Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

STRONG SHARING AND PROTOTYPING SYNCHRONOUS GROUP APPLICATIONS

I. Tou
S. Berson
G. Estrin
Y. Eterovic
E. Wu

September 1993
CSD-930031

UCLA Technical Report CSD-930031

Strong Sharing and Prototyping Synchronous Group Applications [1]

Ivan Tou, Steven Berson, Gerald Estrin
Computer Science Department
University of California
Los Angeles, CA 90024-1596
office (310)825-2786
FAX (310)825-2273
estrin@cs.ucla.edu

Yadran Eterovic, Catholic University of Chile (Santiago, Chile) (yadran@lascar.puc.cl)
Elsie Wu, NCR Corporation(San Diego) (ewu@cheers.sandiego.ncr.com)

July 15, 1993

[1]The work on the coSARA project was done with partial support provided by AT&T, Hughes, IBM, IDE, Mentor Graphics, NCR, Perceptronics, Sun Microsystems, TRW, UNISYS, and the University of California through its Microelectronics Innovation and Computer Research Opportunities (MICRO) Program.

Abstract

Synchronous group applications allow multiple people to work and interact together. Operations performed by any user are made immediately available to the other users. A sharing model called *strong sharing* is proposed to understand and build such applications. A strong sharing platform, *Object World*, was built that realizes strong sharing by extending Common Lisp objects to support replicated objects and broadcasting methods. The *Object World* platform facilitates migrating existing single-user applications to multi-user applications. It was used to convert a single-user system for modeling and analyzing concurrent systems to a multi-user system, called coSARA.

CoSARA provides a set of design tools that can be used to prototype multi-user applications. The coSARA system allows users to prototype a strongly shared application by graphically specifying the application's data model, structure, and behavior; then by linking various library modules, multiple users can execute and test the application. As an example, this paper shows how the coSARA methodology is used to build a strongly sharable block diagram editor.

Keywords: CSCW, Groupware, Sharable Applications, Graphical Programming

1 Introduction

The nature of work has been changing with more emphasis on team-based activities. Competition has caused major companies to look for ways to improve their product development and decision making processes. Companies have taken new corporate directions in areas such as, concurrent engineering, collaborative design, integrated product development, and total quality control [CART91]. These directions all have a common focus, namely reducing costs and increasing quality by getting people to cooperate and work together more effectively.

Computer-supported cooperative work (CSCW) has been one area of research seeking to address how to use computer-based technologies to support cooperative work [GREI88]. One CSCW approach to improving collaboration is through *synchronous group applications.* These applications support multiple people working together on a common computer application where actions by any user can be seen by the other users. Response times between when an action is performed and when the others can see that action need to be timely enough to support dynamic interaction among the users. An example synchronous group application would be a graphical sketching (whiteboard) tool that runs on multiple sites where any sketch made by one user on one workstation is seen immediately by the others on their respective workstations. Other example synchronous group applications include shared text editors for writing documents together and brainstorming tools for collectively developing new ideas.

Research in synchronous group applications has undergone two generations. In the first generation, researchers tried various point solutions. In the second generation, researchers have been building frameworks to facilitate constructing such applications. An underlying understanding of synchronous group applications is needed to build these frameworks. Current toolkits have provided the base communication and coordination tools necessary for synchronous group applications, but they lack support for understanding the complex interactions that can occur in such applications such as when multiple users can jointly contribute input to a multiple step operation like creating a graphical object. One person may want to provide the name while another provides the location. They also provide limited support, given an implementation of a synchronous group application, for rapidly prototyping alternative implementations.

We propose a *strong sharing* model for understanding and building synchronous group applications. With strong sharing, objects and their behavior are made to satisfy the requirement that actions performed on an object are made immediately visible to others sharing that object. Under this model, synchronous group applications can be viewed as strongly shared applications.

We built a strong sharing platform called *Object World* that facilitates building synchronous group applications. *Object World* insulates application developers from the neces-

sary communication details such as defining what messages should be sent across applications and how to interpret those messages. Application developers are able to use a predominantly single-user application development methodology. Using *Object World*, we succeeded in converting Systems ARchitect Apprentice (SARA) [ESTR86], a system for modeling and analyzing concurrent systems to coSARA [MUJI91], a multi-user system for modeling and analyzing concurrent systems. CoSARA provides a set of tools such as a data definition tool, a control flow tool, and a deadlock detection.

Due to the concurrent nature of synchronous group applications, coSARA has been found useful for prototyping such applications. Using a Petri net like model, coSARA allows application developers to specify graphically the desired interactions that occur in their synchronous group applications. The multiple threads of control that occur in such applications are conveniently captured in those graphical models.

In the next section, we describe previous work in this area. In Section 3, we describe in more detail, our strong sharing model. In Section 4, we describe *Object World* and show how *Object World* greatly facilitates converting existing single-user applications into multi-user applications. Next, we describe coSARA, a system for prototyping synchronous group applications. Finally in Section 7, we describe the coSARA methodology for prototyping synchronous group applications and show how a multi-user block diagram editor was built using the coSARA system.

2 Related Work

Current work on synchronous group applications has focused on providing toolkits for building such applications. Example toolkits include GroupKit [ROSE92], COeX [PATE92], MMConf [CROW90], Rendezvous [PATT90] and Weasel [GRAH92]. The toolkits provide two basic construction components: (1) conference management support and (2) a communication infrastructure. A conference manager is provided for handling the setting up, managing, and tearing down of conferences. A *conference* is defined as the situation where multiple people get together to work through a synchronous group application. A conference manager provides facilities to support people starting, joining, leaving, or ending a conference. Another important role of the conference manager is floor control in order to keep the participants coordinated with each other. Toolkits generally provide a variety of floor control policies from which application developers can choose.

Another key component of synchronous group applications is the communication necessary between users. Actions, performed by one user of a synchronous group application, need to be communicated to all the other users.

Building synchronous group applications with these toolkits is a complex task. Appli-

cation developers need to determine a communication protocol among the applications and at which semantic level the applications should talk to each other. They need to determine which operations are to be done locally and which operations are to be communicated with the others. They need to do other activities such as link input from other applications with the application's normal input mechanisms, set up mechanisms for sharing files and data, and set up means to ensure consistency and updates.

Easier methods are needed for building such applications. We propose a simpler model for building multi-user applications based upon CoLab [STEF87] and notions of sharing objects. CoLab is a computer-augmented meeting room developed at Xerox PARC for small groups. The meeting room has workstations for each participant and a common display screen. The computers are connected by a local-area network to support the sharing of ideas. CoLab uses a replicated architecture where each site keeps a copy of the data and applications. Methods invocations on the objects can be made to broadcast to all sites to keep the objects consistent. Such methods are called *broadcast methods*. Two major drawbacks of CoLab are (1) that it uses full replication where all objects are copied to all sites and (2) that it does not support nested broadcast methods. Full replication is not practical if applications contain large amounts of data such as CAD design tools or if users in a conference only want to share a subset of the running applications. Not supporting nested broadcast methods makes programming difficult since application developers need to make sure a broadcast method does not call, directly or indirectly, another broadcast method.

