coSARA: A Computer-based Environment for Collaborative Design, Analysis and Evaluation of Computer Systems†

Sergio Mujica
Departamento de Ingeniería Informática, Universidad de Santiago de Chile
Casilla 10233, Santiago, Chile

Yadran Eterovic
Departamento de Ciencia de la Computación, Pontificia Universidad Católica de Chile
Vicuña MacKenna 4860, Santiago, Chile

Steven Berson, Gerald Estrin, Poman Leung, Ivan Tou, Elsie Wu
Computer Science Department, University of California, Los Angeles, CA 90024

ABSTRACT

This paper describes the design and implementation of coSARA a computer-based collaborative environment for the design, analysis and evaluation of computer systems. The general goal of this work is to create a platform that will encourage fundamental and systematic experimentation towards understanding of techniques to realize collaborative systems and methods for collaborative design of computer systems. In this paper we discuss the environment in which coSARA is assummed to operate; we focus our attention on the the object system implementation; we also provide a review of coSARA's user interface and of its tool integration method.

Keywords: collaborative, CSCW, groupware, design methodology, object-oriented, distributed, integrated, user-interface.

1. Introduction

This paper describes the design and implementation of coSARA[11], a computer-based collaborative environment for the design, analysis and evaluation of computer systems. CoSARA has been conceived to be used by a group of engineers working face-to-face, each at a workstation with graphics capabilities, using the methods and tools of SARA[6] to design and realize complex computer systems. coSARA also includes tools and methods to specify and build other computer-based collaborative systems, different from coSARA. *The general, long-term goal of this work is to create a platform that will encourage fundamental and systematic experimentation towards understanding of techniques to realize collaborative systems and methods for collaborative design of computer systems.*

† This work has been partially supported by Hughes Aircraft Co., TRW, NCR Corporation, UNISYS, AT&T, Interactive Development Environments, SUN Microsystems, the State of California MICRO Program, Perceptronics Inc. and FONDECYT Project No. 0437-92.

There are severe technical problems to create useful computer-based systems in support of group work that are not satisfactorily solved today. Some of these problems are concurrent access to data shared by group members, design of effective user interface devices to convey information about group context, methods and tools to compile and examine group work history, and modeling of group processes. A discussion of these problems can be found in [11].

This research has contributed mainly to the technology for the realization of computer-based environments to support collaborative work. Two main contributions are:

- *A distributed object-oriented sdata management ystem* It includes a graphic language for modeling data using concepts of objects and relations, a library of functions for building programs that manipulate objects, support for distributed, collaborative sharing of objects, and persistent storage of objects.

- *A tool modeling and integration method.* This method includes a tool model for environment extendibility and collaborative tool sharing that allows incremental, partial and full integration of tools; enables tools to operate on collaboratively shared objects; and supports the incremental extension of the environment. It also includes a user interface model that allows sharing of interaction mechanisms, provides support for modeling behavioral response to multi-user actions, and enables early testing and analysis of user interfaces.

In this paper we discuss the environment in which coSARA is assummed to operate establishing the area of CSCW [8] that it covers; we focus our attention on the the object system implementation. We also provide a brief review of coSARA's user interface and the tool integration method. We do not present here the graphic language for object oriented modeling of data, details of the user interface model and implementation and details of the tool integration procedure. A full discussion of topics not covered by this paper can be found in [11].

2. Group work

The basic problems of group work are related to communication among group members and coordination of their joint activities. Communication and coordination tasks are required for the group work to progress towards mutually agreed upon goals. These tasks involve complex problems in management of shared information. It is shared information what makes collaboration feasible, expeditious and effective.

Several factors have an impact in the quality of communication and coordination in group work: existence of a common view of the way in which the group works together, a common language for exchange of information among group members, usage of unstructured information (such as natural language, video images and sound), methods and tools for systematic communication and coordination, methods and tools to compile and trace the history of group work.

coSARA approaches these problems offering a group interface which conveys information common to all group members, an object-oriented data model common to all group members and specification of protocols to share objects at the user interface level. These features contribute to create a common language, establish the way in which the group works together and systematize communication and coordination. coSARA lacks methods and tools to manage history of the group process and capabilities to work with unstructured information.

We can classify collaboration according to location and time (Table 1). Asynchronous and remote collaboration are discussed in detail in [11]. In synchronous, face-to-face situations, all group members work in the same room, each one at a workstation, using a common large

screen to focus group attention (images displayed in the large screen may not be same as those displayed in the individual workstations). Group members can communicate by voice and body language. These forms of communication are highly unstructured and difficult to handle by computer. coSARA is designed to be used in synchronous, face-to-face collaboration.

	face-to-face	remote
synchronous	• inmediate response •social interaction • large bandwidth	• inmediate response • computer-based interaction • small bandwidth
asynchronous	•deferred response •computer-based interaction •any bandwidth	

