Cooperative Systems Configuration in CSDL

Flavio DePaoli, Francesco Tisato

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

Abstract

The aim of a cooperative system is to coordinate and support group activities. CSDL (Cooperative Systems Design Language) is an experimental language designed to support the development of cooperative systems from specification to implementation. In CSDL a system is defined as a collection of reusable entities implementing floor control disciplines and shared workspaces.

A cooperative system may be distributed over a variety of workstations and networks. Therefore, a design language should provide sound supports for system architecture definition and dynamic configuration. This paper surveys the language and discusses constructs for identifying, starting and connecting system entities.

1. Introduction

The demand of systems encouraging and supporting coordination and control of group activities is growing up. In the past few years, *computer supported cooperative work* (CSCW) has been recognized as one of the most promising research discipline. This is because it changes the way users interact with computers and with other users, and because it includes several important and innovative aspects of software systems design and development. A *cooperative system* is fundamentally a distributed system that modifies the traditional user's view of computers as homogeneous and dedicated systems. Consequently, design supports are requested of providing designers with means to control coordination as well as managing heterogeneous and mobile system components.

CSDL (Cooperative Systems Design Language) [5] is a specification and design language that supports the definition of the coordination aspects, and the definition of the logical architecture of a cooperative system. This paper enhances previous definitions of CSDL with constructs related to the design of a more concrete architecture. In particular, it addresses system configuration issues, such as decomposition into modular components, allocation of components in a distributed environment, naming of components and of users, and system start-up.

This work has been partially supported by CRAI-Progetto MADE and MURST 40%.

Among the variety of cooperative systems [7], CSDL deals with computer-based applications in which users interact *synchronously* through a *shared workspace*. The shared workspace can be a traditional single-user application, as well as a *collaboration-aware* application [10, 12, 13].

The hardware supports could be traditional workstations connected by local or wide area networks, as well as multimedia workstations connected by multimedia networks [11, 17]. The goal of CSDL is to provide the designer with a software system architecture and a language for specification and design of cooperative systems.

In recent years many prototypes of a variety of collaborative applications have been developed. Examples are multi-player games, shared agendas, collaborative writing and drawing applications. Most of them are based on tailored architecture, and only a few propose a reference architecture for building distributed, collaborative applications. Examples of architecture proposal include Conference Toolkit [1], MMConf [2], LIZA [9], Rendezvous [14], and GroupKit [18].

The design of a collaborative application embraces a wide range of disciplines, and deals with various design aspects, ranging from user interface to network management. The design of CSDL is based on the *separation of concerns* principle [8]. This allows designers to concentrate on different aspects as independent issues, and provides them with a clean integration framework. We can recognize four basic issues in designing a collaborative system: user interfaces, coordination control, access control, and activities performed cooperatively. Each issue should lead to the definition of an independent module. So that a systems is composed of modules properly connected as in the example in Fig. 1. It illustrates the architecture of a system sharing a standard X11 application [6].

The two central components are the Coordination and Access Control modules. They define the *coordination rights*, and the *access rights* of users, respectively. In CSDL the rights are granted to users on the basis of the role they play inside the system. For instance, in a collaborative drawing system it is possible to identify the two basic roles of drawer and viewer. The drawer can modify the shared canvas, while the viewer can only perceive these modifications.

In term of *access rights*, it means that the drawer can send inputs to the shared drawing tool, on the contrary a viewer cannot send inputs, but she or he can receive outputs. With

1063-6927/94 $03.00 © 1994 IEEE

the term *coordination rights* we intend rights related to the control of coordination policies. For instance we may charge the drawer of starting and quitting the system, or of passing the floor to another user when she or he has finished. Consequently, the user view of a system consists of two interfaces: an interface devoted to coordinate the cooperation, and an interface devoted to interact with the shared application. In the example in Fig. 1 there is a window representing the control menu, and a window reflecting the status of the shared application.

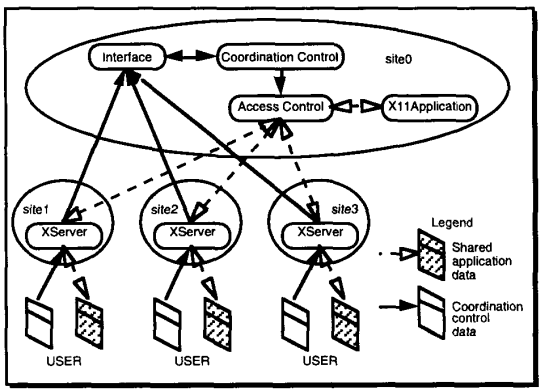

Fig. 1. A cooperative system.

