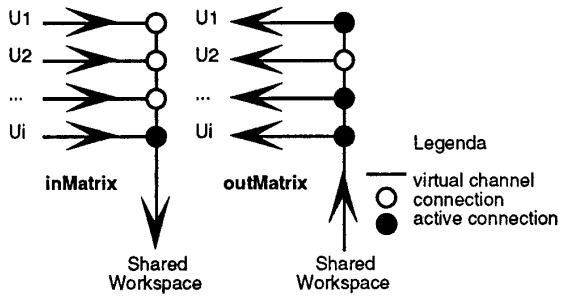

Figure 3. InMatrix and outMatrix.

CSDL programming style suggests to associate a body only to coordinators that are leaves of a tree of coordinators. It means that all application-dependent, and media-dependent features are encapsulated into coordinators at the lowest level. These coordinators are named *communication coordinators.*

In our system only the XWindow coordinator has a body. The communication layer is the XWindow protocol, but it is not relevant at this stage. The body states who has a connection, who is enabled to send data, and who is enabled to receive data. The specification and the body of XWindow is

```
coordinator XWindow {
 invariant #Input ≤ 1;
 group ConnectedUsers {
  type Set;
  requests
   join Output myself {
    actions: insert Output myself; }}
 group Output {
  type Set;
  requests
   leave Output myself {
    actions: extract Output myself;}
   join Input myself {
    actions: insert Input myself; }}
 group Input {
  type Set;
  requests
   leave Input myself {
    actions: extract Input myself; }}}
```

```
coordinator body XWindow {
 switcher inOut XProtocol;
 group ConnectedUsers {
  connected;
  inOff;
  outOff;}
 group Output {
  outOn;}
 group Input {
  inOn;} }
```

A body includes a declaration of a virtual switcher. The declaration defines the *mode* of the channels connected to the switcher: **in** or **out** mode states that the switcher controls a unidirectional data flow, **inOut** mode states that the switcher controls a bidirectional data flow.

Channels are controlled by clauses defining the connections status. Members of ConnectedUsers group are **connected** to the shared workspace through virtual channels, but they are not enabled to send or receive data (**inOff** and **outOff**). When a user joins the Output group, its output channel is enabled (**inOn**), and when he joins the Input group the input channel also becomes enabled (**outOn**). Pictorially, a connection is a white dot in a cross of two virtual channels. When it becomes enabled the dot becomes black.

To complete the definition of a communication coordinator, we must ultimately deal with the mapping between virtual and physical channels. To handle physical channels we need to identify the system's entities, i.e., applications and users. At this stage, anonymous group members are identified as actual entities.

A switcher declaration defines how users and applications can be reached through system or network addresses. They are identified by a Service Access Point (SAP) in the ISO/OSI terminology [20]. Depending on the underlying communication model, SAP may be a process name, a host name, a socket id, a telephone number, etc. If needed, an optional **pragma** specifies system dependent parameters.

A switcher declaration for the system we are presenting could be

```
switcher inOut XProtocol {
 connects U1 through SAP1 pragma { . . .},
          U2 through SAP2 pragma { . . .},
          . . .
          Un through SAPn pragma { . . .},
 to Application through SAPa pragma { . . .}; }
```

It specifies that there is a set of users connected to SAP1, SAP2, ...,SAPn, and an application connected to SAPa. A connection becomes active when a user joins the WorkingGroup group. In fact, he or she becomes member of the Readers group, and then of the ConnectedUsers and Output groups (see Figure 2). Members of the ConnectedUsers group are **connected**, and members of the Output group are enabled to receive data.

Channel definitions above look quite static. They are presented in a declarative form for the sake of readability. An existing prototype version of the architecture underlying CSDL does support dynamic connections. The connection of a new user implies the invocation of a daemon that provides information about the SAP to be used.

4. System architecture

So far we have specified the cooperation part of a cooperative system. In this section we present the influences of specification on system architecture. In particular we present two versions of the system to illustrate how a body can model the actual system configuration.