Table 1: Classification of collaborative activities

3. The Operation Environment of coSARA

CoSARA (collaborative System ARchitect Apprentice) was developed at the UCLA Collaborative Design Laboratory, using the hardware and software system shown in Figure 1. CoSARA is written in CommonLisp and it runs under UNIX®. Its size is about 30,000 lines of code. We wrote a few pieces of code in C language to interface TCP/IP system calls. C code amounts to about 100 lines. CoSARA was designed as a classic object-oriented system that includes classes, metaclasses, multiple inheritance, object composition and polymorphic messages [7,3,9]. CoSARA's object system, that we call OREL (Objects and RELations) was built on top of CLOS (Common Lisp Object System) using PCL, a public domain implementation of CLOS created by XEROX PARC [2]. The graphic interface was built using CLUE (CommonLisp User-interface Environment) which is an X Window System toolkit for CommonLisp.

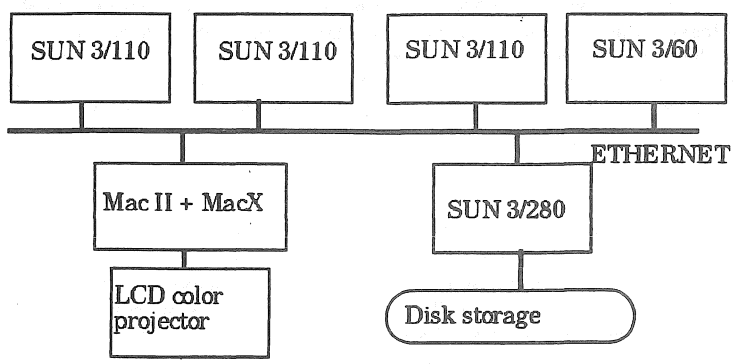

Figure 1: Hardware environment

coSARA is used by a team of engineers for design of complex computer systems in a laboratory shown in Figure 2. There is a large conference table surrounded by workstations. The engineers work looking away from the table to see their workstations. When it becomes necessary that the group discusses a topic, the engineers turn around to face each other at the table. Group

focus on shared information is conveyed by a large screen located at one end of the room. It is, however, desirable that engineers face each other at all times. The massive displays of the workstations prevent such a setup. We expect that technological advances, such as LCD color displays, will help to improve this situation providing high resolution, small and flat screens.

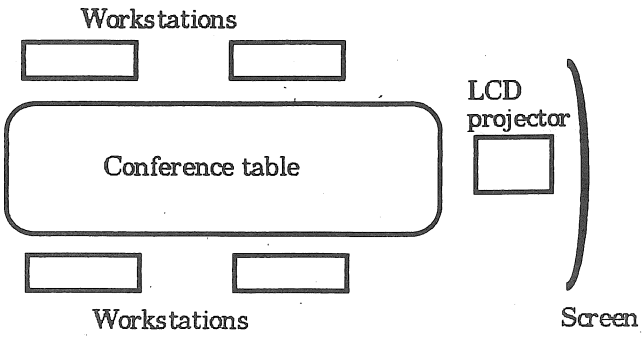

Figure 2: Laboratory layout

4. The design of coSARA

A structural model [6] that shows a modular decomposition of coSARA is shown in Figure 3. The system is first decomposed in two main modules, called ENVIRONMENT and coSARA respectively. ENVIRONMENT encapsulates the assumptions under which coSARA is expected to operate. CoSARA is partitioned into three top-level modules: group-interface, object-world and system-manager.

The behavior of environment is as follows. A user (who is not necessarily a privileged user) initializes the system preparing it for a meeting. Thereafter other group members join the meeting one by one, until a maximum of K (a small integer, usually between 5 and 10) are working jointly. group members operate coSARA using mice, keyboards and graphic displays. A group member can leave a meeting and join it again at a later time. CoSARA creates a new session for each user that joins the meeting, presenting an up to date view of shared objects. All other users are notified of the newcomer. Similarly when a user leaves a meeting all other users are notified. Each group member uses private and shared tools to create modify and examine shared objects in the object-world.

Collaborating group members discuss about shared objects and evaluate modifications made to shared objects, looking at identical images of the objects on their own screens. Figure 4 is an example of how coSARA enables visualization of shared objects. In the example, object-1 is shared by three users who are working at workstation-1, workstation-2 and workstation-3 respectively. Object-1 is drawn in two windows, window-1 and window-2. Window-1 is displayed only on workstation -1, window-2 is displayed on the three workstations. This shows how in user interface elements are also shared objects and therefore several users can share the same window (window-2 is shared by three users in the example). The user at workstation-1 has a private view of object-1 and all three users have a shared view of object-1 on widow-2 .