The separation between coordination rights and access rights is crucial since it permits to separate cooperation policies, e.g., floor control passing, and communication management, e.g., connections and message exchanges [16]. Consequently, it is possible to build access control modules as interfaces to the underlying communication system, and make them available to designers that could ignore network and application dependent issues. Moreover, it is possible to customize collaborative systems by designing new coordination control modules, to implement tailored cooperation policies, and connecting them to existing access control modules.

This paper presents CSDL as a specification language that supports the definition of the coordination control and access control modules of a system (section 2). Then it shows how a specification can be interpreted as system design, and how a specification can be exploited to design dynamic and open systems in which the designer can control system configuration (section 3). Conclusions, state of the project and future work will conclude the paper (section 4).

2. System specification in CSDL

CSDL is an object-based language that defines as objects three entities: users, applications, and *coordinators*. Coordinators are the core of the system since they define the cooperation policies and control the data flowing between users and shared applications. Users and application are considered as independent objects identified and integrated into cooperative systems through proper identifiers and descriptors. This section surveys CSDL constructs for specifying coordinators and their usage. The reader may refer to [3, 4, 5] for a more detailed description of basic architecture aspects and language constructs. The next section is devoted to a detailed presentation of system entities and their interconnection to define the final running system.

The Coordination and the Access Control components of Fig. 1 are designed in CSDL as *coordinators*. A coordinator can define both cooperation rights and access rights, but the suggested design style separates them into independent coordinators connected to form a hierarchical control tree. In general, the coordination layer of a system is a tree of coordinators in which each node defines part of the cooperation policy, and where leaves are interfaces to the underlying communication system. Fig. 2 is an example of coordination layer for a collaborative editing system as the one illustrated in Fig. 1.

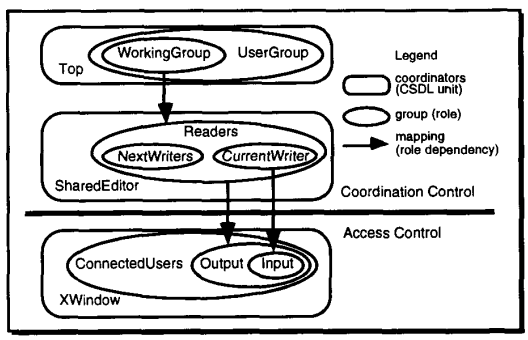

Fig. 2. Coordinators.

Coordinators define cooperation policies through the definition of the roles played by users in term of coordination rights and access rights associated with roles. Each role is represented by a group of users, and the cooperation policy is defined by legal role changes implemented as group membership changes. Fig. 2 is a graphical representation of coordinators showing roles (groups) and their dependencies. There are two kinds of role dependencies. One is an intra coordinator dependency expressed by the concept of subgroup: a group B can be *nested in* a group A to express that B is a *subrole* of A. In other terms, it means that who plays the role B plays the role A, too. The other dependency is an inter coordinator dependency. It is expressed by the concept of mapping: a group A of a coordinator C1 can be *mapped to* a group B of a coordinator C2 to express that C1 *controls* C2 by defining the equivalence of role A and role B. In other terms, it means that groups A and B have the same members that consequently play both the roles.

A coordinator is composed of three parts: a *specification* that defines groups and cooperation policies in term of *requests* exported selectively to members of different groups;

a *body* that defines the access rights associated to groups in terms of communication system control; and a *context* that defines coordinators dependencies in terms of group mapping. Definitions are provided in a declarative way. Fig. 3 illustrates a possible definition of the coordinator SharedEditor of Fig. 2.