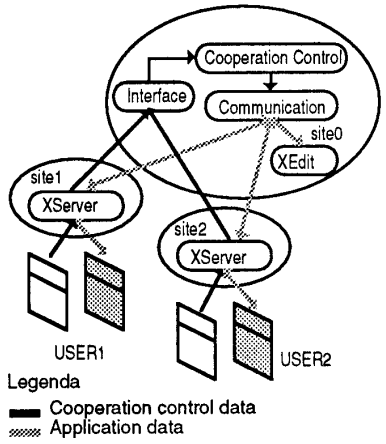

Figure 4. System architecture.

The system is based on a single-user application. The application is shared among several users that may access it according to the cooperation policy defined in the coordination part of the system. This kind of applications can be classified as *cooperation-unaware applications*, since the cooperation control does not depend on their status. In our example we suppose that a user can freely choose his role, the only restrictions being those defined by the policy. The application exchanges data regardless who is the sender or the receiver.

Single-user applications are designed to interact with a user that plays all roles. For instance, the same person can start or quit the application, thus acting as application's manager, and he or she can input data acting as application's user. Most modern single-user software is designed following the client-server schema. In these cases, it is possible to introduce a module that intercepts the input/output streams to simulate a multi-user application. In the X11 context, it means that the module simulates an XServer to the application, while it actually receives inputs and sends outputs to several users [1]. The advantage of this model is the capability of (re)using most of the existing software. The advantages are that users can continue to use those applications they are familiar with, and that no further effort is requested for designing and implementing the applications.

The resulting system has a modular architecture in which modules are independent and reusable in several contexts. In our example, the Cooperation Control module is the implementation of coordinators Top and SharedEditor, the Communication module is the implementation of coordinator XWindow, and the shared application is XEdit (Figure 4). The system architecture is completed by a Interface that allows users to send requests. User interface issue will be discussed in section 4.2.

In CSDL we have modeled the physical channels through inMatrix and outMatrix. Figure 5 is the graphical representation of the communication module. It is split in two parts: The first part is a coordinator with its groups and with users viewed as group members; The second part is a system-dependent part described by virtual switchers. For systems in which the shared workspace is a single application, virtual channels mirror the physical channels. In the Unix environment, modules can be implemented as processes connected through sockets. The module communication is connected to XEdit, and to every user connected to the system. The communication protocol depends on the application (the X11 protocol in this case). CSDL defines a protocol (based on requests) for communication between modules dealing with the cooperation control, namely, between the cooperation control module and the interface, and between the cooperation control module and the communication module.

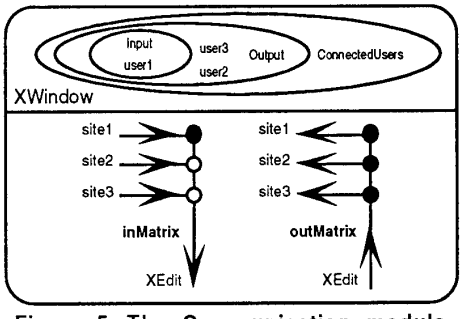

Figure 5 The Communication module.

4.1. Shared vs. Replicated architecture

So far we have considered a cooperating system in which the shared application is implemented by a shared process. Since the environment is a set of workstation connected through a networking system, it may be convenient to spread processes over the sites to exploit the distributed architecture. As expected, the logical description of the system does not change, that is, the

coordinators are not affected by implementation choices. What changes is the definition of virtual switchers that map virtual channels into physical channels.

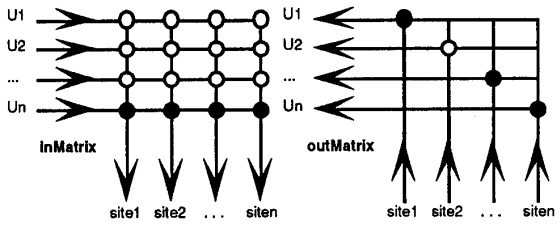

Figure 6. Switchers for replication.