3 Strong Sharing

In order to enable real time, dynamic group interaction like that found in face-to-face meetings, applications need a style of sharing that goes beyond static database sharing. We call this sharing, *strong sharing*. With strong sharing, multiple users are able to access (share) data and applications in real time. Effects of actions performed by one user are made immediately available to other users.

3.1 Strong Sharing of Data

We define a strong sharing "transaction" as a single simple operation on the data instead of the normal database definition of a sequence of operations. A *strong sharing* transaction is a simple atomic operation; however it does not necessarily move the data from one consistent state to another. An advantage of making transactions be a single simple operation is that the operation and its result can be made immediately visible to other users of the data. The tradeoff is that data consistency is not ensured. Users are allowed to see data in inconsistent states. This is however desirable at times since it can promote greater group interaction by

allowing others to see the evolution of ideas.

A set of operations on data in *strong sharing* is only valid if it corresponds to some serial execution of a set of transactions. Similar to distributed databases, *strong sharing* requires that all users see the same serial execution and that local transactions preserve their local ordering. If 'user a' executes transaction T_1^a followed by transaction T_2^a, then the resulting schedule will always have transaction T_1^a precede transaction T_2^a where any intermediate transactions between the two will belong only to other users. The main difference between strong sharing and database sharing is that *strong sharing* transaction T_1^a could leave the data in some inconsistent state and another user, 'user b' could execute *strong sharing* transaction T_1^b that happens to occur between T_1^a and T_2^a, and therefore uses that inconsistent data.

Since consistency is not guaranteed in each *strong sharing* transaction, we define a higher order transaction which is composed of several atomic transactions. This higher order transaction can then be defined as moving data from one consistent state to another. Associated with the higher order transaction, we define a working phase where several users may make modifications to the data resulting in inconsistencies. Following that working phase, we define a resolution phase where users discuss and make modifications to bring the data back to a consistent state. Each phase consists of a series of atomic transactions. A higher order transaction may consist of several working and resolution phases.

Database sharing has the goal of insulating the users from each other so that users do not need to know about the existence of each other. This is too restrictive for synchronous group applications where users need to know what others are doing. In contrast, strong sharing trades off consistency for concurrency with the goal of supporting multiple users working together.

3.2 Strongly Shared Applications

Under the strong sharing model, synchronous group applications can be viewed as *strongly shared applications*. A strongly shared application has the effect that operations on the application by one user are made visible to the other users sharing the application. All application side effects such as modifications to the user interface and modifications to any application data are made available to all users sharing the application. Given a series of operations by multiple users, the results should be some serial ordering of those operations where all users see the same serial ordering. For example, with a strongly shared graphical editor, when one user creates new objects and modifies existing objects, the effects of those creation and modification operations are made visible to all users sharing that application. This feature however is provided at the cost of allowing users to interfere with each other and make conflicting operations. Extra coordination is required to minimize conflicts.

By having an environment that supports strong sharing of data, one can create syn-

chronous group applications or *strongly shared applications*. An application, in an object oriented framework, consists of a set of objects and a set of methods applied to those objects. By making the objects of an application and their operations strongly shared, the application itself becomes strongly shared. The objects of an application include objects to represent the state of the application, the application's user interface objects, and the application's data. Each of these objects will be accessible by all the users. In particular, strongly sharing the application's user interface objects means that each user will be able to access the user interface. Each user is then able to manipulate the shared application through the user interface. Any action by a user is seen by all the other users.

Given an environment that supports strong sharing, if the users of a strongly shared application are indistinguishable (i.e. the application does not assign unique roles to the users), then a design methodology for single-user applications can be used to build strongly shared applications. The application builder just needs to make sure that the application's objects and methods are strongly shared. All actions by each of the users need to be shared and made visible to everyone else. This is the approach we use in the coSARA system which is explained in section 5.

3.3 Coordination and Conference Management

Strong sharing allows maximum concurrency, but trades off consistency and coherency. Consistency is defined to be the validity of the data with respect to itself and other data. Coherency is defined to be the ability to work without being destructive to others and without others being destructive to one's own work.

Multiple users are able to have continuous access to the data. This allows users to see data in inconsistent states such as in the middle of some set of operations that another user is executing. Strong sharing also allows users to make conflicting actions on data. Two users could simultaneously execute conflicting operations on the same data such as one person deleting some data while another is setting a parameter value. In order to curtail such conflicts, an environment supporting strong sharing needs to provide *coordination support* with safeguards to minimize destructive interactions by multiple users. When problems do occur, support needs to be provided for identifying that a problem has occurred. Also some support is needed for helping users fix any problems that do occur.

Another important issue is managing the participants sharing an application. An environment supporting strong sharing needs to provide *conference management support* with protocols to allow designers to form, join, leave, and close a group activity in an orderly fashion.

Software development techniques and tools need to be developed that help application builders create synchronous group applications. We have found that strong sharing provides a

useful perspective for defining such techniques and tools. In the next section, we describe our approach to the implementation of strong sharing, *Object World*, and show how it supports building synchronous group applications.

4 Object World: An Implementation of Strong Sharing

This section describes *Object World*, our implementation of strong sharing. *Object World* is an object-oriented framework implemented in Common Lisp and CLOS (Common Lisp Object System)[1]. It is a platform for creating, managing, and using strongly shared objects and therefore strongly shared applications. Objects created in *Object World* can be shared across multiple sites. Operations on a shared object can be applied to all copies of that object. With strongly shared objects, application developers can create strongly shared object-oriented applications that support multiple users working together.

Data and applications in *Object World* are CLOS objects. *Object World* uses a replicated architecture and achieves shared data by replicating objects to different sites. A site is able to request an object and have a replica transmitted to that site. *Object World* achieves shared operations by allowing special methods similar to CoLab, called *broadcast methods*, that operate all replicas of an object. A less restrictive version of locking than database locking is provided that allows user cooperation to resolve potential deadlocks. *Object World* also provides a global conference manager and local conference managers for each site in a conference. The global conference manager handles the overall conference management activities such as maintaining the participation list and controlling access to the conference database. The local conference managers handles site specific conference management activities such as sending out requests to join or leave a conference.

4.1 Shared Data and Shared Operations

Object World uses a replication on demand approach to sharing data. After a site creates an object, other sites can request a copy of that object. Upon receiving a request, the site owning a copy of the object encodes that object as a string and then transmits the string to the requesting site. The requesting site then decodes the string, hence creating an exact replica of the object. Objects in CLOS are made sharable by having their class inherit from a special class called *sharable*. The *sharable* class provides the necessary methods for encoding and decoding objects.

[1]CLOS is an extension to Common LISP that provides object-oriented programming.

Object World also supports persistent objects. The same scheme to encode objects is used to save objects. A global conference manager keeps the names of all saved objects in a directory. When users request an object, the global conference manager can look up the object in the directory and retrieve it.

The shared operations are achieved by special methods, called *broadcast methods*, which operate on all shared copies of an object, regardless of which machines store it. In our implementation, existing methods can be upgraded to broadcast methods with very little work.