The specification starts with the declaration of group identifiers that may include the definition of a *type* and of a *nesting*. The type states a rule for storing and retrieving members. Built-in types are Set, Queue, and Stack. Set is the default type for a group. CSDL supports user-defined types as abstract data types. All types export the operators insert, extract, select and remove. **Insert** and **extract** modify the membership by inserting and extracting a user. **Select** and **remove** operators return a member selected according to the group types. Remove also extracts the user from the group. Select and remove provide support for anonymous manipulation of users. For example, since the group NextWriters is of type queue, then the select operator returns the *first* member of the group.

```
coordinator SharedEditor {
 group Readers;
 group CurrentWriter
  nestedIn Readers;
 group NextWriters
  type Queue;
  nestedIn Readers;
 invariant #CurrentWriter ≤ 1;
 requests {
  exportedTo extern {
   join Readers myself {
    actions: insert Readers myself; }
   join Readers other {
    actions: insert Readers other; }}
  exportedTo Readers {
   join NextWriters myself {
    actions: insert NextWriters myself; }}
  exportedTo CurrentWriter {
   leave CurrentWriter myself {
    actions: extract CurrentWriter myself;}
   join CurrentWriter {
    requires: #NextWriters ≠ 0;
    actions: extract CurrentWriter myself;
             insert CurrentWriter
             remove NextWriters; }}
  exportedTo NextWriters {
   join CurrentWriter myself {
    requires: #CurrentWriter = 0 and
              myself = select NextWriters;
    actions: insert CurrentWriter myself; }
   leave NextWriters myself {
    actions: extract NextWriters myself; }}}
```

Fig. 3. An example of coordinator specification.

The specification of a coordinator is completed by an invariant that states some constraints on group cardinality and memberships through a logical expression, and a set of requests exported selectively to members of groups. The designer can export requests in accordance with the desired policy. The clause **exportedTo extern** means any sender not belonging to any group of the coordinator. Requests exported to external entities support the first inclusion of users (e.g., for system initialization), and allow a coordinator to control another coordinator (e.g., for group mapping). To ensure system consistency when a group is controlled, requests modifying that group can only be sent by the controller coordinator.

```
coordinator context SharedEditor {
 controls X: XWindow;
 group Readers {
  mappedTo Output of X;}
 group CurrentWriter {
  mappedTo Input of X;} }
```

Fig. 4. An example of coordinator context.

A request can be a **join** or **leave** referred to a group and a user. The user is specified by the reserved word **myself**, which means the sender, or by the reserved word **other**, which means any other user but the sender. A request is composed of two optional parts: a pre-condition that defines the request applicability, and a list of actions that implements the request. Actions are group operators as well as system control operators. Control operators will be discussed in the next section since they deal with system design and configuration.

```
coordinator XWindow {
 group ConnectedUsers ;
 group Output
  nestedIn ConnectedUsers;
 group Input
  nestedIn Output;
 invariant #Input ≤ 1;
requests {
 exportedTo extern {
  join Output other {
   actions: insert ConnectedUsers other;
            insert Output other; }
  join Input other {
   requires: other in Output and #Input = 0;
   actions: insert Input other; }
  leave Output other {
   actions: extract Output other;
            extract ConnectedUsers other;}
  leave Input other {
   actions: extract Input other; }}}
```

Fig. 5. A specification for XWindow.

In the example of Fig. 3 the invariant states that there can be at most one writer at a time (#CurrentWriter ≤ 1). The control policy defined by the SharedEditor states that a reader can become the writer in two steps. First she or he should send the request "**join** NextWriters **myself**" to join the group of users requesting to write. Then, she or he can send the request "**join** CurrentWriter **myself**" to become the writer. This request makes the sender current writer

(**insert** CurrentWriter **myself**) if and only if there is not already a writer and the sender is the first member of the group of next writers (#CurrentWriter = 0 **and myself** = **select** NextWriters).

The control mapping is expressed in the context section of a coordinator. It contains the declaration of one or more controlled coordinators, and of mapped groups. The context of SharedEditor is illustrated in Fig. 4. According to what specified in Fig. 2 it controls a coordinator X of type XWindow. In particular, the group Reader controls the group Output of X, and the group CurrentWriter controls the group Input of X. It means that whenever the membership of Readers (CurrentWriter) changes, a request for changing the group Output (Input) is sent to coordinator X. For instance, if a new member joins the CurrentWriter group, then the request "**join** Input *User*" is sent to X. A definition of XWindow is reported in Fig. 5. A controller coordinator can only refer to requests exported to external entities. In the example, it is legal to control the Input and Output groups, but not to control the ConnectedUsers group.

Since the coordinator XWindow does not export any request to members of its groups, it does not grant any coordination rights to its members. It is a passive interface to the underlying communication channels. Members of its groups have the access rights defined in the body section.

