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

GENERATING STRONGLY SHARABLE CONCURRENT ENGINEERING APPLICATIONS

S. Berson
G. Estrin
Y. Eterovic
I. Tou
E. Wu
S. Mujica
D. Berry

June 1992
CSD-920032

Generating Strongly Sharable Concurrent Engineering Applications [1]

Steven Berson, Gerald Estrin, Yadran Eterovic, Ivan Tou
Computer Science Department
University of California
Los Angeles, CA 90024-1596
(310)825-2786

Elsie Wu, NCR Corporation(San Diego)
Sergio Mujica, Universidad de Santiago de Chile
Daniel Berry, Berry Computer Scientists (Haifa, Israel)

March 10, 1992

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

Abstract

Central to the mission of concurrent engineering is the early discovery and correction of problems that may arise during the life cycle of a product. To achieve this mission, an environment supporting the *strong sharing* of applications and data, and their creation is essential. This environment must allow multiple collaborating users to share the same data through one or more shared applications. Changes made by one user are to be seen immediately by the other users. Such an environment provides a foundation for critiques and debates among all the parties involved in requirements analysis, specification, design, manufacturing, maintenance and product changes.

This paper describes the results of research in the UCLA Collaborative Design Laboratory in which we have built such an environment, called coSARA, to support strong sharing and to support prototyping strongly sharable concurrent engineering applications. CoSara realizes strong sharing by extending Common Lisp objects to support replicated objects and broadcasting methods. The coSARA system allows users to prototype a strongly sharable 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 demonstrates how the coSARA methodology is used to build a strongly sharable block diagram editor.

Keywords: Concurrent Engineering, CSCW, Groupware, Sharable Applications, Multi-user Interfaces, Graphical Programming, Executable Specification.

1 Concurrent Engineering and Strong Sharing

The terminology, *concurrent engineering* was first reported in 1986 in the Institute for Defense Analyses (IDA) Report R-338. That report defines concurrent engineering as:

> "...a systematic approach to the integrated, concurrent design of products and their related processes, including manufacture and support. This approach is intended to cause the developers, from the outset, to consider all elements of the product life cycle from concept through disposal, including quality, cost, schedule, and user requirements."

Concurrent engineering is an organizational activity involving people of varying disciplines in a product development enterprise. The chief goals of concurrent engineering applications are to reduce time-to-market, increase quality, and decrease life cycle cost. Since people's activities and decisions are interdependent, we must make it easier for people to work together, sharing ideas, getting critiques, making suggestions, and informing others.

Communication and coordination support are the keys to successful concurrent engineering. If an environment helps to expose progress and enables timely feedback, product team members can argue with each other, are more likely to expose weaknesses, and can work to improve a product as it progresses from abstract representation to physical reality and beyond to its eventual obsolescence. Errors and poor decisions may then be more readily detected and corrected early in the product development process.

A fundamental requirement for concurrent engineering applications is sharing without introducing a priori restrictions on that sharing. In particular we should not simply accept the notion of restricted sharing usually found in a database. Multiple users should be able to work closely together sharing common data and using the same or several different applications. This means that users should be able to see in *real time*, the decisions and actions of each other as reflected in changes to the data. By doing so, users should be able to work together, tracking each other's work, transferring information among themselves sooner, and making decisions and suggestions sooner, so as to improve the product development. This required style of sharing is what we call *strong sharing.*

Strong sharing has two aspects: strong sharing of data and strong sharing of applications. As shown in Figure 1a, *strongly shared data* would allow multiple users to access and modify the same data concurrently using different applications. As one application modifies the data, the other applications are able to see the results immediately. For example, a marketing person could use a cost analysis spreadsheet application at the same time as a manufacturing person uses an inventory application and a designer uses a design parts program. As the designer selects the parts to be used in a new product, the spreadsheet program shows the

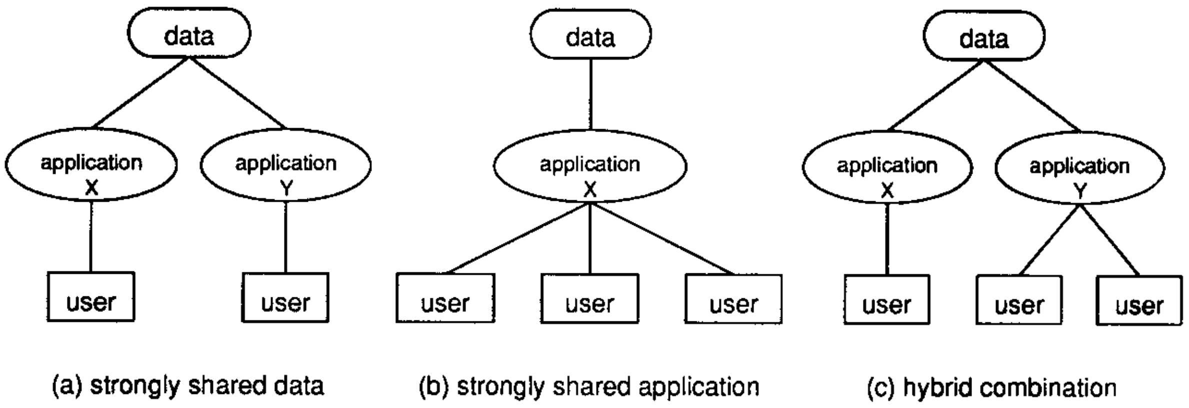


Figure 1: Strong Sharing

change in cost and the inventory application shows how many of those parts are in stock. Thus the designer, manufacturer, and marketing person are able to discuss the tradeoffs of using various parts for the new product.

As shown in Figure 1b, a *strongly shared application* would allow multiple users to concurrently use an application and its data. Each user has the same view of the application and any action taken by a user is seen by all the users. Some examples of strongly shared applications useful for concurrent engineering include a brainstorming tool for capturing and coordinating group ideas, a shared sketch editor and a shared block diagram editor for product teams to define a high level definition of a product, navigation tools for product teams to jointly navigate through vast product data and to set up a shared focus for group discussions, a shared history tool for facilitating annotations of design objects and design processes, and a voting tool for facilitating group decisions or determining the opinions of the group.