Broadcast methods are composed of two different methods, one method that is executed on the local site, and a second that is executed on all the sites. The local site method, when executed, broadcasts to all the sites including itself, a request to execute the second method. The second method contains only the original method code, and does not have the side effect of broadcasting. If a remote site were to broadcast back the method that originally caused the broadcast, the system would go into an infinite broadcast loop. The second method also causes any invocations of broadcast methods in its body not to broadcast in order to prevent methods from being unintentionally re-executed.

4.2 Conference Management Support

Object World provides a global conference server and for each site participating in a conference, a local conference manager (see Figure 1). The global conference server and the local conference managers handle the setup, joining, leaving, and shutdown of a conference. A conference is a set of users who are working together through a set of shared applications. To set up a conference, a global conference server needs to be started. Any number of conference participants can enter and leave a conference at any time during the conference.

A conference participant enters a conference by starting up a local conference manager and specifying the desired conference to be joined. To join a particular conference, the local conference manager sends a request message to a specific global conference server. In order to work with other conference members on the same application, the joining member just requests a copy of the application from a participating site.

The global conference server can be started at any site and multiple global conference servers can be started to represent different conferences. The global conference server handles registering and unregistering of participants to a conference. It keeps a list of all participants. The global conference server also acts as a database server providing conference participants with access to the persistent store. Through a global conference server, local sites are able to retrieve objects from the persistent store and save objects to the persistent store. Upon startup, a global conference server sets up an object directory of all those objects available for sharing from the persistent store. The global conference server is also able to store

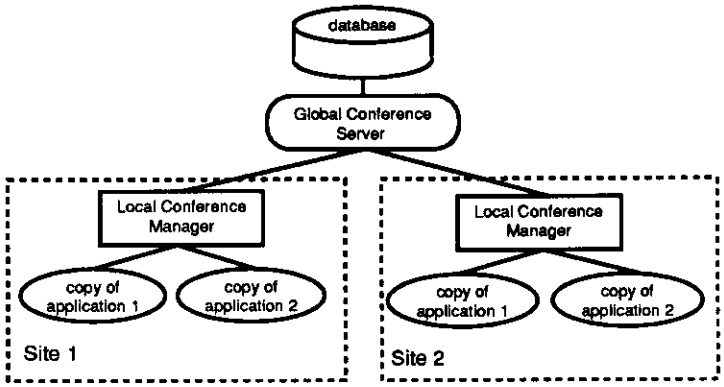

Figure 1: Conference Management Support

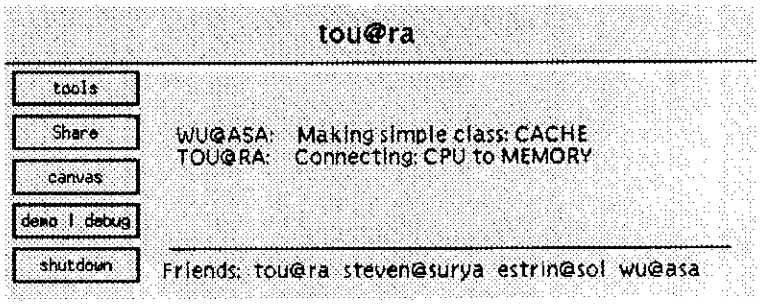

Figure 2: Local Conference Manager

application states and retrieve them at a later time. This feature is useful when participants want to close a conference and later restart where they left off.

Upon starting up a local conference manager, the local conference manager obtains a list of conference participants from the global conference server. The local conference manager then multicasts to all participants that it is joining the conference. All participating sites at that point add the new participant to their participants list. Also at that time, an object directory is copied from any participating site to the local site. This object directory is similar to the global server's object directory. When a site wants a specific object, it uses this directory to determine which site to ask for a copy. If the requested object is not found in the directory, then the object is assumed to be a saved object and the object is requested from the global conference server. The object directory is also used to determine whether a site should handle an incoming broadcast method. If an incoming method applies only to objects not local to the site, then that method is ignored.

The local conference manager (see Figure 2) informs the user which sites are participating in a conference (**Friends:** at bottom of Figure 2) and everyone's current activities (message region at center of Figure 2). The local conference manager also provides a set of conference

commands (button region at left of Figure 2). Applications that can be used in a conference need to be pre-registered with the local conference manager. The local conference manager provides a top level menu of applications that can be executed in the conference. Multiple applications can be executed within a given conference.

Upon leaving a conference, the local conference manager multicasts a message to all the participants' local conference managers and the global conference server, that it is leaving. Upon receiving such a message, each local conference manager removes the participant from its participants list and from its object directory.

4.3 Coordination Support

Strong data sharing is inefficient (and even counter productive) unless data consistency can be preserved. Increased conflict and greater chances of inconsistency are a consequence of greater concurrency and interaction among users from strong sharing. In *Object World*, five measures are taken to ensure data consistency [WU91] [MUJI91].

First, a special broadcast protocol is used to ensure that the same broadcast method operate in all the sites on data in the same state. When a broadcast method is invoked, the sites that will apply the method are those with at least one object from the method's argument list. If a site is lacking some of the objects in the argument list, then at that point, those missing objects are copied to the site. This ensures that the state of the objects where the broadcast method was invoked will be the same as the other sites. The broadcast messages are queued up at each site where the message orderings in the queues are kept the same by the underlying networking system.

Problems however occur with remote sites where broadcast methods need to send messages across different networks to a foreign site. In this case, the ordering of the messages may be different or messages may be lost. In order to detect possible consistency problems, we use a dependency detection model [STEF87]. This is our second measure for handling site consistency problems.

Under the dependency detection model, timestamps are attached to the broadcasting method's arguments. Invocations of a broadcasting method at different sites need to have matching argument timestamps to ensure that the method is applied to the same object states at all sites. This is used to alarm users of possible inconsistency and to facilitate access negotiations. Tools are provided to aid conflict resolution when an inconsistency is detected. A warning dialog box appears on all owners of an object that is suspected to be inconsistent across the sites. A dialog box then displays a list of the differing objects across the sites. At this point, participants can negotiate with each other as to which object is the valid one. Selecting that object in the dialog box causes the selected object to be propagated to the appropriate sites and the offending broadcast method to be discarded.

Third, when data access becomes highly contentious, locking can be used to prevent undesirable access. Locking is implemented by our extension to the dependency detection model called DDL (data dependency checking with locking) [WU91]. Lock requests and releases have timestamps just like broadcast methods. Whenever multiple users try to access an object at approximately the same time, timestamps are used to resolve the conflict and determine which request to accept.

Both read and write locks are available. When locking conflicts are detected, users are supplied with locking information on the object(s), and can resolve them through negotiation. We assume that data accesses are coordinated. When users share data in a face-to-face setting, such coordination can be easily achieved with the richness of face-to-face interaction. When users are geographically dispersed, we assume the aid of multimedia conferencing systems, telephones, e-mail, or FAX.