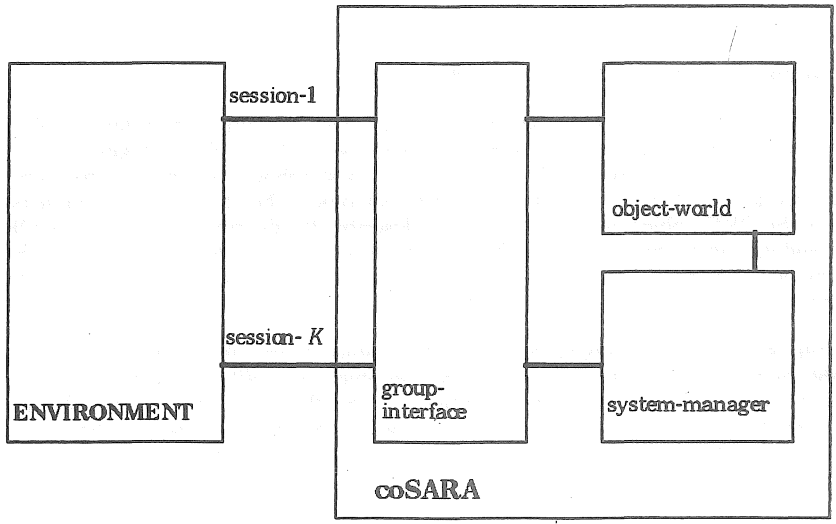

Figure 3: Structural model of coSARA

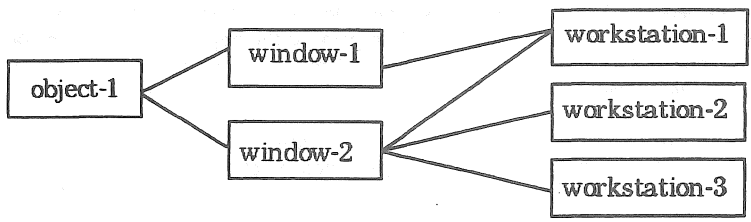

Figura 4: Visualización de objetos compartidos

The group-interface module is a multi-user interface that provides functionality necessary to support group activities. It operates according to protocols for sharing objects at the user interface level. The user interface of a tool specifies a protocol that rules how that tool lets users manipulate objects. The user interface behavior, including collaboration protocols, is specified using SARA models [6].

The object-world module is a realization of the OREL object system and provides functionality for sharing data objects and their operations (note that a tool is an object). Shared objects are stored using a method that we call replication on demand. There are multiple copies of each object, one in each workstation that has read it. This method guarantees that each workstation will store only that subset of objets it is using. Operations done on an object are sent to all workstation that have copies of the object to maintain consistency of data.

The system-manager module allows users to join and leave meetings in progress and the incremental integration of tools into the coSARA environment.

In the following three sections we give a brief description of how coSARA has been designed and implemented. We focus mainly on the object system because we believe it is a fundamental

component of computer-based environments for collaboration. A full discussion of coSARA's design and implementation can be found in [11].

5. Implementation of the object-world module

OREL objects follow a classical model that includes classes, multiple inheritance, polymorphic messages and object composition. Distributed operation of OREL objects takes place in the OREL *object world.* We envision the object world as a storehouse of objects, where users can find and operate on objects in coordination with each other. The object-world module is a realization of the OREL object world. This section describes the distributed operation of OREL objects and its implementation in the object-world module. A full description of the OREL data modeling language and a discussion of design issues concerning coSARA's object system can be found in [11].

Three rules summarize the properties of the OREL object world: the existence of all objects is *knowable* by all collaborating users at all times; an object is *modifiable by only one user at a time,* it can be used otherwise, but not modified by more than one user at a time; changes made to an object that is shared by two or more users, can be seen by other users as they are made, *unless the user who is modifying the object has chosen to restrict such access until changes have been completed.*

Coordination among users is achieved by using long, interactive transactions, controlled by users. To perform some task that will affect shared objects, a user must start a transaction. When the task is finished, or when the user decides that a break is convenient, the transaction is committed and ended. At any point the user may decide to abort the transaction, restoring the objects that were modified to the state they had prior to the beginning of the transaction. A user may have several transactions in existence at any time, but only one of them can be active. The intent is to *provide a way in which the user can organize a set of on-going tasks.*

To execute operations a user acquires hierarchically implied locks on composite objects[1] at any time and in any arbitrary sequence. All locks are released atomically when the transaction is committed to prevent cascading roll-backs, that could be very costly because transactions are long . The locking scheme is a non-strict 2-phase locking protocol[1], that may lead to deadlocks. We rely on the fact that such long, interactive transactions, controlled by human users, allow resolution of deadlocks by social negotiation among users. The object world provides all necessary information to carry on such negotiations.

For example consider the following scenario: designer 1 needs to have exclusive access to modules A and B to perform some task. Designer 1 successfully obtains a lock on module A, but locking module B fails and designer 1 is informed that designer 2 already has a lock on module B. Designer 1 then asks designer 2 for how long she plans to use module B. Designer 2 answers that she will use module B only for a few minutes as soon as she gets a lock on module A. At this time designer 1 realizes that a deadlock exists and informs designer 2 that he already has a lock on A. Designer 1 then agrees to release his lock on A and wait until designer 2 is done. The interaction illustrated by this example is possible because the object world provides information such that designer 1 could identify designer 2 as the current owner of a lock on module B. The interaction was carried out easily because designer 1 and designer 2 were working face-to-face and coSARA provided information about who was using each object.