Combinations of strongly shared data and strongly shared applications can be used to generate hybrid multi-user configurations as shown in Figure 1c. Multiple users are able to interact using the same strongly shared data. Some of these users run independent single-user applications. Others may even be working closer together, sharing the same application.

1.1 Barriers to Concurrent Engineering

Three major barriers to effective collaboration are the time barrier, the place barrier, and the work barrier. The time barrier has two aspects. The first aspect deals with the accessibility of data on demand. When a group of people are collaborating on or reviewing a product design and one of them locks the data, all others that need this data are forced to wait until the lock is removed. Such delays can inhibit reasonable interaction between the individuals. The second aspect has to do with the feasibility of sharing the dynamic aspect of the data. If a product design were questioned as to whether the design was meeting prescribed requirements, the designer might want to show a simulation experiment to the members of the group to convince them of the design's validity. Another member of the group might want to suggest a "what if" experiment. The project may suffer if such data sharing can not be done in a timely fashion.

The place barrier deals with being able to work together and share across different locations. There is a universal desire to reduce the need for physical movement from one location to another in order to proceed with product development. Large delays are incurred because of travel necessary to coordinate activities of individuals. Even though we do not expect to achieve the same quality of interaction as occurs in face-to-face meetings, we expect that an environment providing stronger sharing, enhanced by proper protocols and multi-media facilities could attract an enthusiastic following.

The work barrier deals with the sequential nature of work. In planning any product development, we seek opportunities to divide tasks among as many individuals as we can afford and then through coordination, bring their results together. Such parallelism always carries a risk that individual subsystems may be incompatible with each other. We do not want to destroy potential parallelism by forcing each participant to complete his or her task and then be forced to evaluate the results one at a time to discover inconsistencies. We want to be able to observe progress in each other's work and to be able to flag inconsistencies as soon as we notice them.

Strong sharing overcomes many of the problems associated with the time, place, and work barriers. With respect to the time barrier, strong sharing allows group members full access to the data at all times and it allows the sharing of the dynamics of the data. With respect to the place barrier, strong sharing allows users to work in remote locations and intimately share the same information limited only by inherent network delays. With respect to the work barrier, strong sharing allows users full access to the data and therefore full parallelism. Users no longer have to wait for information to be unlocked. Also since data is accessible as it is changed, incompatibilities are detectable early, thereby possibly making the final integration process smoother. We believe strong sharing to be essential for next generation concurrent engineering applications that will allow users to work more closely together. However software development tools and techniques need to be developed that help application builders create these strongly sharable concurrent engineering applications.

1.2 Requirements for Strong Sharing

In building a system that supports strong sharing and the generation of strongly sharable concurrent engineering applications, we have identified two basic requirements for strong sharing. They are:

1. *Shared Data* — The data must be accessible on demand by all users sharing the data. Users of the data also need to be able to save the data into some permanent storage so that the data extend beyond the life of an application.

2. *Shared Operations* — One needs to be able to write functions on the shared data such that the effects of those functions are seen by all users of the data. These sharable functions should operate in real time such that everyone sees the actions at the same time.

Strong sharing allows maximum concurrency, but trades off consistency. Multiple users are able to have constant access to the data. This allows users to see the data in inconsistent states and allows users to make conflicting actions on the data. In order to prevent this chaos, an environment supporting strong sharing will need to satisfy an additional requirement:

3. *Coordination Support* — Strong sharing systems need safeguards to prevent inconsistencies and to prevent the users from destroying each other's work.

Satisfying these three requirements allows one to achieve both strong sharing of data and strong sharing of applications. By definition, strong sharing of data is achieved when these requirements are satisfied since each user will be able to access the data (shared data requirement) and the actions on the data will be seen by all the users (shared operations requirement). How the requirements help realize strong sharing of applications is explained in the next subsection.

1.3 Strongly Sharable Applications

Strong sharing of applications can be achieved using the results of the strong sharing requirements. 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 becomes strongly sharable. 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. By making the application's methods be shared (the shared operations requirement), any action by a user is seen by all the other users. The coordination support requirement is then needed to help the multiple users operate the shared application in a non-destructive fashion.

Given an environment that supports strong sharing, if the users of a strongly shared application are indistinguishable from the perspective of the application (i.e. the application does not assign roles to the users), then a design methodology for single-user applications can be used to build that strongly sharable application. 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 are shared and made visible to everyone else. This is the approach we use in the coSARA system which is explained in more detail later.

1.4 Related Work

Two types of architectures, centralized and replicated, can be used to realize strong sharing [AHUJ90]. With *centralized architecture*, only one instance of the data or application is used and it is stored at a central site. For strongly shared data, all sites are able to access the data at all times from that central site. For strongly shared applications, all sites send their inputs to the application running at a central site. The application then sends the outputs to all involved sites. The chief advantage of this architecture is that synchronization and consistency are easy to maintain. The centralized version has several disadvantages though. It generates more network traffic since inputs and outputs need to be broadcasted to all the sites. The centralized system is sensitive to single point failures at the central site or in the network. The central site also acts as a bottleneck to the system.

With *replicated architecture*, each site keeps a copy of the data and applications. For strongly shared data, operations on the data are applied to all the copies. For strongly shared applications, each user's inputs are broadcast to all sites. The outputs are then handled locally by each of the replicated applications. The chief advantages of this approach are the reduced network traffic and the robustness of the system. The chief disadvantage is the difficulty in ensuring consistency across the different sites. The replicated architecture has two variations: (1) full replication where each site has a complete copy of all the shared data and applications, and (2) replication-on-demand where each site only has a copy of data and applications that the site is using.

One example system using full replication is CoLab [STEF87]. 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.

Full replication for a product development environment is however not practical. Each site would be required to have vast amounts of data and execute numerous applications. Running a copy of an application that one is not interested in, unnecessarily degrades the computer's performance. Product development environments require a more opportunistic approach where data and applications are shared only when needed. Our system, coSARA uses the replication-on-demand paradigm. This saves space and processing resources since only those data and applications of interest need be copied to a given site.

Little work has been reported on providing tools for generating strongly sharable applications. Most of the work has shown how to convert existing single-user applications to multi-user applications. The main approach taken uses a centralized architecture where a conference manager interface collects the inputs from different sites and passes them to the application [GREE90]. The conference manager interface then takes the output from the application and transmits it to all the involved sites. This approach does not support the strong sharing of data. Heterogeneous applications are not able to share the same data.