The fourth measure is just to inform users what other users are doing. By knowing what other people are doing, users can tailor their activities to reduce conflict. So if one person is currently making modification to some module and I am informed about that, then I will know that the module may be in an inconsistent state. If that module needs to be used, it should be used with caution. Associated with each conference site is a local conference manager window (see Figure 2). This window displays various status information. Written descriptions of each user's activities are displayed in everyone's local conference manager window. This is done automatically by having application methods not only do the desired function, but also invoke special library functions that display messages in the local conference manager window.

The fifth measure is to control the granularity of sharing. This is done at the application development level by determining which methods broadcast and which do not. Broadcasting only high level methods means that object changes will occur at a high level across the different sites. Broadcasting only primitive low level methods implies that fine changes will be made visible at all the sites. The application developer has a spectrum of different levels from which they can select to broadcast corresponding to an application's functional abstraction layers. The advantage of broadcasting high level methods is that objects can be made to move from one consistent state to another. The disadvantage is that interesting intermediate steps will not be visible. The application developer can use this tradeoff to tailor an application depending upon what level of detail users are interested in sharing.

In our experience, these measures have been extremely useful in coordinating data access, detecting corrupted data, and dealing with lock contention.

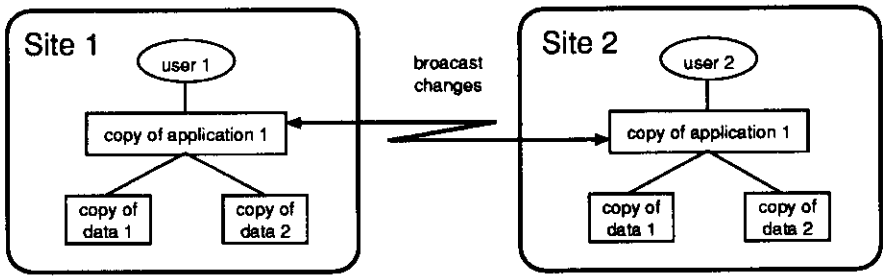

Figure 3: Multi-user Application in Object World

5 Converting Single-User to Multi-User Applications

Converting existing single-user applications to multi-user applications is greatly facilitated with *Object World*. The overall strategy is to make an application and its data be strongly shared. All interested sites should have copies of an application and its data, and any action taken by the application at one site should be made known to the other sites. So whenever a user provides input to a strongly shared application, that input or its effects are broadcast to all copies of the application so that all the copies have the same actions happen to them (see Figure 3).

Using *Object World*, the procedure for converting single-user applications involves a three step process:

1. *The application's objects and data need to be made sharable.* This just involves having the base classes of the application and its objects inherit from the *sharable* class. Inheriting from the *sharable* class causes all applications objects to inherit methods for encoding, transmitting, and decoding.

2. *Appropriate methods need to broadcast.* Methods can be made to broadcast by using the macro, *defbroadcast*. Application developers need only replace the standard method definition macro, *defmethod* with *defbroadcast* for those methods that should have their behavior shared. By determining which methods should broadcast and which should not, the application developer can control which operations are local and which are made public. For example, an application developer may not want rubberbanding to be visible to all users. In that case, the rubberbanding method should not be made a broadcast method.

3. *The application needs to be registered with a local conference manager.* All shared applications are started through the local conference manager so that it can inform other sites about the existence of each application. This is necessary when other users are interested in joining (sharing) that application.

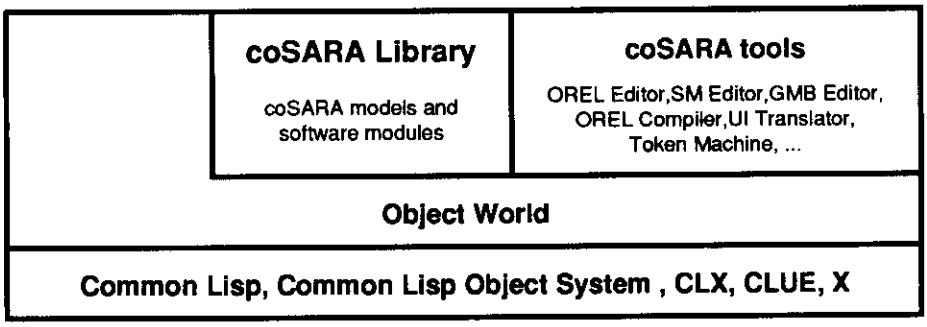

Figure 4: coSARA Architecture

Object World was used with the above process to convert a complex single-user research design system SARA [ESTR86] to a strongly shared, multi-user system, coSARA. SARA is a system developed by Estrin et al. for modeling and analyzing concurrent systems. The SARA system consists of a set of applications for graphically specifying concurrent systems in a hierarchical fashion using both top-down and bottom-up design. The SARA toolset incorporates tools for analyzing those systems. Due to the concurrent nature of synchronous group applications, coSARA has been extended to be a system for prototyping such applications.

6 The coSARA System

The coSARA system [MUJI91] provides a graphical approach to prototyping synchronous group applications. Users create strongly shared models of an application where these models are graphical specifications of the application. These models have the special feature that they are executable. By graphically programming an application in a strongly shared environment, the resulting application supports multiple users. The coSARA design methodology is described in more detail in the Section 7. In this section, an overview description of the coSARA system is presented.

The coSARA system (see Figure 4) consists of three main components: (1) the Object World infrastructure, (2) the coSARA tools, and (3) the coSARA library. The Object World infrastructure provides a framework for realizing strong sharing. Models created with coSARA are strongly shared so that multiple users are able to access those models. As changes are made to the states of the model, all users sharing the model see those changes in real time.

The coSARA tools are a set of design and analysis tools used for prototyping strongly shared applications. An application's data structures are graphically specified using the OREL (Object RELation) editor [MUJI91] (for an example see Figure 6(b)). An applica-

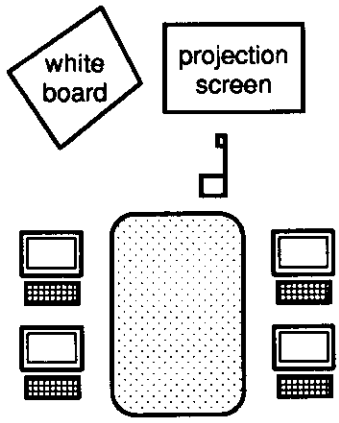

Figure 5: coSARA Environment

tion can be decomposed hierarchically into a series of interconnecting blocks and subblocks using the SM (Structure Model) Editor [ESTR86] (for an example see Figure 7(a)). The behavior (functional semantics) of the individual building blocks of the SM model can be specified using the GMB (Graphical Model of Behavior) Editor [ESTR86] (for examples see Figures 7(b),(d), and (e)). A resulting application is then executed using the Token Machine [ESTR86]. The resulting application allows multiple users to use it since its executable specification (models) are strongly shared. All the coSARA tools are also strongly shared and therefore allow multiple designers to work together in prototyping these strongly shared applications.