---

[1] This means that if an object x is composed of objects a and b, then locking x locks a and b as well. The algorithm used to lock composite objects is the simple tree-warning algorithm described by Ullman [1987 (Section 9.7, page 504)].

Work done on any object should be visible to all users at any time, unless whoever is changing the state of the object chooses to hide changes until they are complete. We define a *dirty-read* lock that enables a user to read an object while it is being modified by another user, seeing thereafter the object's changing, intermediate and uncommitted states in real-time. A dirty-read lock also warns the user that the state of the object is unreliable because the user who is making changes may abort his transaction, because delays in the network may not maintain strictly the real-time requirement, or because failures may occur in the communications system. We mean to warn the user who has a dirty-read lock because it is possible that the user is misled by the "dirty" state of objects displayed and proceeds to make decisions based on such an unreliable state, destroying the strictness of the transaction model. Such user mistakes could lead to manual execution of cascading roll-backs. Dirty-reading is meant to be used for discussion purposes and joint decision processes. It is not meant to be used in support of individual decisions. Other types of locks are more conventional: write and read locks for normal operation and write-warning locks in support of the tree-warning algorithm for locking composite objects. Locks are not enforced by the system. The observance of locks is left to the tool implementor. In this way we avoid checking for locks unnecessarily.

There are several possible approaches to build an implementation of the object world, some of which have been used successfully in other systems such as: centralized storage, proxy objects [5], fully replicated objects [13], and finally replication of objects on demand, which is the method that we have chosen for our implementation. In this method, each machine has only a copy of those objects that have been requested locally. All copies of an object which reside in some subset of the machines are maintained up to date.

Broadcast methods are used to operate on shared OREL objects. Let us consider the case when an object is shared by more than one user. We will say that each user has a *replica* of the shared object, and our objective is to keep all replicas identical to each other at all times. The most straightforward way to achieve this sort of sharing objects, is to make the smallest possible change to the object's state be applied to all existing replicas in the system atomically and exclusive of any other operation that may change the object.

There are two problems with the approach just described. Changes may be distracting when they propagate to each user's replica in fine-grained fragments and transactions may require excessively large amounts of CPU cycles and network bandwidth. Each individual change of the state of an object would require at least a 2-phase commit operation at a cost of $3n$ messages, when there are n processes participating in the 2-phase commit and no failures occur (Bernstein, Goodman & Hadzilacos [1987], page 236). This cost is quadratic when there are failures.

To alleviate these problems we have adopted broadcast methods, that operate on all the existing replicas of an object simultaneously but do not include any concurrency control mechanism. Using broadcast methods we give control to the tool implementor to determine the proper grain size of state changes that a collaborating user will see in a shared object. By choosing whether a method is broadcast or not, the tool implementor can control whether the method is executed at a single site or at several sites.

For example, consider the class of shared rectangles. One operation that can be done on rectangles is to reshape them, The user is given two operations for reshaping a rectangle:

try-shape: changes the shape of the rectangle only temporarily and displays the rectangle in its temporary shape. When the another operation on the rectangle is performed, its original shape is used and the temporary one is discarded.

set-shape: takes no argument and replaces the rectangle shape with a temporary shape set by **try-shape.**

Now the tool designer can choose what will be the grain size of changes made to a rectangle that will be seen by all users who have a replica of the rectangle. If both **try-shape** and **set-shape** are made broadcast methods, every shape that a user tries will be seen by others. However, it is possible that the tool designer determines that collaborating users do not need to see intermediate shapes. Then, the tool designer would make **try-shape** a non-broadcast method and **set-shape** a broadcast method. If the process of finding the right shape for a rectangle is intended to be done as a group activity, **try-shape** would have to be a broadcast method.

Sharing objects poses a problem of concurrency control. It is necessary that one user does not destroy other users' work. In the previous example, suppose that **set-shape** is a broadcast method and that **try-shape** is not a broadcast method. If two users working at different workstations issue a **set-shape** at the same time, it is not possible to predict the results. If both executions of **set-shape** are interleaved, it could happen that the resulting shape is composed by the height set by the first user and the width and position set by the second user. Furthermore, one could ask, what is the meaning of two users trying different shapes for the same rectangle? The answer to this question is methodological. It depends on the collaboration process that the tools are intended to support.

In summary, the user must manage transactions, lock objects and negotiate solutions to conflicts using the information provided by the object world. The tool implementor must define the granularity with which remote updating of shared objects is done using broadcast methods and provide functionality to handle conditions raised by locking failures.

The implementation of shared objects, replicated on demand, has a central server and zero or more client sessions. The server provides three management services: locking, persistent object storage and meeting configuration. A client consists of a main Lisp process plus several processes that provide network services needed by the client to send and receive shared objects, and to send and receive requests for operation on shared objects. The sending and receiving of objects between clients is done synchronously. Each time a method is performed on a shared object, a request to apply the method is broadcast to all the other participants using an asynchronous broadcast channel.