SharedX [GARF89] and COMIX [SRIN92] are two examples of the conference manager approach where both systems act as the conference manager interface. SharedX is an X window server tool that allows multiple users to share the same application. It provides strong sharing of an application. All users sharing the application see exactly the same windows and are able to use the application. COMIX is an application sharing server that also allows multiple users to share the same application. SharedX and COMIX suffer from the previously mentioned problems of using a centralized architecture and using a conference manager approach. They both also limit the way multiple users can interact with the shared application. The COMIX system only allows one user to use the system at a time where control is determined by a chalk passing protocol. SharedX either allows one user at a time access or a first come, first serve merging of all users' inputs. In contrast, strongly shared applications generated using coSARA overcome these problems by using a replicated architecture and by allowing application builders greater control over how multiple users interact with the application.

2 The coSARA System

Given the need for strongly sharable concurrent engineering applications, the next question becomes how does one generate those applications. We show that one solution is offered by the coSARA (collaborative Systems ARchitect Apprentice) system [MUJI91]. CoSara has its roots in the Systems ARchitect Apprentice (SARA) system developed by Estrin et al. [ESTR86]. Transformation of the SARA system to make it strongly sharable by multiple users, resulted in the coSARA system.

The coSARA system is both an environment for realizing strong sharing and an environment for prototyping strongly sharable concurrent engineering applications. The coSARA system (see Figure 2) 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. The coSARA tools are a set of design and analysis tools used for prototyping strongly sharable applications. The coSARA library consists of a set of models and software modules which are also used for prototyping such applications. The coSARA system is written in Common Lisp running in a UNIX and X Windows environment on a network of SUN workstations.

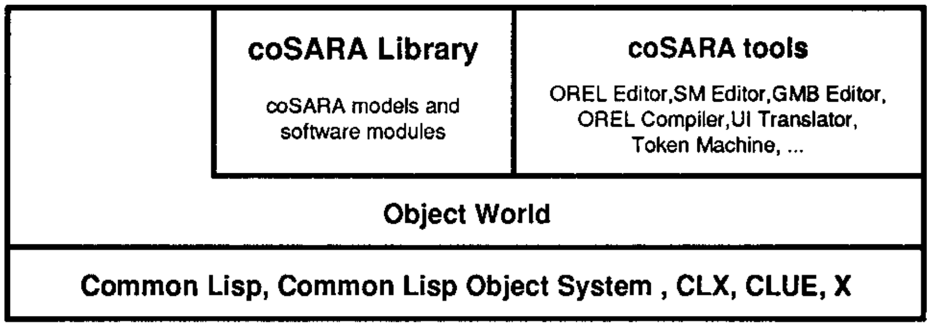

coSARA Library
coSARA models and software modules
coSARA tools
OREL Editor,SM Editor,GMB Editor, OREL Compiler,UI Translator, Token Machine, ...
Object World
Common Lisp, Common Lisp Object System , CLX, CLUE, X

Figure 2: coSARA Architecture

2.1 Realizing Strong Sharing

The Object World infrastructure provides the necessary support for realizing strong sharing of data and applications. It satisfies the strong sharing requirements (as discussed in Section 1.1) by supporting the sharing of data and their operations, and by supporting the coordination of the usage of the data.

Since objects provide a convenient and natural granularity for sharing, we used an object-oriented infrastructure for supporting strong sharing. That infrastructure is called Object World. This infrastructure defines a top-level object class, *sharable* that allows the sharing of its instances. This is accomplished by being able to transmit a copy of an instance from one site to another. All objects that inherit from this class will also be sharable. The shared objects are able to share their operations by broadcasting their method invocations to all sites with copies of the object. Such methods are called *broadcast methods.* By specifying whether a method broadcasts itself or not determines whether the actions are to be done globally on all sites or locally on a single site. This flexibility allows users to control the level of sharing. An operation may be composed of several intermediate steps. The level of sharing deals with whether those intermediate steps are broadcasted to all the copies or whether only the final resultant step is broadcasted. The end results are the same, but the amount of shared and local activity is different.

Applications built on top of the Object World infrastructure are able to strongly share their data. Those applications need to make their data be subclasses of the class *sharable*, and need to use the broadcast methods that manipulate the shared data. So whenever one application makes a change to the data, the other applications are notified of that change. This allows multiple users, using heterogeneous applications, to work together on the same data. We have tested this ability by being able to use independently, a graphical browsing tool and a graphical editor tool on the same design data. The browsing tool allows a user to navigate through the design in both high and low level views. The editor tool allows a user to build and modify the design. As the design is changed by the editor, the browsing tools reflects those changes immediately.

How strong sharing is realized is explained in more detail in section 4 on "Implementation of Strong Sharing."

2.2 Generating Strongly Sharable Applications

The coSARA tools and library are used for prototyping strongly sharable applications using the *coSARA Design Methodology.* This methodology works in two major steps: (1) constructing formal, graphical models of the application; and (2) linking the models with the coSARA library software modules and actual windows, and then executing the complete linked structure. 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 the application's data model, structure and behavior. The tools supporting the second step are (1) the GMB Editor, which links executable software modules to the GMB models, (2) the User Interface Translator, which links real windows to the application's user interface, and (3) the Token Machine, which executes the application by interpreting its GMB models.

The resulting application is strongly sharable. This is because all the objects of the application are built using the Object World infrastructure. The modules in the library also use the Object World infrastructure to achieve shared operations. Finally, the models of the application use the Object World infrastructure and are therefore strongly sharable. Thus, the application as represented by its objects (application object, data objects, and user interface objects) and its specification models are sharable.

Each user is able to join an application by requesting the application's objects and models. This results in the creation of a copy of all these objects and models at the requesting site. Requesting just the main application object is sufficient since it will retrieve all the other objects and models. Leaving a shared application consists of informing the other users that one is leaving and deleting the application from ones local workspace.

Strongly sharable applications allow any number of users to join and leave at any time. Each user appears no different from any other user to the application. However as the number of users increase, the coordination of the users and the load on the system become problematic. We recognize these problems and are currently only addressing strongly sharable applications for small groups of less than ten designers.