The coSARA library consists of a set of predefined strongly shared models and software modules useful for prototyping strongly shared applications. When building a strongly shared application, the application developers do not have to build everything from scratch; they can select common functions and models from the library such as a menu module and modules for creating and manipulating graphical objects.

The coSARA system is used in UCLA's Collaborative Design Laboratory (see Figure 5). This laboratory consists of four SUN workstations in a room around a conference table. Remote users outside the conference room can also join the conference but communication needs to be supplemented with audio and/or video supports. Images on a workstation can be projected onto a large projection screen at the foot of the conference table. This allows everyone to focus on a common screen instead of their workstations where they may easily get distracted by other things such as incoming e-mail messages.

7 Generating Strongly Shared Applications

The coSARA tools and library can be used to prototype strongly shared applications using the *coSARA Design Methodology*. This methodology works in two major steps: (1) constructing formal, graphical models of an application; and (2) linking the models with the coSARA library software modules and actual windows. The completed structure can then be executed.

The first step is based on three formal, graphical modeling languages, each supported by an appropriate editor: OREL (Object RELation) [MUJI91], SM (Structural Model), and GMB (Graph Model of Behavior) [ESTR86], which respectively are used to specify an application's data model, structure, and behavior. This next step is supported by tools as follows: (1) the GMB Editor links executable software modules to the GMB models, (2) the User Interface Translator links real windows to the application's user interface specification, and (3) the Token Machine executes an application by interpreting its GMB models. The resulting application is strongly shared. This is because all the objects of the application are built using the Object World infrastructure. Each user is able to join such an application by requesting the application's objects and models. Requesting just the main application object results in the creation of a copy of all objects and models of the application at the requesting site.

The coSARA methodology has six main attributes that make it useful for prototyping strongly shared applications:

1. The coSARA methodology has its roots in modeling concurrent systems [ESTR86]. This ability is useful for modeling strongly shared applications since such applications can have multiple user operations occurring at the same time. For example, multiple users may be independently, but simultaneously creating a block in a block diagram editor. To capture this feature, the coSARA modeling languages are able to represent multiple threads of control with the Petri net like GMB model.

2. The models are graphical. This facilitates specifying and understanding the concurrency and interactions that can occur in strongly shared applications. Users are able to see graphically the concurrency from the multiple threads of control rather than having the concurrency buried in some textual description. This makes it easier to define complicated interactions and identify potential problems.

3. The methodology uses primarily a single-user focus. Application developers do not have to concentrate on the intricacies of multi-user interaction. The multi-user attributes are provided as a benefit of the application objects being strongly shared. This makes it easier for them to build shared applications. They do not have to learn a completely new paradigm for application building. Also because the applications are designed

primarily using a single user focus, it should be easier for users to build a conceptual model of the application. This can make it easier to learn how to use the application and thereby be more acceptable to the users.

4. The methodology requires formally specifying the application. This provides structure to the design process and makes it easier for tools to detect errors in the design. coSARA analysis tools are provided to reveal problems.

5. coSARA supports a rapid prototyping paradigm. Application developers can easily make changes to their application models and then execute those models to see how those changes affects the application's functionality and behavior.

6. coSARA is a multi-user system and therefore allows multiple application designers to work together in prototyping strongly shared applications. They can build and study an application together and thereby be more effective in discovering errors.

Using the coSARA system, we have built strongly shared applications such as a multi-user graphical browsing tool and a multi-user block diagram editor.

7.1 coSARA Design Methodology

In the coSARA design methodology, strongly shared applications are seen as consisting of three major components: the *semantics* is the collection of methods that implement the services provided by the application; the *user interface* is the component that allows and controls the interaction between the users and the application's semantics; and the *data model* is the collection of data structures and their basic manipulation methods needed to support the operation of the application's semantics and user interface. The methodology works in six steps:

1. Graphically specify the application's data model;

2. Graphically specify the structure of the application's semantics and user interface;

3. Graphically specify the behavior of the application's semantics and user interface;

4. Link the models to software modules from the coSARA Library (and code any application specific modules not yet in the library);

5. Instantiate the application and its user interface objects and link them to the models; and

6. Execute and debug the application with the coSARA Token Machine and analysis tools.

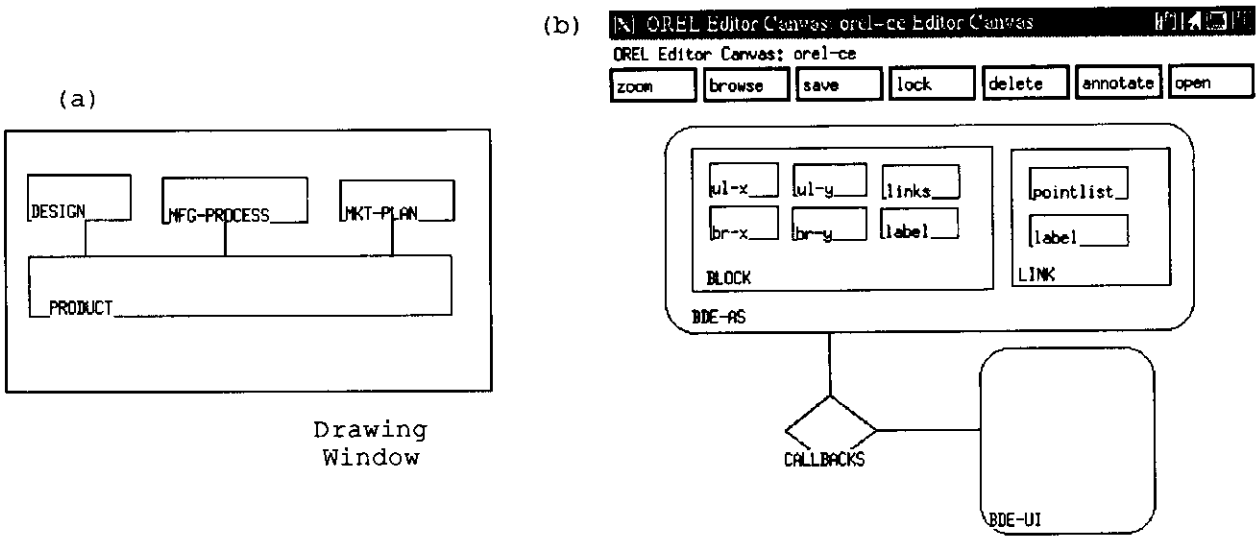

Figure 6: The block diagram editor: (a) during operation; (b) OREL model.

We illustrate the methodology by presenting these steps as they were applied in the design of a simple, strongly shared block diagram editor (BDE). This editor allows multiple users to concurrently draw and move rectangular blocks and the links between them. It communicates with the users through a drawing window and a dialog box. Fig. 6(a) shows a sample drawing window using the block diagram editor.

Operationally, a block is drawn by pushing down the mouse's left button at a selected upper lefthand corner, moving the mouse to a selected bottom righthand corner, and releasing the button; a link is drawn by a sequence of single-clicks of the left button starting at the origin of the link and ending with double-click at the termination. As soon as a new block or link is drawn, the editor opens a small dialog box in which the user types in the figure's label. Both blocks and links are moved by pressing down the mouse's right button inside a block or on a link's segment, dragging the object with the mouse to a new location, and releasing the button. This brief description will serve as the basis for determining the editor's data, semantics, and user interface models in the next subsections.