Suppose that on each site there is a copy of XEdit, i.e., the shared workspace is implemented as a set of replicated processes. The virtual switcher in Figure 3 is modified by adding columns (i.e., virtual channels) to obtain square matrixes (Figure 6). Inputs are sent to every copy of the application. Since every copy holds the same status, a user can receive outputs from one copy of the application. This is defined by putting only a dot per row in outMatrix.

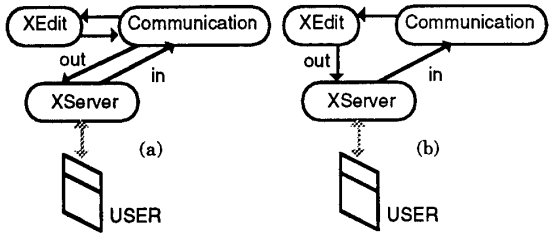

Figure 7. (a) centralized, and (b) partially centralized data-flow control.

The switcher represented by inMatrix and outMatrix could be implemented as a centralized process (Figure 7a). This seems a proper solution for the input stream, since input messages must be sent to every copy of the replicated application, and it guaranties message ordering. Instead, it is more convenient that output messages are sent directly to a user by a local copy of the application (Figure 7b). This improves performance by reduction of message traffic, and introduction of parallelism. This is defined by putting dots on the diagonal of outMatrix.

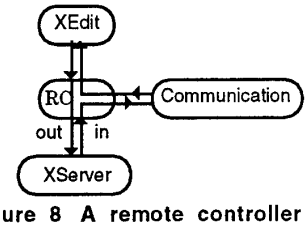

Figure 8 A remote controller RC.

This scheme can correspond to the implementation of applications that have independent input and output channels. Since in our case there is only one bidirectional channel (a socket), distribution can be achieved by the introduction of processes that act as Remote Controllers (RC), as illustrated in Figure 8. RC lets the output data flow through without changes. The input data flow is diverted to the centralized Communication process. Input data for the application are those coming from the Communication process.

Figure 9 illustrates the architecture of the whole system when XEdit is replicated on each site. This time the Communication process is connected to Remote Controllers that act as described above. In general, their behavior is defined by the cooperation policy, thus the virtual switchers associated with remote controllers are driven by the Communication process. In our case they are static since all users receive outputs, and inputs are controlled by the Communication process.

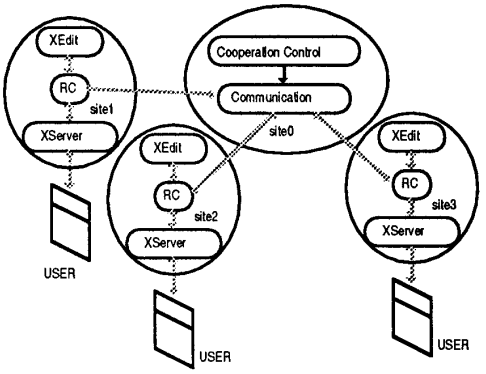

Figure 9 A replicated version of the system.

4.2. Multiuser interface

The multiuser interface of a system should reflect the user interface of system's components. In the system we are studying, there are two types of components: the application implementing the shared workspace, and modules dealing with coordination. On a user's screen there are two related windows, one dealing with cooperation control and another associated with the shared application.

The shared application has its own interface that allows data exchange and application control, as well. A user of the cooperative system will see on his screen a window of its virtual XEdit. Through it the user can read the editing text and modify it when he or she plays the role of writer.

The specification of a coordinator defines which commands can be issued by members of groups, through the definition of available requests. A standard interface may straightforth reflect these commands, as shown in Figure 10. In the example we suppose that user Robert is

the current writer. So, he is enabled to send join commands affecting only the CurrentWriter group and all users, but himself. For instance, the command <join> <CurrentWriter> <Next> sends the anonymous request *join CurrentWriter* to the coordinator SharedEditor. The presented interface is very naif. A designer may develop tailored multiuser interfaces on the top of the CSDL communication protocol. Ad hoc interfaces are useful and desirable, but a discussion about human-computer interaction is out of the scope of the paper.