The coSARA methodology has three main attributes that make it useful for prototyping strongly sharable applications. First, it allows formally specifying an application. This provides structure to the design process and makes it easier to have tools for detecting errors in the design. Second, the coSARA methodology has its roots in modeling concurrent systems [ESTR86]. This ability is useful for modeling strongly sharable 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. Third, the models are graphical. This facilitates specifying and understanding the concurrency involved in strongly sharable applications. Users are able to see the concurrency graphically rather than having it buried in some textual description.

Using the coSARA system, we have built such strongly sharable applications, e.g. a group browsing tool and a strongly sharable block diagram editor. The following section demonstrates in more detail how the block diagram editor was constructed using the coSARA system and the coSARA design methodology.

3 Generation of Strongly Sharable Applications

In the coSARA design methodology, strongly sharable 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, as outlined in Fig. 3:

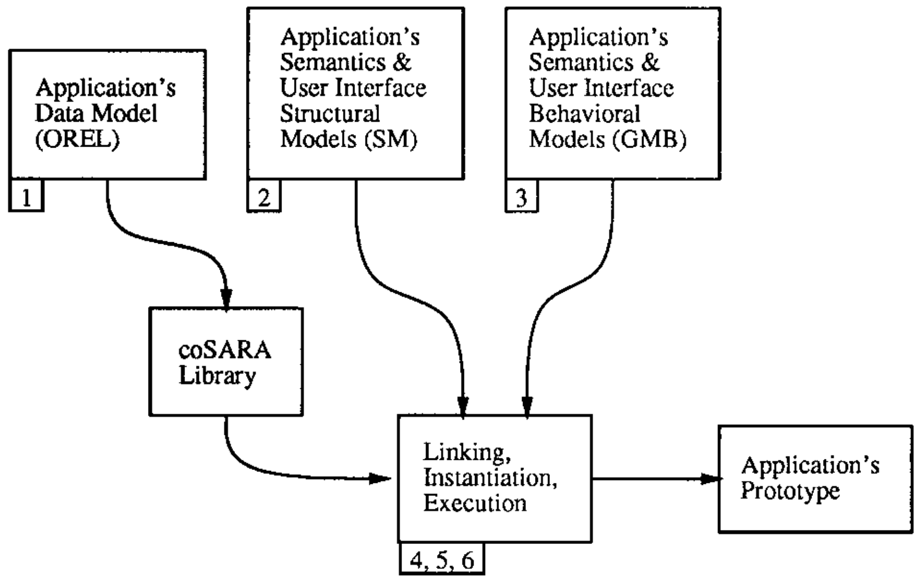

Application's Data Model (OREL)
1
Application's Semantics & User Interface Structural Models (SM)
2
Application's Semantics & User Interface Behavioral Models (GMB)
3
coSARA Library
Linking, Instantiation, Execution
4, 5, 6
Application's Prototype

Figure 3: Diagram of the coSARA design methodology (the numbers below the boxes refer to the steps in the methodology).

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.

We illustrate the methodology by presenting these steps in greater detail, as they were applied in the design of a simple, strongly sharable 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. 4(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 the upper lefthand corner, dragging the mouse to the lower righthand corner, and releasing the button; links are drawn by a sequence of single-clicks of the left button at the vertices of the link, ending with a double-click. 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 the 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.

3.1 Modeling the Application's Data

The data model of a strongly sharable application is a description of the collection of classes and their manipulation methods needed to support the operation of the application. OREL, an object-oriented modeling language, which incorporates relations and is supported by the OREL graphical editor, provides six primitives to specify (1) simple classes, (2) composite classes, (3) recursive composite classes, (4) class attributes (slots), (5) relations among classes, and (6) class inheritance.

Fig. 4(b) shows 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

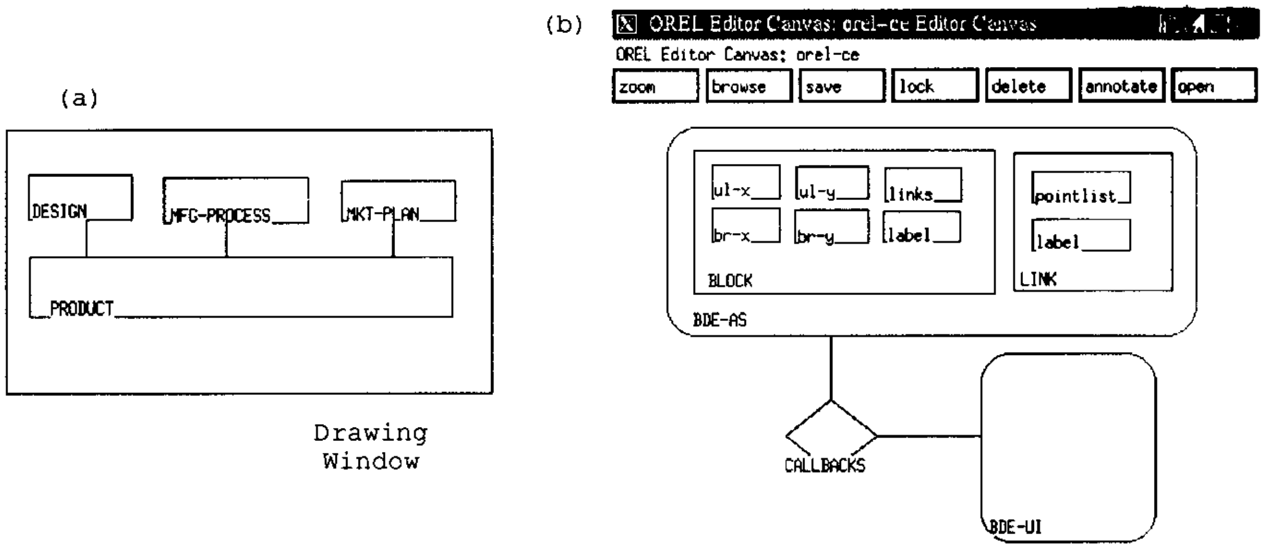


Figure 4: The block diagram editor: (a) during operation; (b) OREL model.

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 describe 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 the coSARA Library.

3.2 Modeling the Application's Structure

The structural model of a strongly sharable 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 represent application components, (2) sockets represent the modules' communication ports, and (3) interconnections represent connections between the modules' sockets.