The operation of the system requires that a server be running. To join an existing design session, a client tells the server that it is joining and requests from the server a site from which it can get a copy of the current data structures. Clients must join sessions one at a time. This guarantees that each client has an up to date copy of system-wide data structures, therefore the server can choose any arbitrary site to send session data to the newly joined client. If this is the first client to join (and therefore no other site can supply the data structures), the client data structures are initialized to be empty. When a client leaves a session, the server is notified so that it will not refer data structure requests of any newly joining sessions to the departing client. The other clients are also notified so that they can update their tables to prevent sending of requests for objects to the departed client.

We define the *storable* class of objects which is a superclass of all classes of shared objects defined in the object-world. Storable objects[2]can be encoded into an ASCII representation.

---

[2]Here and in the rest of this section, when we use the term storable objects, we will mean any object whose class is a subclass of storable.

This representation is the form in which objects are sent between sites and to the database. Each site can then reconstruct the object from its encoded ASCII representation. There are three levels of storage:

Local cache: contains a set of replicas of objects that are stored in the local workstation. It is implemented using a hash table that provides access to an object using either a unique identifier or a combination of object class and symbolic name, which is not necessarily unique.

Global cache: is the union of all the local caches. Each workstation has a global directory listing all the objects that are stored in some local cache. Each entry of the global directory has a list of the machines that store replicas of the object.

Secondary storage: is resident on disk devices and stores objects permanently. This store is managed by a central server and is accessed through a directory that is maintained by the central server.

There is functionality to read, save, delete, rename and copy objets. For example, the function `read-object` takes as arguments either the unique identifier of an object or its name and type. It will search the local cache, the global cache and the central secondary storage in that order. If the object is found the function returns it otherwise it returns `nil`.

There are two functions to support transmission of objects, through the network, between different machines: `encode-object` which produces an ASCII encoding of an object and `construct-object` that takes as argument a string that contains the encoding of an OREL object and produces an instance of the object.

Data such as numbers, symbols, and strings can be encoded trivially. However, storable objects often contain data that cannot be encoded trivially. Two specific types of data that we have had to deal with are pointers to other storable objects, and site-specific data of objects in the Lisp/X Window System interface. Objects dealing with windows have pointers to several device dependent data structures.

Pointers to storable objects are encoded as a call to `read-object`. For example the following string can be the result of applying `encode-object` to a binary tree, i.e. it is the encoded form of a binary tree. The string `"RA.asa-105558"` is the unique identifier of the root, `"RA.asa-105560"` is the unique identifier of the left subtree and `"RA.asa-105561"` is the unique identifier of the right subtree.

```
"(SEARCH-TREE NIL
      "RA.asa-105558"
      (KEY . 15 )
      (LEFT . (read-object :id "RA.asa-105560"))
      (RIGHT . (read-object :id "RA.asa-105561")))"
```

The string "`RA.asa-105558`" and similar ones are the unique identifiers of the objects. The left and right subtrees have a value which is a function to read in the correct subtrees. The `construct-object` function, when reading in this description and filling in the value for the `left` and `right` slots, will evaluate the calls to `read-object`. This will occur recursively, so if the left and right subtrees have their own subtrees, these will all be read in all the way down to the leaves.

Broadcast methods are implemented in three parts. The first and second parts are run on the local machine, while the third is executed on remote machines only. Broadcast methods are implemented by a Lisp macro called `defbroadcast` that generates three different methods, representing the following three parts: code that implements that method's functionality; an :after method that broadcasts a request to invoke the method over the network; a plain version of the method without an :after method which is the version that other sites will run instead of the original method (if they attempt to run the original method, the system would be caught in an infinite broadcast loop).

We present an example of a class counter object to clarify this concept. Counter objects have one slot which contains their current value. The counter class has several methods defined on it including value, increment, decrement, and reset. Value returns the current counter value. Increment (decrement) adds (subtracts) one to the current counter value. Reset sets the counter value to zero. Increment, decrement and reset all return the counter object. We define the counter class, using CommonLisp and CLOS as follows:

```
(defclass counter (storable)
 ((value :accessor value :initform 0)))
```

In this definition, the class counter is a subclass of storable. It has one slot named value which has an initial value of zero and whose value can be accessed by `value`. We can create an instance of this class with the following instruction:

```
(setf count-1 (make-instance 'counter))
```

We use the `defbroadcast` macro to define broadcast methods. Our definition for the `increment` broadcast method:

```
(defbroadcast increment ((c counter))
     (setf (value c) (+ 1 (value c)))
     c)
```

The broadcast method `increment` takes a counter as an argument, adds one to the counter's value, and returns the counter object. When the `increment` method is invoked, two things happen. First, the actual incrementing of the counter occurs. Then the `:after` method is executed broadcasting this method to other sites. Remote sites execute a version of this method that does not broadcast.