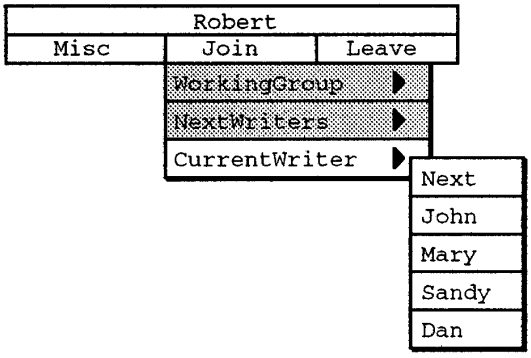

Figure 10. An example of user menus.

5. Conclusions

The goal of the paper was to present CSDL features through the development of a simple cooperative system. CSDL has been designed to support all phases of the system development. It supports system specification by defining coordinators. System architecture can be designed by defining switchers. The paper discusses two possible architecture for systems specified by the same coordinators. The switcher definitions encapsulate all system-dependent parameters that eventually permit the physical connections.

Coordinators are independent from the communication system, and from the user identity. They define rules for collaboration control. Since they are structured in a modular way, a complex cooperation policy can be layered into simpler sub-policies that can be better understood, and developed incrementally.

In the system presented in this paper coordinators are also independent from the shared workspace, i.e., from the applications that are used in collaboration. This is typical when the cooperative system exploits conference unaware application, such as the family of X11 applications. But when the application is *conference aware*, the coordination activity depends partially or totally on it.

CSDL supports the design of these applications. For example, a chess game application can be split into two modules: a module devoted to the game, and another devoted to players control. Figure 11 sketches a possible architecture. This time the control of the cooperation is performed directly by the application. In fact, only the application can define when a player can move.

A prototype of a conference aware application has been developed at CEFRIEL [2]. It allows users to share still images, and to add textual and graphic annotations over the image. The system is implemented by replicated processes, thus allowing tailored presentation on each workstation. The user can choose its own set of pen, its image size, color palette, etc.

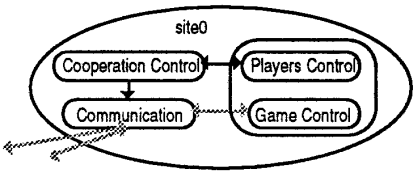

Figure 11. A system architecture for chess games.

6. Future works

CSDL is still in an experimental version. The next step is the complete definition of the language, and the development of the run-time supports for cooperative systems' execution. The ultimate goal is the definition of a complete environment for development and run-time management of computer-based, multimedia cooperative systems. So far we have developed some prototype systems based on the cooperation model presented in section 2. In particular a prototype implements the system described in this paper.

The current run-time supports has been developed in Unix environment. It is formed of a daemon process that allows processes activation and acts as name server for connecting ports, and a generic user interface that allows user to send requests to a system. About communication coordinators, we have already implemented switchers for generic applications based on sockets, and for X11 applications. We aim at completing this set with switchers for multimedia applications based on ISDN. This part of the research will be developed at CEFRIEL, in the context of a project sponsored by Olivetti.

Acknowledgments

The ideas in this paper generalize and extend the results of experimental work carried on at CRAI (Consorzio per la Ricerca e le Applicazioni di Informatica, Rende, Italy) and at CEFRIEL, a consortium among universities and major industries based in Milano. Both activities are supported by Olivetti. In particular, the graphic cooperative annotator IMAGINE has been developed at CEFRIEL under the guide of Silvano Pozzi, to whom the authors are greatly indebted.

References

1. Bonfiglio, A., Malatesta, G., and Tisato, F. Conference Toolkit: A Framework for Real-Time Conferencing. In *Proceedings of the First European Conference on Computer-Supported Cooperative Work,* Gatwick, September 13-15 1989, pp. 303-316.

2. Concolino, P., DiNitto, E., Lisco, M.J., and Molinaro, A. The Image Desk Conference Application Prototype. Tech. Rept. RI 92043CEFRIEL, June, 1992.

3. Crowley, T., Milazzo, P., Baker, E., Forsdick, H., and R, T. MMConf: An Infrastructure for Building Shared Multimedia Application. In *Proceedings of Conference on Computer-Supported Cooperative Work,* ACM SIGCHI &SIGOIS, Los Angeles, Octoberober 1990, pp. 329-342.