The structural model of the block diagram editor is shown in Fig. 5(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. 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.

3.3 Modeling the Behavior of the Application's Semantics

The semantics component of a strongly sharable application is a representation of the methods implementing the services provided by the application. An application's semantics module contains one submodule for each component class in the data model. In Fig. 5(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.

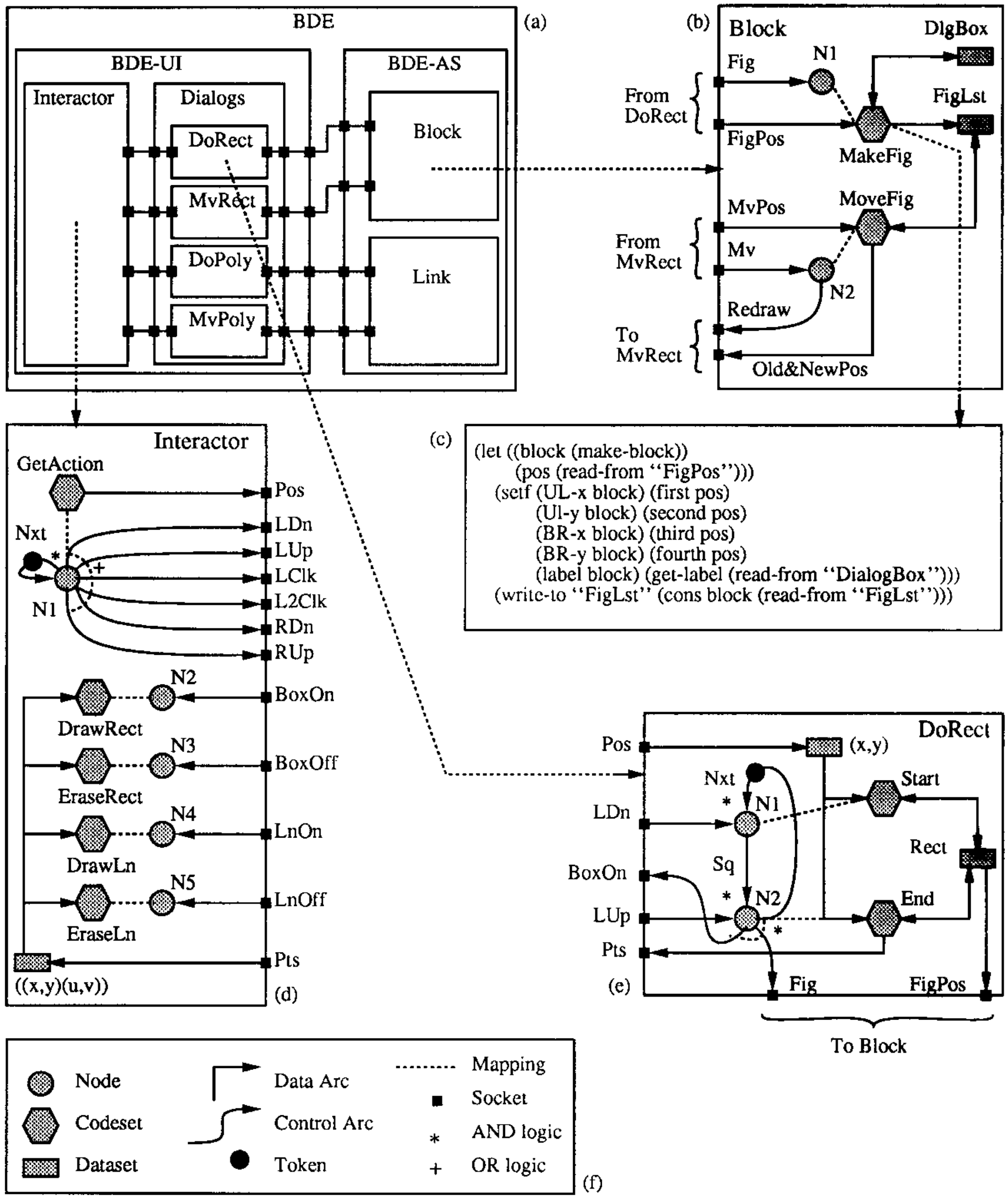
GMB, supported by the GMB graphical editor, models three related aspects of the behavior of an application: (1) a control graph models the flow of control among the events (nodes) that occur in the application, similar to a Petri net; the partial ordering of their activity is determined by directed control arcs connecting them; (2) a data graph models the flow of data between computation units (codesets) and data storages (datasets); the type of access of the codesets over the datasets is determined by directed data arcs 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. The GMB model for `Block` is shown in Fig. 5(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 "Executing the Application".

3.4 Modeling the Behavior of the Application's User Interface

The user interface of a strongly sharable 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. For example, in Fig. 5(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. An interactor models the actions, produced by the users, to which it responds (e.g., pressing a key on the keyboard, clicking or moving the mouse, etc.), and 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 communi-



(c)

```
(let ((block (make-block))
      (pos (read-from "FigPos")))
   (setf (UL-x block) (first pos)
         (Ul-y block) (second pos)
         (BR-x block) (third pos)
         (BR-y block) (fourth pos)
         (label block) (get-label (read-from "DialogBox")))
   (write-to "FigLst" (cons block (read-from "FigLst")))
```

Figure 5: The models of the block diagram editor: (a) Structural model; (b) GMB model of `Block`; (c) Interpretation for codeset `MakeFig`; (d) GMB model of `Interactor`; (e) GMB model of `DoRect`; (f) Legend for GMB models.

cation 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. 5(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 `DR`, ... `EL` to draw and erase rectangles and lines. Modules `DoRect`, `MvRect`, `DoPoly` and `MvPoly`. in Fig. 5(a), define the action sequences required to define and move rectangles and polylines. The behavior of `DoRect` is shown in Fig. 5(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.

3.5 From Models to Executable Prototypes

We explain now the linking and instantiation steps of the coSARA design methodology. 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 specification of the application. 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. A collection of methods to create windows reside in the coSARA Library and can be used by the designers. 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, as explained below.