7.2 Modeling the Application's Data

The data model of a strongly shared application is a description of the collection of classes and their manipulation methods needed to support the operation of the application. OREL, a graphical object-oriented modeling language, which incorporates relations and is supported by the OREL graphical editor, provides six primitives to specify (1) simple classes (graphically represented by a rectangle), (2) composite classes (rounded rectangle), (3) recursive

composite classes (rounded rectangle with a shadow), (4) class attributes or slots (rectangle with lowercase letters), (5) relations among classes (diamond), and (6) class inheritance (displayed as a parenthesis list next to the class name).

Fig. 6(b) shows an OREL editor with the OREL model for the block diagram editor. It is constructed as follows: the application's semantics and user interface are represented as two top-level composite classes (`BDE-AS` and `BDE-UI`) participating in the same relation (`CALLBACKS`); each type of object handled by the application becomes a component class (`BLOCK` and `LINK`) of the composite class `BDE-AS`; and objects' properties (`label`, `links`, `pointlist`, etc.) become attribute slots in these component classes. For simplicity, we do not detail the class representing the user interface (`BDE-UI`). The relation `CALLBACKS` indicates that each instance of the application uses an instance of the user interface to communicate with the users.

The OREL compiler translates an OREL model into the appropriate class definitions, and the necessary methods for creating instances, assigning values to their slots, adding or removing the component objects of a composite object, etc. Common Lisp Object System (CLOS) code is produced and stored in the coSARA Library.

7.3 Modeling the Application's Structure

The structural model of a strongly shared application is a representation of the hierarchy of software components implementing the application. SM, a language supported by the SM graphical editor, provides three primitives to describe such structures: (1) modules representing application components (graphically represented by unfilled rectangles), (2) sockets representing the modules' communication ports (filled rectangles), and (3) interconnections representing connections between the modules' sockets (lines segments).

The structural model of the block diagram editor is shown in Fig. 7(a). Starting with a top-level module with no sockets (`BDE`, which represents the application itself), the application's software structure is hierarchically decomposed by refining modules into submodules, until the behavior of each module is precise enough to be represented by a single GMB model (as exemplified by `Interactor`, `DoRect`, and `Block`). `BDE` contains submodules `BDE-AS` and `BDE-UI`, representing the application's semantics and user interface. The components of `BDE-AS` and `BDE-UI` are discussed below.

7.4 Modeling the Behavior of the Application's Semantics

The semantics component of a strongly shared application is a representation of the methods implementing the services provided by the application. An application's semantics module

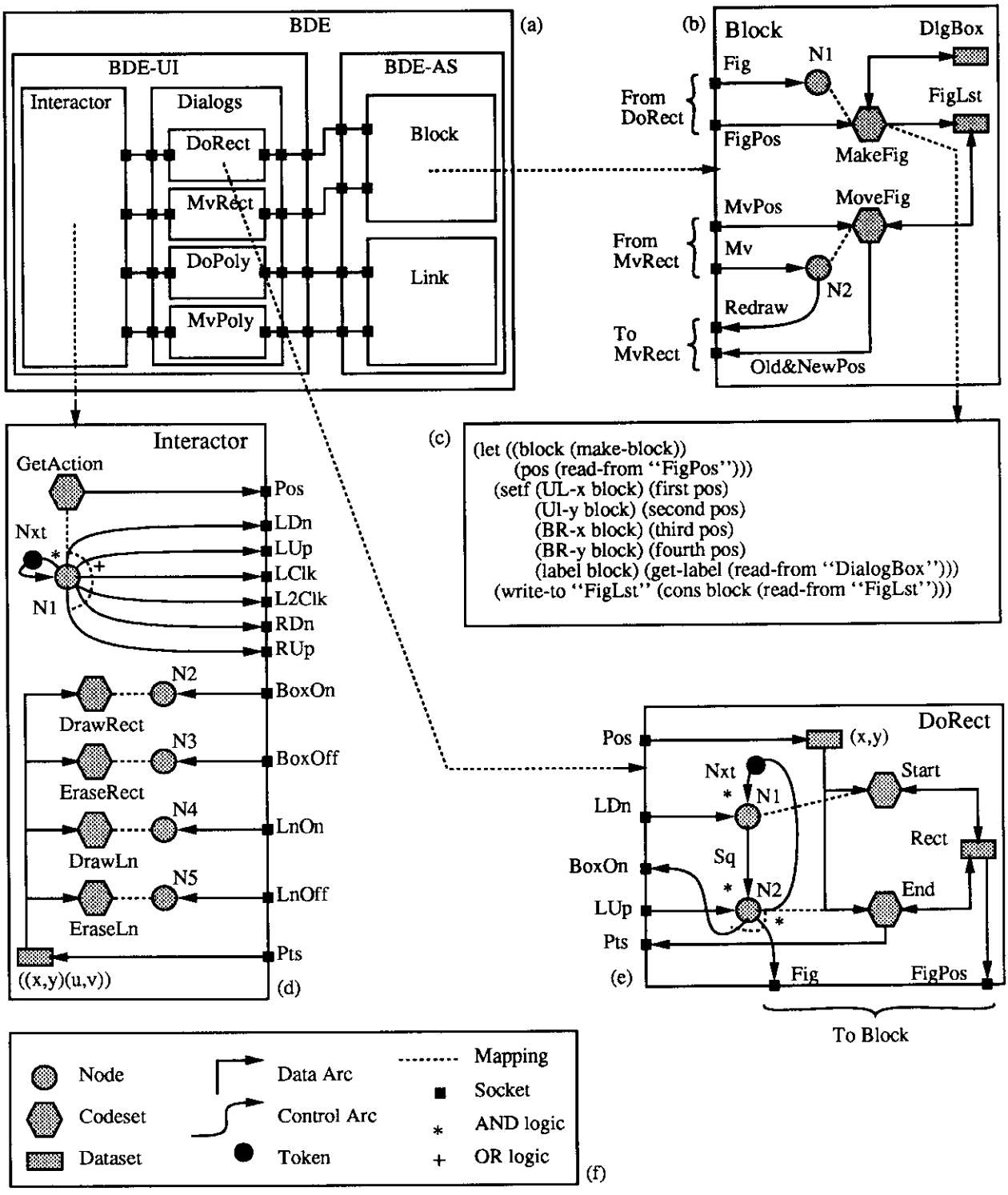

Figure 7: The models of the block diagram editor: (a) BDE Structural model; (b) Behavior (GMB) model of module `Block`; (c) Interpretation for codeset `MakeFig`; (d) Behavior (GMB) model of module `Interactor`; (e) Behavior (GMB) model of module `DoRect`; (f) Legend for behavior (GMB) models.

contains one submodule for each component class in the data model. In Fig. 7(a), submodules `Block` and `Link` in module `BDE-AS` represent the ability of the block diagram editor to manipulate blocks and links. The behavior of these submodules is specified in terms of GMB models.