The final goal of a cooperative system is to allow users and shared application(s) to exchange information each other in a controlled way. Data are exchanged through communication channels, e.g. Unix sockets, ISDN connections, high-speed video channels, and the like. The logical management of communication channels is achieved in CSDL by *virtual switchers* that model multiplexing and demultiplexing of data streams with connection matrixes, as sketched in Fig. 6.

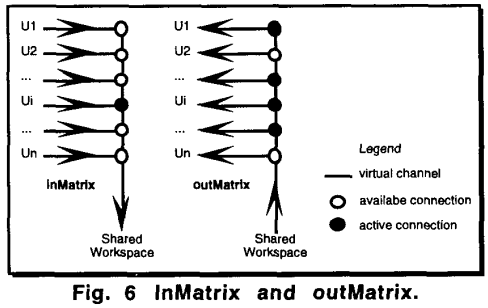

Fig. 6 InMatrix and outMatrix.

A matrix models a unidirectional flow of data between the shared workspace and the users. Bidirectional data-flows can be modeled by pairs of matrixes. This abstraction models any kind of physical channels, if bidirectional channels (e.g., sockets) or unidirectional channels. This facility is crucial when dealing with multimedia applications where logical data flows can be realized by multiple channels of different nature.

Switchers can be declared in coordinator bodies. They are declared of kind **in**, **out**, or **inOut**, and can be controlled by the declarative clauses: **connected**, **disconnected**, **inOff**, **inOn**, **outOff**, and **outOn**. Clauses are associated with groups to reflect the status of a member inside the switcher. For example, the clause connected means that when a new member joins the group she or he is connected to the system through the switcher. When a member leaves the group, she or he is disconnected. A body can declare more switchers, in that case, clauses must be followed by the clause **at** *switcher* to state which switcher it is referred to. An example of coordinator's body is illustrated in Fig. 7. Members of group ConnectedUsers are connected but they can neither send nor receive data since they have both input and output channels disabled. Since group Output is nested in group ConnectedUsers, its members are still connected with the input channel disabled, but they have the output channel enabled. When a user becomes member of Input, also the input channel is enabled. Since coordinator XWindow has been designed to handle X protocol communications, the switcher has been declared as inOut to model TCP/IP bidirectional sockets [19].

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

Fig. 7. A body for coordinator XWindow.

CSDL defines the common interface presented above for a generic switcher, since the actual implementation of switchers is irrelevant in this context. The CSDL interface is in a declarative form since a switcher is defined as an abstract data type exporting a set of operations. Their implementation depends on particular media, protocols, and devices. Next section will discuss switcher definition in details.

3. System design in CSDL

In the previous section we have presented the basic constructs provided by CSDL for specifying coordinators, that is, the coordination part of a system. Coordinators could be implemented as independent processes connected by sockets, or they could be collected in a single process. CSDL supports both implementations, leaving the decision to the designer. The former implementation requires an explicit design, while the latter does not require any further development: the CSDL run-time support generates the process directly from the specifications. In the sections below, first we face the problem of automatic generation of a system from CSDL specifications, then the problem of building dynamic and open systems.

The execution of a specification needs the definition of switchers and user interfaces. A switcher provides supports for connecting users and applications, and controls the data flows. We have already discussed how a switcher can be used within a coordinator. Subsection 3.2 will discuss the definition of switchers as interface to communication systems. The user interface allows users to send requests to the system. The user interface is specified by a coordinator through the definition of exported requests: consequently it provides customized menus reflecting the requests exported by a coordinator, and a set of request to control and to monitor the system. The structure of a user interface will be discussed in subsection 3.3.

CSDL is a language independent from any particular environment, and its abstract run-time model can be implemented on top of a variety of systems. For the sake of readability, in the rest of the paper we present some concepts referred to the current implementation on top of UNIX and X11. In particular, we assume that the control part of the system and its user interface are implemented as UNIX processes connected through sockets, and interacting to users through X servers. About shared applications we do not need to make any assumption: they can be UNIX processes, as well as multimedia applications. They are included into systems through suitable switchers.

3.1. System entities

Before discussing systems design we need to define in which way system entities are created and identified inside the system. A CSDL system defines global name spaces of two kinds. There are names of entity types and names of entity instances.

Coordinator and *switcher* specifications settle names of user-defined types. Built-in types are userName, coordName, and AppName. They are defined as strings of characters. An instance of a built-in type holds a symbolic name that identifies unequivocally a system entity and that can be passed to operators as parameter. The symbolic name must be supplied with the instance declaration.