3.6 Executing the Application

The *token machine* is an interpreter of GMB models. The meaning of a GMB model can be defined as follows: each node has an input and an output logic, and is mapped to one codeset; the input logic defines the required distribution of tokens in the node's input control arcs for it to fire, which means that the tokens are removed and the interpretation of the codeset mapped to the node is executed; the output logic defines which of the node's output control arcs receive a token when the codeset finishes its execution. 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, drag the mouse, and release the button. The interactor in Fig. 5(d) represents the drawing window. When the user pushes down the button, and then when the button is released, `GA` sends the action's window position through arc `Pos`, and places a token 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. 5(e), that really 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 places 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: `End` sends both positions through arc `Pts` and places a token on arc `ROn`; this token eventually activates codeset `DR` in the interactor, which does the drawing.

In `Block`, as shown in Fig. 5(b), the token on `Fig` fires node `N1` thereby activating codeset `MakeFig`. `MakeFig`'s interpretation, shown in Fig. 5(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. 4(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`.

3.7 Strong Sharing of Applications

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 sharable applications because of the 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 sharable. By using the software modules in the coSARA Library, which also uses the Object World, the application's operations are shared. Therefore, the application itself is strongly sharable. 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.

Each user that shares the application is then able to use it. Actions by the users are broadcast to all the sites having the application. The granularity of the multi-user interaction is determined by the software modules. It depends on which methods in the modules 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 5(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.

When multiple users share the block diagram editor, each user gets a copy of the editor and its user interface. Each user is then able to create and move the blocks and links. Each user's actions are broadcast to all sites having copies of the editor. The resulting effect of creation and move actions by multiple users is as if the actions are merged into some serial order and executed by all the shared copies of the editor.

Besides its benefits, strong sharing allows the users to inadvertently interfere with each other. In order to avoid such situations, the users are expected to coordinate their efforts by either face-to-face communication or using some communication technology (e.g., telephone and e-mail). The users can also use locking to prevent others from modify an object under construction. The Object World provides locking and an inconsistency detection mechanism.

The multi-user aspect of the block-diagram editor hinges on the Object World infrastructure. The next section explains in more detail how the Object World implements strong sharing.

4 Implementation of Strong Sharing

This section describes how the *Object World* satisfies the three main features necessary to support strong sharing as mentioned earlier; 1) shared data, 2) shared operations, and 3) coordination support. Shared data are objects for which a replica can be created and transmitted to a remote site, while shared operations are methods that operate on an object and all of its replicas. A less restrictive version of locking than database locking is provided that allows user cooperation to resolve potential deadlocks rather than aborting one of two competing transactions. This section also describes how the Object World is used to create shared operations which are stored in libraries for use by concurrent engineering applications, in particular, the block diagram editor described in the previous section.

4.1 Sharing Data

In order to share data, the Object World encodes an object as a string and then transmits the string to a remote site using TCP sockets. The remote site then decodes the string, hence creating an exact replica of the object. The Object World provides the *encode-object* and *decode-object* methods through the *sharable* class to do this. The *sharable* class is an abstract class and does not itself have any instances, but all sharable data in the coSARA system are an instance of a class which inherits from the *sharable* class. In addition to the *encode-object* and *decode-object* methods, two slot variables are inherited from the sharable class, the unique object identifier, and the name of the object. The object identifier and the name of the object are used to request a copy of the object from another site. The actual form of the encoded string is a list with the first three elements being the object class name, the object name, and the object identifier. The rest of the encoding is a list of (*slot name*, *slot value*) pairs. This type of encoding can be done dynamically in Common Lisp, since the class (and hence the slot names and slot values) of each object is known.

The Object World also supports persistent objects. The *save-object* method makes a saved image of the object. The *save-object* method applies the same encoding as is used by *encode-object* for sharable objects. In addition to encoding the object to be saved, it is also necessary to recursively save any objects pointed to by the object. An object server daemon keeps the names of all saved objects in a directory (which is also sharable). When users request an object, the object server can look up the object in the directory and retrieve it.

An example will help make this encoding clear. Consider the block diagram editor of the previous section. As part of the OREL model in figure 4b, the BLOCK class would be expanded into the following class definition:

```
(defclass BLOCK (sharable)
((label :accessor label)
 (links :accessor links)
 (br-x :accessor br-x)
 (br-y :accessor br-y)
 (ul-x :accessor ul-x)
 (ul-y :accessor ul-y)))
```

Note that the only difference between this class definition and a class definition that does not support shared data is that one of the superclasses[1] of the BLOCK class is the *sharable* class.

An object of the *BLOCK* class has six slots (besides the slots of its superclasses), one for the X and Y coordinates of the upper left (`ul-x` and `ul-y`) and bottom right corners (`br-x` and `br-y`) of the block, one for the label of the block, and one for any links to the block. The *:accessor* field of each slot provides a function to access the slot value. If B1 is an object of class BLOCK, then the function `(br-x B1)` would return the X coordinate of the bottom right corner of block B1. A *MAKE-BLOCK* function is also generated from the OREL model which creates and returns a new instance of the block class.

Consider an instance of the BLOCK class that has its upper left corner located at the point (10,10) and its bottom right corner located at the point (25,25). The encoded representation of this *BLOCK* as produced by the *encode-object* method is:

```
"(BLOCK \"block-1\" \"RA.asa-105558\" (LABEL . NIL) (LINKS . NIL)
                                        (BR-X . 25) (BR-Y . 25) (UL-X . 10) (UL-Y . 10))"
```

Many Common Lisp data types are very simple to convert to the encoded form. Numbers (e.g. 25) and strings (e.g. "RA.asa-105558") are encoded as a substring of the whole encoded string. Unfortunately, sharable objects often contain data that are not so simple. Two specific complex types of data that do not have an obvious encoding are site dependent data and pointers to other sharable objects. Site dependent data are necessary for dealing with objects in the Common Lisp X (CLX) window system interface. Objects dealing with windows have pointers to several device dependent data structures (e.g. the name of a machine, or the number of pixels in the display). Thus a window object on a site named "ra" should point to the window manager on site "ra", while a replica of that object on a site named "sol" should point to the window manager on site "sol". To accomplish this, the *decode-object* method must be specialized to fill in certain slots with local values of the site dependent data.

To handle slots whose values are pointers to sharable objects, the slot value in the encoded form is represented by a call to *read-object*, the function to access an object. Assuming the

[1]Multiple inheritance is available but not used in this example.