There is a problem that can cause extra messages to be broadcast erroneously. This happens if one broadcast method calls another broadcast method. On receiving the first broadcast, the remote sites will execute the no-broadcast versions of the same method, which calls a broadcast method. That broadcast method has to be smart enough not to broadcast. But the local site must also be smart enough not to broadcast the second method or else all the remote sites will do the second operation twice. This problem is solved having a dynamically scoped variable (named `*sync*`) which is set to true if a broadcast method is currently being executed and prevents further broadcasting.

Another problem is that a single method may apply to several objects. Some of these objects (but not others) may exist on other sites. If a method of this nature is executed, remote sites may apply the (no-broadcast) method to objects that do not exist on that site. There is a `send-args` function that makes sure all objects mentioned in the method's argument list are stored at the sites where the broadcast method will be executed.

6. The tool model and the group interface

CoSARA supports a method for modeling and integration of tools into integrated extensible environments. The user interface model and the tool integration procedure include the case of partial integration of foreign, pre-existing tools. We define an integrated, extensible environment as a system that provides a common user interface and a common data management system for a set of tools, such that a new tool can be added without rebuilding any other part of the system.

When we build a new tool from scratch, we build it as an OREL object and we model its user interface using SARA methods and tools. This process results in a fully integrated tool that makes use of all the capabilities of the comon user interface and the common data management system. The method for modeling and integration of foreign pre-existing tools requires the construction of modules to bridge user interface interaction between the coSARA multi-user interface and the foreign tools as well as the construction of modules to bridge interaction between the foreign tools and the object world. How much of the common user interface and the common data management systems a foreign tool shares with other tools is determined by the corresponding bridges. Given the formalism of OREL we envision providing computer support in the construction of such bridge modules. However, in this current work we have not gone so far as to provide algorithms that would allow such automatic support.

Tools are passive in the sense that they do not take control of the environment as would happen in Unix when a program is started (if you run vi, you talk to vi!) Tools are not programs in this sense. Tools are objects, and provide functionality for the invocation of methods that act on data objects. In our view, tools are artifacts that are plugged to the user interface in order to define specific ways in which the system is going to react to gestures made by designers. A tool provides access to a set of one or more related methods to change or to analyze design models. For example, the GMB Simulator [6] is a tool that provides methods to simulate the behavior of a model, do performance analysis and do interactive debugging. The methods that are accessed by a tool are written in CommonLisp, operate on data defined in the collaborative design environment data model and may invoke other similar methods.

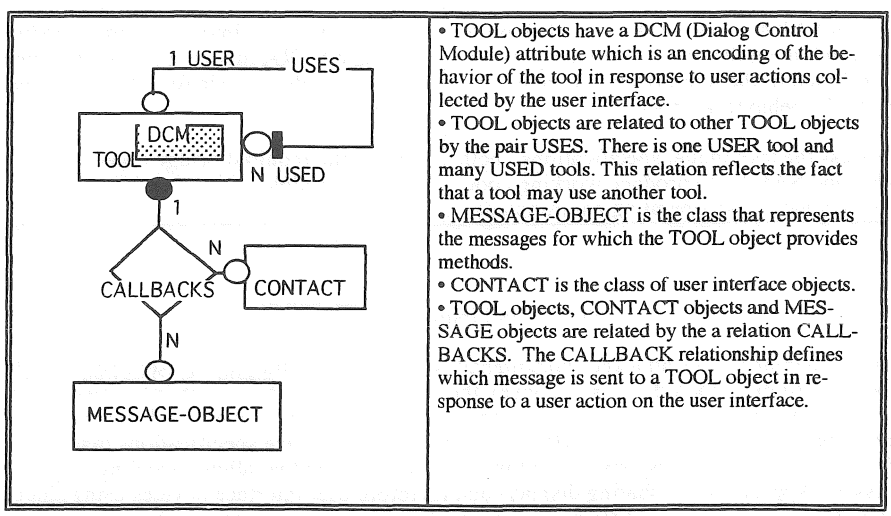

Figure 5 Tool integration procedures

Figure 5 shows a data model of tools, expressed in terms of the OREL graphic language for data modeling. TOOL is the class that encapsulates the properties common to all classes of tools. Specific tools are defined as subclasses of TOOL and therefore inherit all its properties. A particular tool class may have its own data attributes and participate in other relations as we will show in an example later in this section.

To model a particular tool we specialize the class TOOL, defining subclasses of TOOL. Subclasses of TOOL may then participate in relations and have any other arbitrary property. The data model that defines the class of a tool must be integrated into the coSARA data model. This step is done by editing the coSARA data model using the OREL graphic editor.