A coordinator can be created in a static or a dynamic way. A static declaration is composed of a local name and a type name separated by a colon:

instanceName : *typeName*;

The declaration creates an instance of type *typeName* in the same address space in which the declaration is included. It is accessible through *instanceName*, and its visibility is limited to the entity in which it is included. For the sake of readability, the type name can be preceded by the reserved word **coordinator**.

Dynamic creation requires the definition of a symbolic name. Coordinators are instanced through the CSDL operator **create** which takes the entity type and an optional site location. In the current implementation, it creates a new process at the specified site. If no site is specified, the new process runs on the same site of the process executing the create. For instance, an instance of coordinator SharedEditor could be created as follows:

Coord: **coordName** "TopCoord";
create Coord: SharedEditor;

Switchers and names can only be created statically as shown for coordinators. The declaration of a switcher can only be included inside a coordinator body, as discussed in the previous section. The creation of users and application is controlled by switchers, as described below.

3.2. Definition of switchers

Switchers provide supports for entity connections and data-flow control. They are specified as abstract data types with the common interface shown in Fig. 8. A switcher implementation is tailored for a particular media and communication protocol. The run-time system has the task of converting the declarative clauses included in body specifications in operational calls to the switcher. For instance, when a user joins the group Input of coordinator XWindow the run time system calls the operator setIn. When a user leaves that group the operator resetIn is called.

A switcher specification defines a type name and a protocol type. The name is used to reference the switcher (e.g., to declare switchers inside coordinator bodies). The protocol type specifies which kind of media and which kind of access will be used by the switcher. Protocol types are InterfaceProtocol, TCPProtocol, XProtocol, VideoProtocol, AudioProtocol, etc. The abstract data type is defined by a list of operators along with their parameters. Each parameter is a name of a system entity.

The specification is completed by a declaration of the so called Service Access Points (SAPs) that specify in which way system entities can be reached [20]. A SAP definition depends on the underlying communication model. It could be a process id, a host name, a socket id, a telephone number, etc.

```
switcher inOut switcherType: protocolType {
 SAP
  user userName at SAP1 pragma { . . .},
  user userName at SAP2 pragma { . . .},
  user userName at SAP3 pragma { . . .},
  application appName at SAP4 pragma { . . .};
 create EntityName ;
 delete EntityName ;
 connect EntityName ;
 disconnect EntityName ;
 setIn EntityName ;
 resetIn EntityName ;
 setOut EntityName ;
 resetOut EntityName ; }
```

Fig. 8. An example of switcher definition.

SAPs can be declared statically as in Fig. 8. Each SAP declaration may include a **pragma** that specifies system

dependent parameters. SAPs can be also supplied dynamically by declaring a name server instead of a list of SAPs. An example of name server declaration is: **nameServer at** *SAP*. The implementation of the name server, as well as the implementation of switcher operators is out of the scope of CSDL.

The current implementation provides a default switcher, called TCPSwitcher, for TCP/IP connections in which operators are linked to C functions that perform creations and connections of sockets. An example of definition is:

connect isBoundedTo *CFunctionName*;

This approach allows the programmer to define the system-dependent actions in a suitable language or formalism.

3.3. The user interface

A coordinator can define a set of requests available to members of its groups. The user interface should provide menus that reflect these requests, and some other commands to let the users control and monitor the system. User interfaces can be automatically generated from the coordinators' specification, or can be designed to provide customized menus.

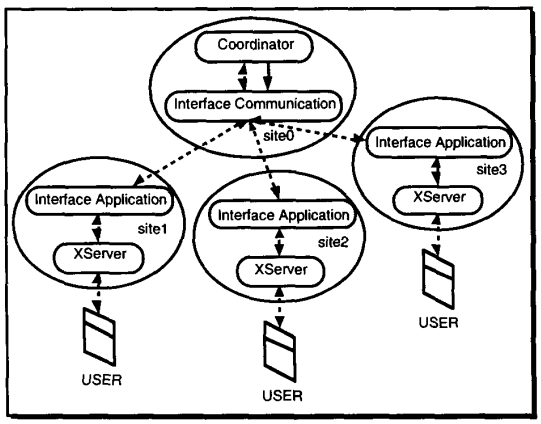

Fig. 9. Replicated interfaces.