OREL model of the *LINK* class has already been compiled and loaded, if our original block example had a link object, then the block encoding would be:

```
"(BLOCK \"block-1\" \"RA.asa-105558\"
          (LABEL . NIL) (LINKS . ((read-object :id \"RA.asa-105561\")))"
          (UL-X . 10) (UL-Y . 10) (BR-X . 25) (BR-Y . 25))
```

The string "RA.asa-105561" and similar strings are the unique identifiers of the objects. This encoding is similar to the one before except now the block has (a list of) one link to it. This one link is represented as the function to read in the object whose unique object identifier is "RA.asa-105561". The *decode-object* function when reading in this description and setting the value for the *links* slot will evaluate the *read-object* function and set the slot value accordingly. The evaluation of the *read-object* function will return a pointer to the proper object. In the case where the object is stored locally, a pointer to the proper object is simply returned. In the case where the object is stored on another site, the *read-object* function will request the object by its unique object identifier from another site, and in the case where the object is stored in persistent storage, *read-object* will request the object from the object server daemon. Once that object is received and decoded, its pointer is returned as the value of *read-object*. Thus the new replica of the block object will be identical to the original block instance.

4.2 Sharing Operations

The other major feature of sharable objects is that special methods, which we call ***broadcast methods***, 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 as will be demonstrated in the example.

Broadcast methods are composed of two different methods, one method that is executed on the local site, and a second that is executed on the remote sites. The local site method implements the actual method code and as a side effect, broadcasts to all the remote sites a request to execute the second or remote site method. The remote site 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.

Continuing the previous BLOCK example from the block diagram editor should make shared operations clear. The *BLOCK* class has a *move* method which takes as arguments a block and the distance to move in the X and Y directions. The method then moves the position of the block, and returns the modified block.

We use the *defbroadcast* macro to define broadcast methods. Our definition for the *move* broadcast method is:

```
(defbroadcast move ((B BLOCK) DX DY)
   (setf (ul-x B) (+ (ul-x B) DX))
   (setf (ul-y B) (+ (ul-y B) DY))
   (setf (br-x B) (+ (br-x B) DX))
   (setf (br-y B) (+ (br-y B) DY))
   B)
```

The broadcast method *move* takes a BLOCK as an argument, along with a distance to move in the X direction (DX) and a distance to move in the Y direction (DY). The *move* method adds the distance moved to the X and Y coordinates of both the upper left corner and the bottom right corner, and returns the object. Note that the only difference for a broadcast method from an ordinary method is that instead of using the normal method defining function, (i.e. *defmethod*), *defbroadcast* is used. The *defbroadcast* macro is the way that sharable methods are written. When expanded, a broadcast method will consist of the local site and remote site methods as mentioned before. The local site method uses the original name (i.e. *move*) and functionality, but also broadcasts the *move* method to all remote sites. The remote site method has the original name with "nb-", for non-broadcast, prepended. The *nb-move* method provides the *move* method functionality without the additional code to broadcast the method.

When a user executes a *move* method, two things happen on his machine. First, the *move* method is broadcast to all sites. These remote sites run the version of the *move* method, *nb-move*, that does not broadcast. Then, the actual moving of the BLOCK occurs on the original local site. To understand this better, the actual macroexpanded code for the *move* method is shown below:

```
(defmethod move ((B BLOCK) DX DY)
  (if (not *sync*)                          ;;; Are we in a broadcast?
      (let ((*sync* nil))                   ;;; No - create new *sync* variable
        (declare (special *sync*))
        (setf *sync* t)                     ;;; ... and turn off future broadcasts
        (broadcast-message                  ;;; ... but broadcast the nb-move method
          (ow-update-socket *ow*)
          (concatenate 'string
                       "(propagate-update "
                       (get-dests B)
                       (write-to-string (cons 'nb-move (list (encode-all B))))
                       ")"))))
  (setf (BLOCK-ul-x B) (+ (BLOCK-ul-x B) DX))    ;;; Do the method code
  (setf (BLOCK-ul-y B) (+ (BLOCK-ul-y B) DY))
  (setf (BLOCK-br-x B) (+ (BLOCK-br-x B) DX))
  (setf (BLOCK-br-y B) (+ (BLOCK-br-y B) DY))
  B)

(defmethod nb-move ((B BLOCK) DX DY)
  (if (not *sync*)                    ;;; Are we in a broadcast?
      (let ((*sync* nil))             ;;; No - create new *sync* variable
        (declare (special *sync*))
        (setf *sync* t)))             ;;; ... and turn off broadcasting
  (setf (BLOCK-ul-x B) (+ (BLOCK-ul-x B) DX))    ;;; Do the method code
  (setf (BLOCK-ul-y B) (+ (BLOCK-ul-y B) DY))
  (setf (BLOCK-br-x B) (+ (BLOCK-br-x B) DX))
  (setf (BLOCK-br-y B) (+ (BLOCK-br-y B) DY))
  B)
```

The *nb-move* method should be clear. When the main method (*move*) executes the call to *broadcast-message*, a message is broadcast to all remote sites. A process running on each remote site receives the broadcast and checks which objects are involved in this method. If a receiver of the message finds that it stores the objects, that site executes the *nb-* or non-broadcast version of the method. In this case, the function call is to *nb-move*.

There exists a problem that could cause extra and erroneous messages to be broadcast. This is illustrated in figure 6. Consider a broadcast method *f()* which as part of its code calls another broadcast method named *g()*. Method *f()* is invoked on site 1 as shown in box 1 in the figure. Correct execution of *f()* would mean that both site 1 and site 2 execute *f()* (or *nb-f()*) and *g()* exactly once. Since *f()* is a broadcast method, the non-broadcast version of *f()*, *nb-f()*, is invoked on site 2 as indicated by the dashed arrow pointing to box 2. On site 1, *f()* calls broadcast method *g()* while on site 2, *nb-f()* calls broadcast method *g()*. Neither *g()* on site 1 nor *g()* on site 2 should broadcast as the broadcasting of *g()* would cause