Table 2 contains the tool integration procedures. The condition that any foreign tool has to meet in order to be partially integrable into the coSARA environment is that the above mentioned bridge modules be constructible.These procedures allow *incremental extension of the environment.* We have shown how the user interface model and construction procedure permits the incremental integration of individual tools into the common user interface. We have shown by way of the procedures stated above that a tool can be incrementally integrated into the coSARA environment provided that we only add to the common data model. If the existing data model is altered in ways other than additions, it may be necessary to rebuild tools and reinitialize the object world database where persistent objects are stored.

Full integration of native tools	Partial integration of foreign tools
1. Modify the common OREL data model as needed by the tool, adding new definitions for the model of the tool object as well as any subsidiary objects that are needed.	1. Modify the common OREL datamodel as needed by the tool, adding new definitions for the model of the tool object as well as any subsidiary objects that are needed.
2. Build a SARA model of the tool's behavior and implement the tool functionality which is included in the interpretation domain of the tool's behavioral model.	2. Specify the user interface bridge modules using a SARA model. Process this specification to obtain code for dialog management.
3. Process the tool model specifications, both OREL and SARA, to produce working code to be included in an operating environment.	3. Build the required bridge object world bridge modules.

Table 2: Tool integration procedures

Severalr esearchers have attacked fundamental issues in supporting multi-user operation. Focus has been on finding good devices to convey the multi-user activity to each individual, how to deal with public and private information, and how to share display devices. However, little or no attention has been paid to formal methods for multi-user interface specifications that allow modeling of behavior in response to multiple user actions and that allow modeling of group processes. Approaches to sharing displays and therefore user interface devices using shared window servers are also unsatisfactory. Use of shared window servers for this purpose results in a relatively easy way to port a single user tool into a collaborative environment but it does not support what we believe is an essential characteristic of collaborative systems: *make each*

individual as aware of other users' actions as possible, as opposed to the aim of conventional systems that intend to give each individual the illusion of being the single user of the system.

Our user interface formalism is based on SARA and OREL. SARA allows us to model concurrent systems and provides tools and methods for early simulation and analysis. By modeling a group of designers as a concurrent system we can incorporate group process models in the user interface. Use of OREL for modeling and construction of user interface objects results in true semantic sharing of user interface objects such as windows, buttons and menus.

The group-interface (Figure 3) is composed of one session per designer working in the environment. The individual sessions that compose the group-interface module do not communicate directly with each other. The different sessions communicate and coordinate by sharing objects that are stored in the object-world module. Our user interface model is based on event processing, operates under external control, has a high bandwidth of communication with the application tools and uses an object oriented programming style.

The process to build and extend the user interface should translate the formal specification of a tool's user interface into working code. Figure 6 displays a dataflow model of the process to build and extend tools that is implemented in coSARA.

Contacts are areas of the screen sensitive to user actions. For example menus, buttons and graphic windows are contacts. Contacts are specified using SARA models. A Contact model can be graphically created and edited using SARA editors, possibly using other predefined contact models stored in the SARA building block library by a separate process. A Tool model can be defined similarly.

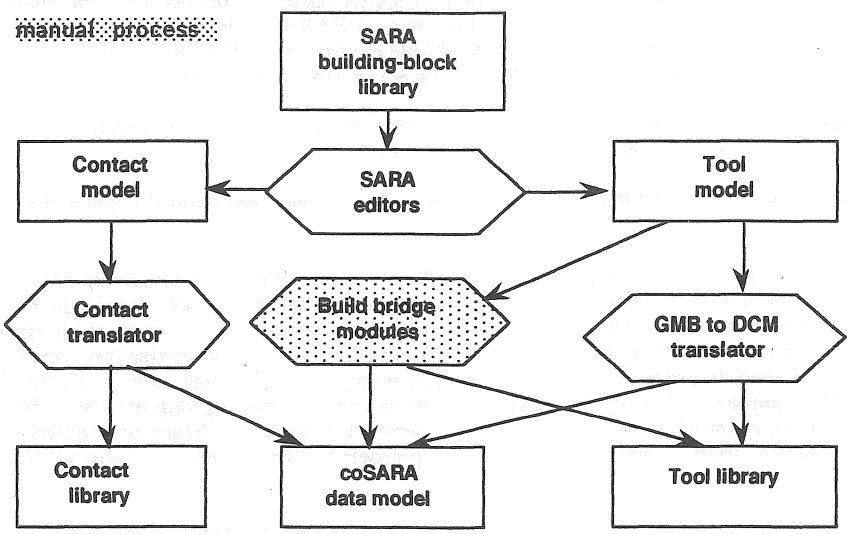

Figure 6: Dataflow model of the process to build and extend tools

The Contact translator processor converts a Contact model into data structures and executable code that implements the contact behavior. It also updates the coSARA data model. The executable code is stored in the Contact library. Similarly, the GMB to DCM translator converts a Tool model into data structures and executable code, to be used by the run time dialog manager, and updates the coSARA data model. The executable code is stored in the Tool library. The Contact library and the Tool library are called collectively *user interface libraries.*

The system is initialized with a set of *basic contacts* (screen buttons, menus, text collectors, dialog boxes, graphers and canvases) and a set of *basic user interface tools,* stored in the different libraries. The contacts implement simple techniques for interaction between the system and the designers, and can be used to compose more complex contacts. The basic user interface tools provide functionality to zoom and pan in a graphic window, select objects pointing to their graphic representation and to display and edit graphic representations of objects.