A generic interface is made up of two components: an Interface Communication and an Interface Application, as shown in Fig. 9. The Interface Communication includes a switcher devoted to the management of an interface communication protocol designed to support user interfaces. It connects the coordinator to the Interface Applications, in the same way the switchers described in the previous sections connect a shared application to users: in this context the coordinator is the shared application. The data flowing from Interface applications to the coordinator (dotted lines) are interpreted by the coordinator as requests coming from the users. A second connection between the coordinator and the Interface Communication allows the coordinator to pilot the Interface Communication as it controls any other coordinator (black line).

The Interface Applications are designed as independent and replicated components for three main reasons: first, each user plays a different role, and consequently she or he should view a personalized menu; second, it is possible to handle heterogeneous environments; third, the design of a customized interface only requires the design of the remote components exploiting the standard interface communication protocol.

3.4. Building systems from specifications

Once the definition of coordinator's hierarchy and the definition of switcher is completed it is possible to generate a system automatically. This is useful for a variety of reasons. For example, the system can be tested in a host environment, or the designer does not need to optimize the generated system. Specification generates a process including all coordinators connected as specified in the context sections. Switchers are also included in the same process. To execute a system it is necessary to provide the run-time system with the definition of users to be connected and the definition of the root coordinator. This information is provided by a script called startingProgram. An example is illustrated in Fig. 10.

```
startingProgram Minimum {
  C: SharedEditor;
  U1: userName "Robert";
  U2: userName "Kim";
  U3: userName "Mary";
  C.join Readers U1;
  C.join Readers U2;
  C.join Readers U3; }
```

Fig. 10. A starting script.

When the starting script is interpreted, a coordinator of type SharedEditor and references for users Kim, Mary and Robert are created. The coordinator X of type XWindow defined in the context of SharedEditor is also instanciated, and the application mentioned in the SAP table of switcher XSwitcher is started. According to the exported requests, users U1 (Robert), U2 (Kim), and U3 (Mary) are inserted in the Readers group of C by sending requests of kind "**join** Readers **other**". The effect of making users Robert, Kim and Mary members of group Readers is the creation of a control interface for each user, and the creation of an application interface, since they become also members of group Output of coordinator XWindow.

Note that coordinators X and C are private entities that are not visible to other system entities, while symbolic user names can be passed as reference to users. Symbolic names are converted in SAPs to support connections by switchers (see section 3.2).

3.5. Dynamic and open systems design

So far, we have presented CSDL constructs supporting static design of systems. To support dynamic configuration CSDL introduces the set of control operators for creating and connecting system entities presented in Fig. 11. Operators are referred to a switcher definition by the **at** *Switcher* clause that can be omitted when it is not ambiguous. Operators can be included as actions associated with a request, or they can be exported selectively as requests in the same way join and leave requests are.

```
create EntityName at Switcher
delete EntityName at Switcher
connect EntityName at Switcher
disconnect EntityName at Switcher
join GroupName UserName
leave GroupName UserName
setMapping GroupName to GroupName of CoordName;
resetMapping GroupName to GroupName of CoordName;
```

Fig. 11. Control operators.

The create and delete operators generate a new process, and kill an existing process as defined by the referenced switcher. In other terms, when a coordinator executes a create (delete) it executes the create (delete) defined by the switcher. As already discussed the switcher receives the symbolic name of an entity and determines its SAP by a static table or by a name server. The connect and disconnect operators are handled in the same way.

```
coordinator OpenSharedEditor {
 . . .
 requests {
  exportedTo extern {
   connect userName;
   join Readers myself {
    actions: insert Readers myself; }
   join Readers other {
    actions: insert Readers other; }}
   connect coordName;
   disconnect coordName;
   setMapping GroupName
          toGroupName of coordName;
   resetMapping GroupName
          toGroupName of coordName;
. . . }
```

Fig. 12. Public requests to add coordinators.