4. DePaoli, F. and Tisato, F. A Model for Real-Time Co-operation. In *Proceedings of the Second European Conference on Computer-Supported Cooperative Work,* Amsterdam, September 25-27 1991, pp. 203-217.

5. DePaoli, F. and Tisato, F. Coordinator: a Basic Building Block for Multimedia Conferencing Systems . In *Proceedings of GLOBECOM '91,* IEEE, Chicago, December 2-5 1991.

6. DePaoli, F. and Tisato, F. CSDL: A Language for Cooperative Systems Design. Tech. Rept. 92.60, , Politecnico di Milano - Dipartimento di Elettronica e Informazione, 1992.

7. Ellis, C.A., Gibbs, S.J., and Reln, G.L. GROUPWARE. SomeIssues and Experiences. *Communication of the ACM 34,* 1 (January 1991), 38-58.

8. Fish, R., Kraut, R., Leland, M., and Cohen, M. Quilt: a collaborative tool for cooperating writing. In *Prooceedings of the Conference on Office Information Systems,* ACM, Palo Alto, March 23-25 1988, pp. 30-37.

9. Ghezzi, C., Jazayeri, M., and Mandrioli, D. *Fundamentals of Software Engineering,* Prentice Hall, Englewood Cliffs NJ (1991).

10. Gibbs, S.J. LIZA: An Extensible Groupware Toolkit. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'89),* ACM SIGCHI, Austin, Texas, April 30 - May 4 1989, pp. 29-35.

11. Ishii, H. TeamWorkstation: Towards a Seamless shared Workspace. In *Proceedings Conference on Computer Supported Cooperative Work,* ACM SIGCHI&SIGOIS, Los Angeles, October 1990, pp. 13-26.

12. Ishii, H. and Miyake, N. Toward an Open Shared Workspace: Computer and Video Fusion Approach of TeamWorkStation. *Communication of the ACM 34,* 12 (December 1991), 36-50.

13. Kaiser, G.E., Kaplan, S.M., and Micallef, J. Multiuser, distributed language-based environments. *IEEE Software 4,* 6 (November 1987), 58-67.

14. Knister, M.J. and Prakash, A. DistEdit:A distributed toolkit for supporting multiple group editors. In *Proceedings of the Third International Conference on Computer-Supported Cooperative Work,* Los Angeles, October 8-10 1990.

15. Lauwers, J.C., Joseph, T.A., Lantz, K.A., and Romanov, A.L. Replicated Architecture for Shared Window Systems: A Critique. In *Proceedings of the Conference on Office Informations Systems,* ACM, Cambridge, Massachusetts, April 25-27 1990, pp. 249-260.

16. Lewis, B.T. and Hodges, J.D. Shared Books: Collaborative Publication Management for an Office Information System. In *Proceedings of the Conference on Office Information Systems,* ACM, Palo Alto, March 23-25 1988, pp. 197-204.

17. Newman-Wolfe, R.E. and Pelimuhandiram, H.K. MACE: A Fine Grained Concurrent Editor. In *Proceedings of the Conference on Organizational Computing Systems,* ACM, Atlanta, November 5-8 1991, pp. 240-254.

18. Patterson, J.F., Hill, R.D., Rohall, S.L., and Meeks, W.S. Rendezvous: An Architecture for Synchronous Multi-User Applications. In *Proceedings of Conference on Computer-Supported Cooperative Work,* ACM SIGCHI & SIGOIS, Los Angeles, October 1990, pp. 317-328.

19. Roseman, M. and Greenberg, S. GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications. In *Proceedings of Conference on Computer-Supported Cooperative Work,* ACM SIGCHI & SIGOIS, Toronto, October 31- November 4 1992, pp. 43-50.

20. Tanenbaum, A.S. *Computer Networks - Second Edition,* Prentice Hall, Englewood Cliffs, New Jersey (1988).

21. Special issue on Collaborative Computing. *Communication of the ACM 34,* 12 (December 1991).