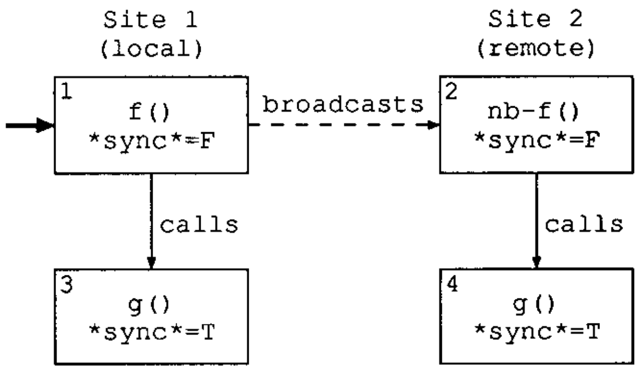


Figure 6: Preventing extra rebroadcasts

additional invocations of *nb-g()* which would be an error. This problem is solved by having a dynamically scoped variable named **sync** which is set to TRUE if a broadcast method is currently being executed. The dynamic scoping (really indefinite scope and dynamic extent - a Lisp global variable) means that the variable can be referenced anywhere as long as its binding is currently in effect. The *defbroadcast* macro provides code that tests and sets the value of **sync** when the first broadcast method is entered (*f()* in this case). Since **sync** is TRUE when *g()* is executed on both site 1 and site 2, neither site broadcasts which is the desired result.

4.3 Coordination Support

Strong data sharing is inefficient (and even counter productive) unless data consistency can be preserved. In coSARA, two measures are taken to ensure data consistency [WU91]. First, we use a dependency detection model [STEF87] to detect data inconsistency. This is not to restrict access to users, but to alarm users of possible inconsistency and to facilitate access negotiations. Second, when data access becomes highly contentious, locking can be used to prevent undesirable access. Locking is implemented with our extension to the dependency detection model. Both read and write locks are available. When locking conflicts are detected, users will be 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, email, or FAX. Such coordination is sufficient to prevent users from frequently and destructively interfering with each other. In our experience, these two measures have been sufficient in coordinating data access, detecting corrupted data, and dealing with lock contention.

5 Conclusion

We have identified *strong sharing* as an essential requirement for generating concurrent engineering applications. Strong sharing of data and applications allows engineers to work closely together. It overcomes the time, place, and work barriers that impede the concurrent engineering process. In addressing this requirement, we have developed coSARA, an environment for both realizing strong sharing and an environment for prototyping strongly sharable concurrent engineering applications. With this environment, we have built strongly sharable applications such as a block diagram editor and a graphical browsing tool, and we are currently building a strongly sharable spreadsheet. We have also have shown how heterogeneous applications can intimately share the same data.

Our system allows application builders to use a single-user tool design methodology for building multi-user tools. This makes it easier for them to build shared applications. They do not have to learn a completely new paradigm for application building. This also makes it easier to take existing applications and make them strongly sharable. Also since the applications are designed using primarily 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 the application and thereby be more acceptable to the users.

The coSARA system itself is a strongly sharable concurrent engineering application. The coSARA tools are strongly sharable since they have been built on top of the Object World infrastructure. Multiple users are able to work together on the same coSARA models. Not only does coSARA support the prototyping of strongly sharable applications, but since it is based on the SARA design tools, it also supports the design, modeling, and analysis of general concurrent hardware and software systems[ESTR86, LOR91].

Our work on strong sharing and the coSARA system has opened several additional problems and challenges for the future:

- Explicit Specification of Multi-User Interface Granularity - The granularity of the multi-user interface is currently specified at the library level by choosing whether to use broadcast or non-broadcast modules. However, this is hidden from the graphical specification models of the application. In the future, we would like to be able to specify the multi-user granularity in the graphical models where it is more apparent to the application builders. We are looking at extending the GMB to allow richer specifications of behavior to take into account better the effects of multiple users. One particular extension we would like to add is being able to specify local versus shared models. Local models would make a local copy for each user so that only one user affects that particular model. Shared models are what coSARA uses now, in which all users affect and share the same model.
- coSARA Compiler - The coSARA methodology uses an interpretation model for executing the specifications of the strongly sharable applications. This is useful for prototyping. A desirable tool would be a compiler that can take the specification models and compile them into efficient code.
- Extending Strong Sharing to the C++ Environment - Currently strong sharing is supported in the Common Lisp/CLOS environment. We are currently looking at the issues for providing strong sharing in the C++ environment which is more efficient, but less dynamic.
- Building up the coSARA Library - In the future, we hope to supplement the coSARA Library with more coSARA models and program modules. One main area in which libraries are useful is the multi-user interface. The models for the most common dialogs could be stored in a library for immediate use.

References

[AHUJ90] S. Ahuja, J. Ensor, and S. Lucco, "A Comparison of Application Sharing Mechanisms in Real-Time Desktop Conferencing Systems," from the Proceedings of the ACM/IEEE Conference on Office Information Systems (COIS), April 25-27. 1990.

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

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

[GARF89] D. Garfinkel, P. Gust, M. Lemon, and S. Lowder, "The SharedX multi-user interface user's guide, version 2.0," Research report STL- TM-89-07, Hewlett-Packard Laboratories, Palo Alto, CA, March 1989.

[GREE91] S. Greenberg, M. Roseman, D. Webster, and R. Bohnet, "Issues and Experiences Designing and Implementing Two Group Drawing Tools," University of Calgary. Dept. of Computer Science Research Report No. 91/438/22, July 1991.

[GREE90] S. Greenberg, "Sharing Views and Interactions with Single- User Applications," from the Proceedings of the ACM/IEEE Conference on Office Information Systems (COIS), April 25-27, 1990.

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

[LOR91] K. E. Lor and D. Berry, "Automatic Synthesis of SARA Design Models from System Requirements", IEEE Transactions on Software Engineering, Vol. 17. No. 12, Dec.1991, pp. 1229-1240.

[MUJI91] S. Mujica, "A Computer-based Environment for Collaborative Design," UCLA PhD thesis in Computer Science, 1991.

[SRIN92] K. Srinivas, R. Reddy, A. Babadi, S. Kamana, V. Kumar, and Z. Dai; "MONET: A Multi-media System for Conferencing and Application Sharing in Distributed Systems," CERC Technical Report - Research Note, CERC-TR-RN-91-009, Feb. 1992.

[STEF87] Mark Stefik, Gregg Foster, Daniel Bobrow, Kenneth Kahn, Stan Lanning, and Lucy 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.