The connect and disconnect operators referred to coordinators, and the setMapping and resetMapping operators support dynamic system configuration by adding (or removing) branches of the coordinator control tree. The coordinator SharedEditor, renamed OpenSharedEditor could be modified as illustrated in Fig. 12. The exported requests support the dynamic connection of a coordinator and the definition of mappings to groups of controlled coordinators. Parameters are of type coordName to hold the reference of the connecting coordinator, and of type string of character to hold the textual name of groups. OpenSharedEditor has no context section, since its context can be defined dynamically after its creation.

```
startingProgram OpenSystem {
 C: OpenSharedEditor;
 X: coordName "Yellow";
 create X: XWindow;
 C.connect X;
 C.setMapping Readers to Output of X;
 C.setMapping Input to Input of X;
 U1: userName "Robert";
 U2: userName "Kim";
 U3: userName "Mary";
 C.join Readers U1;
 C.join Readers U2;
 C.connect U3; }
```

Fig. 13. Dynamic system configuration.

Exploiting the new interface of OpenSharedEditor the system discussed in the previous section can be started as described in Fig. 13. This time, an instance of coordinator XWindow is created as independent entity, and then put under control of C by explicit requests. The users declared in the starting program and connected to a coordinator (e.g., U3) form the group of *external* entities of the coordinator. They can send requests that the coordinator exports to *extern* through the user interface created by the connect request.

4. Conclusions and future work

CSDL proposes a complete environment to build cooperative applications. The CSDL reference architecture emphasizes separation of concerns by splitting the system into independent components connected through well-known protocols.

CSDL architecture supports either static or dynamic configuration of systems. A static configuration is defined including all information in the coordinator and switcher definitions. In this case to run the system it is sufficient to run the root coordinator and let it know the list of users. As discussed in section 3.4, CSDL generates a single control process including all coordinators and the interface communication.

A dynamic configuration can be achieved by associating each coordinator with an independent process. The process connections can occur any time upon request. Coordinators must define and export requests for new components to be added. This facility supports the design of "open" systems that can be reconfigured dynamically by their status or upon explicit requests.

Moreover, CSDL architecture supports the development of cooperative systems based on standard single-user applications as sketched in Fig. 1. It is accomplished by controlling the data flowing between users and application.

The ability of building collaborative systems on top of

existing application is relevant for two reasons, besides the obvious advantage of saving cost in developing new applications. First, it allows the designers to use tested applications, and second it allows users to use the same applications when they work alone, as well as when they work in collaboration.

CSDL formalizes and generalizes ideas deriving from experimental work. Experiences in CSCW field started with the design and implementation of a platform to share X11 applications [1]. A first definition of an architectural model has been presented in [3, 4], and a prototype environment has been implemented at CEFRIEL laboratories. On top of that environment a complete application has been designed and implemented [15]. Currently, an experimental version of CSDL run-time environment is under advanced development. Future work will deal with the refinement of the formal definition of the language and with the completion of the environment, including tools that for specification, design and implementation of cooperative systems.

Acknowledgments

The authors wish to thank S. Pozzi and E. Di Nitto from CEFRIEL for helpful discussions and comments.

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

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

4. 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.

5. DePaoli, F. and Tisato, F. Language Constructs for Cooperative Systems Design. In *Proceedings of 4th European Software Engineering Conference,* Springer-Verlag, Garmisch-Partenkirchen, September 13-17 1993.

6. DePaoli, F. and Tisato, F. Development of a Collaborative Application in CSDL. In *Proceeding of the International Conference on Distributed Computing Systems,* Pittsburgh, May 25-28 1993.

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

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

9. 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.

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

11. 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.

12. 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.

13. Lauwers, J.C. and Lantz, K.A. Collaboration Awareness in Support Collaboration Trasparency: Requirements for the Next Generation of Shared Window Systems. In *Proceedings of the Conference on Human Factors in Computer Systems (CHI '90),* April 1990, pp. 249-260.

14. 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.

15. Pozzi, S., Peterc, D., Concolino, P., DiNitto, E., and Molinaro, A. ImageAnnotator: An Image-Based Cooperative Application. In *Proceedings of the Conference on Image Communication IMACOM '93,* Bordeaux (France), March 1993.

16. Rodden, T. and Blair, G. CSCW and Distributed Systems: The Problem of Control. In *Proceedings of the Second European Conference on Computer-Supported Cooperative Work,* Amsterdam, September 25-27 1991, pp. 49-64.

17. Rodden, T., Mariani, J.A., and Blair, G. Supporting Cooperative Applications. *Computer-Supported Cooperative Work (CSCW) 1,* 1-2 (1992), 41-67.

18. 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.

19. Stevens, W.R. *UNIX network programming,* Prentice Hall, Englewood Cliffs, NJ 07632 , Software Series(1990).

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