GMB, supported by the GMB graphical editor, models three related aspects of the behavior of an application: (1) a control graph models the token driven flow of control among the events (circular nodes) that occur in the application, similar to a Petri net; the partial ordering of their activity is determined by logical constraints among directed control arcs connecting them; (2) a data graph models the flow of data between program segments (hexagonal codesets) and data storages (rectangular datasets); the type of access of the codesets over the datasets is determined by directed data arcs (writing to or reading from datasets) connecting them; and (3) an interpretation associated with the data graph describes the values stored in the datasets and the computations implementing the activity of the codesets.

The GMB model of each semantic submodule contains datasets for storing all the instances of the component class, and codesets representing the methods that can be applied to those instances. For example, the GMB model for `Block` is shown in Fig. 7(b). It includes the necessary behavior to create instances of blocks (codeset `MakeFig`) at specific positions on the drawing window, to store them (dataset `FigLst`), and to move them to different positions (codeset `MoveFig`). The meaning of a GMB model is defined by a token machine, as explained in Section 7.7.

7.5 Modeling the Structure and Behavior of the Application's User Interface

The user interface of a strongly shared application is the component that allows and controls the interaction between the users and the application's semantics. It is specified as a collection of *interactors* connected to a collection of *dialogs* [ETER92]. For example, in Fig. 7(a), module `BDE-UI` contains submodules `Interactor` and `Dialogs`.

Interactors are abstractions of things such as buttons, menus, dialog boxes, scroll bars, drawing windows, etc., based on the point-and-click paradigm of user interfaces. An interactor models the user actions to which it responds (e.g., pressing a key on the keyboard, clicking or moving the mouse, etc.), or the responses that it produces due to requests received from the dialogs (e.g., highlighting a screen region, drawing or erasing a figure, etc.). *Dialogs* model the syntax of the communication between the interactors and the application's semantics, specifying which sequences of actions received from the interactors are valid and the points in these sequences at which information is sent back to the interactors. Each dialog describes one valid sequence of actions.

Interactors and dialogs are represented by SM modules, which communicate via sockets and interconnections. Their behavior is described using GMB models. The user interface of the block diagram editor contains one interactor and four dialogs. Module `Interactor` in Fig. 7(d) represents the drawing window where users perform all the actions and observe the responses to them. The actions result in tokens placed on node `N1`'s output control arcs: `LDn`, `LUp`, ..., `RUp`. The responses are the result of the activation of the codesets `DrawRect`, ..., `EraseLn` to draw and erase rectangles and lines. Modules `DoRect`, `MvRect`, `DoPoly` and `MvPoly`, in Fig. 7(a), define the action sequences required to define and move rectangles and polylines. The behavior of `DoRect` is shown in Fig. 7(e). Modules representing interactors and dialogs that behave in this way can be obtained from the coSARA Library, or they can be defined by the UI designers. More information on the interactor/dialog model of user interfaces can be found in [ETER92].

7.6 From Models to Executable Prototypes

After building the models of the application, the next step of the coSARA methodology involves instantiating the application and user interface objects and linking them to the models. The interpretation software modules (i.e., the executable code defining the functionality of the codesets and the data types of the datasets) for the GMB models of an application reside in the coSARA Library and are linked to the models using the GMB graphical editor. This tool requests the modules' names from the designer, looks them up in the coSARA Library, and links them to the corresponding codesets and datasets. The coSARA Library contains a collection of general purpose software modules, as well as modules generated by the OREL compiler from the data models. Any application specific module which is not yet part of the Library and is not generated automatically by the OREL compiler has to be coded by the designer and stored in the Library before linking it to the models.

At this point, the models represent a complete formal specification of the application. The application can be analyzed using coSARA's analysis tools [ESTR86]. The Control Flow Analyzer can be used to analyze the models for pathologies of the form of non-proper terminal states, deadlocks, potentially infinite states and critical transitions. The Performance Analyzer, which uses a queueing model, can be used to evaluate the possible performance with multiple users.

An executable application object can be created now by instantiating the application class and linking it to the models. As we said before, the class definition and the function to instantiate it are produced by the OREL compiler from the application's data model. Before the application object can be executed, the user interface's interactors have to be linked to actual windows. The User Interface Translator uses the interactor modules to create the desired windows. In the case of BDE, only one window is generated based on the specifications of module `Interactor` (see Fig. 7(d)). The resulting windows are linked

to the user interface's interactors during the installation of the application object. Now the application object can be executed by the system's token machine.

7.7 Executing the Application

The *token machine* is an interpreter of GMB models. Each node in a GMB model has a set of inputs and outputs. Associated with each node is one codeset. The inputs define the required distribution of tokens in the node's input control arcs for the node to fire. Firing a node causes the tokens on the input control arcs to be removed and the interpretation of the codeset mapped to the node to execute. When the interpretation finishes its execution, tokens are placed on the node's output control arcs. We describe now the activities that take place in the block diagram editor model when a user draws a rectangle, defining a block.

To draw a rectangle on the drawing window, a user has to push down the mouse's left button, move the mouse, and release the button. The interactor in Fig. 7(d) represents the drawing window. When the user pushes down the button and then when the button is released, codeset`GetAction` sends the action's window position through arc `Pos`, and causes a token to be placed on one of `N1`'s output arcs. When the user pushes down the button, the token is placed on arc `LDn`; when the button is released, the token is placed on arc `LUp`.

The tokens and data produced by the interactor travel to dialog `DoRect`. It is `DoRect`, as shown in Fig. 7(e), that requires a sequence of two actions to produce a rectangle. After the second action is received, `DoRect` informs the semantic module `Block` that a new rectangle has been defined. Codeset `End` stores the positions of both actions in dataset `Rect` and causes node `N2` to place a token on arc `Fig`. `DoRect` also informs the interactor that the rectangle was defined, so that the interactor can draw it on the screen. Codeset `End` sends both positions through arc `Pts` and causes node `N2` to place a token on arc `BoxOn`; this token eventually activates codeset `DrawRect` in the interactor, which does the drawing.

In `Block`, as shown in Fig. 7(b), the token on `Fig` fires node `N1` thereby activating codeset `MakeFig`. `MakeFig`'s interpretation, shown in Fig. 7(c), consists of a sequence of calls to the CLOS methods produced by the OREL compiler, which are stored in the coSARA Library. First, `MakeFig` creates a new block, i.e., a new instance of the class `Block`. Then, it gets the values of the various slots (see Fig. 6(b)) of this new block: it reads the positions of the rectangle's upper-left and bottom-right corners through arc `FigPos`, and stores them into the slots `UL-x`, `UL-y`, `BR-x` and `BR-y`; and it opens the dialog box in dataset `DlgBox`, asking the user for a label, and stores the user's reply into the slot `Label`. Finally, `MakeFig` stores the new block by adding it to dataset `FigLst`.