Basic contacts and basic user interface tools constitute a toolkit for our user interface development system. The dialog manager is provided by SARA in the form of a token machine interpreter, which is part of the SARA GMB simulation tool. Analysis components of SARA (simulation, static control flow analysis and performance analysis tools) provide a set of analysis tools for our user interface development system.

8. Conclusions and future developments

We have identified management of shared objects as a fundamental technology for the realization of computer-based environments that support collaborative work. The design and construction of more powerful and flexible concurrency control mechanisms, will enable us to produce better tools for collaboration, in all possible combinations of time, space and group work style.

As a response to this observation we have built coSARA based on functionality that enables groups to share data and tools in a collaborative way. coSARA contains tools and libraries to specify, analyze and generate other computer-based environments that support collaborative work. From this point of view we say that coSARA is a meta-environment.

The model of the way in which a group uses interface objects, necessary to specify a tool, is a collaboration protocol that represents the group process.

Our research work on coSARA has uncovered a number of new problems that are a challenge for future work:

- The creation of intelligent tools to record and examine the history of group processes, more precisely the acts in which a group commits a decision. Tools to manage history should meet several requirements. We can mention as examples of such requirements that history recording should be as unobtrusive as possible—otherwise the overhead work load on group members may be large enough to discourage them. It is clear that a large amount of history information can be recorded during a group process. It becomes necessary therefore to provide tools to examine history easily. Some researchers have produced tools to manage history of group processes that are not satisfactory yet [4,10,15].

- Creation of concurrency control methods based on formal specifications of collaboration protocols. These methods should allow sharing of data with a granularity that depends on the state of the group process, as modeled by a collaboration protocol. The rules for concurrent access to data that are applied at any time, depend also on the state of the of the group process.

- Creation of a library of building blocks for coSARA, aimed at the modeling of computer-based environments for collaborative work. For example, we need to create building blocks for modeling group interfaces (such as shared dialog boxes) and collaboration protocols.

- It is necessary to study methods to evaluate the impact of introducing computer-based systems in group work.

- We must extend OREL, coSARA's distributed object management system, to operate on wide-area networks since it only works on local area networks.

- Creation of tools for collaborative design of computer systems when groups work remotely and asynchronously.

9. References

1. P. A. Bernstein, V. Hadzilacos & N. Goodman [1987], *Concurrency Control and Recovery in Database Systems*, Addisson-Wesley Publishing Company, Reading Massachusetts.

2. D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G.Kiczales & D. A. Moon [February 1990], "Common Lisp Object System Specification", Limited Distribution Draft.

3. D.G. Bobrow & M. Stefik [January 1986], "Object-Oriented Programming: Themes and Variations", *The AI Magazine*.

4. J. Conklin & M. L. Begeman [1988], "gIBIS: A Hypertext Tool for Exploratory Policy Discussion", *ACM Transactions on Office Information Systems* 6 (4), 303-331.

5. D. Decouchant [1990], "A Distributed Object Manager for the Smalltalk-80 System", in *Object-Oriented Concepts, Databases, and Applications*, Won Kim & Frederick H. Lochovsky, eds., ACM Press 1990, 487-520.

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

7. A. Goldberg & D. Robson [1984], *Smalltalk-80: the Language and its Implementation*, Addisson-Wesley, Reading, Massachusetts.

8. I. Greif [1988] *Computer-Supported Cooperative Work:: A Book of Readings*, Morgan Kaufman Publishers, San Mateo, CA.

9. S. Keene [1989], *Object-Oriented Programming in Common Lisp*, Addison-Wesley, Reading, Ma..

10. J. Lee [1990], "SIBYL: A Tool for Managing Group Decision Rationale", Proceedings of the Conference on Computer-Supported Cooperative Work, October 7-10, Los Angeles, California, ACM Press, 79-92.

11. S. Mujica [1991], *A Computer-Based Environment for collaborative design*, Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles.

12. M. Stefik, D. Bobrow, G. Foster, S. Lanning & D. Tatar [April 1987], "WYSIWIS Revised: Early Experiences with Multiuser Interfaces", *ACM Transactions on Office Information Systems* 5, 147-167.

13. M. Stefik, G. Foster, D. Bobrow, K. Kahn, S. Lanning & L. Suchman [January 1987], "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings", *Communications of the ACM* 30, 32-47.

14. J. D. Ullman [1988], *Database and Knowledge-Base Systems* (Volume I), Computer Science Press, Rockville, Maryland.

15. K. C. Yakemovic & J. Conklin [1990], "Report on a Development Project Use of an Issue-Based Information System", Proceedings of the Conference on Computer-Supported Cooperative Work, 105-118, October 7-10, Los Angeles, ACM Press.