7.8 Strongly Sharing the Application

It is important to note that the entire coSARA system is built on top of the *Object World* infrastructure. The OREL, SM, and GMB editors, and the token machine are all strongly shared applications because of *Object World.* This means that the models, the instantiated application and its user interface objects, and the actions of the token machine are all strongly shared. By using the software modules in the coSARA Library, which also uses the Object World, the application's operations are shared. This results in the prototyped application itself being strongly shared. Given one instance of the application, other users are able to request the application. Such a request results in the creation of a copy of all the application's objects (e.g., the application object, the user interface objects, and the models) at the requester's site. It should be clear by now that a group of designers can build and study these shared models together and thereby be more effective in discovering errors before a system is built.

Each user that shares the application is then able to use it. Actions by the users are broadcast to all the sites having a copy of the application. As each user operates the application, tokens are generated in the model. Multiple threads of control (i.e. - multiple tokens flowing through different parts of the models) can occur for example when one user is creating a block in one location, while another is moving an existing block and another is creating a block in a third position. Each user has an associated token flowing through the models. The graphical models allow the application developers to develop group applications that allow intricate overlapping threads where multiple users can contribute to a single thread of control. For example two users can work together in defining a block where one user draws the shape of the box and the other user provides the block's name. Most existing group applications only allow one user per thread. The graphical models also allow applications developers to see the multiple threads and the interactions that can occur among the users. This is useful for identifying potential user conflicts in the application.

By changing the graphical models, applications are able to quickly modify the behavior of an application. This allows the developers to explore alternative implementations.

The granularity of multi-user interaction is determined by the codesets. It depends on which methods in the codesets broadcast themselves and which do not. Executing non-broadcast methods results in the actions occurring locally at the site invoking the method. Executing broadcast methods results in the actions occurring at all sites. For example, the dialog to create a block (`DoRect`) consists of pressing the mouse button (followed by moving the mouse) and releasing the mouse button (see Fig 7(e)). Associated with the actions are software modules to be executed (the interpretations associated with codesets `Start` and `End`). If both software modules broadcast, then all users would see the intermediate steps when one user creates a rectangle. By only having the last module broadcast, the other users would only see the final creation of a rectangle. The latter approach is used by the block diagram editor example.

8 Summary and Conclusion

Through the development of the coSARA system, we have provided a different perspective for understanding synchronous group applications through our strong sharing model. Synchronous group applications can be viewed as strongly shared applications where actions of each site are made known to the other sites. This insight led us to build a strong sharing platform, called *Object World*, using Common Lisp and CLOS (Common Lisp Object System). This platform provides shared data through replication, shared operations through broadcast methods, coordination support through a variety of features such as locking and automatic site consistency checking, and conference management support through a global conference server and local conference managers. Key features of *Object World*, besides its support for strong sharing, include the ability to support replication on demand, the ability to support remote sites not on an immediate LAN, and coordination support through a consistency checker and locking mechanism.

Object World provides an easy methodology for converting existing single-user applications to multi-user single applications. A single-user modeling and analysis system for concurrent systems, SARA was easily converted to the multi-user system, coSARA using this methodology. We then modified coSARA to support prototyping multi-user applications. Such applications are built by specifying graphical models of the application and then linking software libraries. coSARA has six key attributes:

- coSARA's modeling languages were originally designed for general concurrent systems. The models were easily extended to handle multi-user applications since they have a concurrent nature to them.
- The graphical coSARA models facilitates specifying and understanding the concurrency and interactions that occur in multi-user applications.
- The underlying strong sharing framework of coSARA allows using primarily a single-user focus in building new applications. By the nature of strongly shared objects, the multi-user attributes are easily obtained.
- The coSARA methodology enables application developers to formally specify their applications. This makes it possible to analyze the applications for concurrency anomalies.
- coSARA supports a rapid prototyping paradigm. Application developers can easily make changes to their application models and then execute those models to see how those changes affects the application's functionality and behavior.
- coSARA is a multi-user system and therefore allows multiple application designers to work together in prototyping strongly shared applications.

Synchronous group applications cannot be the only vehicle for collaboration. The permissiveness of simultaneous parallel change by individual collaborators, carries with it the need for extensive coordination and means to repair damage to each others efforts when that coordination or protection fail. There is a full spectrum of sharing that needs study ranging from presentation of individual work to synchronous group applications. The model of strong sharing that we have explored and exercised is an important step because it allows us to analyze and understand sharing behavior more deeply and to seek models and methods that will let us support seamless change from a strong sharing mode to, say a "version sharing" mode to a "database sharing" mode to an isolated "single-designer" mode and back as needed by the designers.

References

[CART91] D. Carter and B. Baker, "Concurrent Engineering: The Product Development Environment for the 1990's", Mentor Graphics Corporation, 1991.

[CROW90] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson, "MMConf: An Infrastructure for Building Shared Multimedia Applications," Proceedings of the Conference on Computer-Supported Cooperative Work, 1990, pp. 329-342.

[ESTR86] G. Estrin, R. Fenchel, R. Razouk, and M. Vernon, "SARA (System ARchitect Apprentice): Modeling, Analysis, and Simulation Support for Design Of Concurrent Systems," IEEE Transactions on Software Engineering, SE-12, Feb 1986, pp. 293-311.

[ETER92] Y. Eterovic, "Executable Specifications of Multi-Applications Multi-User Interfaces," Computer Science PhD Dissertation, University of California - Los Angeles, June 1992.

[GRAH92] T. C. N. Graham and T. Urnes, "Relational Views as a Model for Automatic Distributed Implementation of Multi-User Applications," Proceedings of the Conference on Computer-Supported Cooperative Work, 1992, p. 59-66.

[GREI88] Edited by Irene Greif, Computer-Supported Cooperative Work: A Book of Readings, San Mateo, CA, Morgan Kaufmann Publishers, 1988.

[MUJI91] S. Mujica, "A Computer-based Environment for Collaborative Design," Computer Science PhD Dissertation, University of California - Los Angeles, 1991.

[PATE92] D. Patel and S. Kalter, "A Toolkit for Synchronous Distributed Groupware Applications," Groupware '92, David Colemen (ed.), Morgan Kaufmann Publishers, San Mateo, CA, 1992, pp. 225-227.

[PATT90] J. Patterson, R. Hill, S. Rohall, and W. S. Meeks, "Rendezvous: An Architecture for Synchronous Multi-User Applications," Proceedings of the Conference on Computer-Supported Cooperative Work, Oct. 1990, pp. 317-328.

[ROSE92] M. Roseman and S. Greenberg, "GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications," Proceedings of the Conference on Computer-Supported Cooperative Work, 1992, pp. 43-50.

[STEF87] M. Stefik, G. Foster, D. Bobrow, K. Kahn, S. Lanning, and L. Suchman, "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving Meetings," from Computer- Supported Cooperative Work: A Book of Readings, edited by Irene Greif, San Mateo, CA, Morgan Kaufmann Publishers, 1987, pp. 335-366.

[WU91] E. Wu, "Concurrency Control and Remote Sharing for a Replicated Collaborative Environment," UCLA Computer Science Department Technical Report, No. 910065, 1991.