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

A COMPUTER-BASED ENVIRONMENT FOR COLLABORATIVE DESIGN

Sergio Tadeo Mujica

December 1990
CSD-900050

UNIVERSITY OF CALIFORNIA

Los Angeles

A Computer-based Environment for Collaborative Design

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Sergio Tadeo Mujica

1991

The dissertation of Sergio Tadeo Mujica is approved.

Marvin Adelson

Marvin Adelson

Bagrodia

Rajive L. Bagrodia

Daniel M. Berry

Charles Eastman

Michel A. Melkanoff

Michel A. Melkanoff

Gerald Est

Gerald Estrin, Committee Chair

University of California, Los Angeles

To Estela,
Macarena and
Sergio Alfonso

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I wish to thank the many people who have contributed to the successful course of my studies.

In particular I wish to express my gratitude to Professor Gerald Estrin, chair of my dissertation committee, who has been a friend, a teacher and much more during these years. He always gave me his support when I needed it. His guidance helped me improve myself in many ways.

The members of my committee Professors Marvin Adelson, Rajive Bagrodia, Daniel Berry, Charles Eastman and Michel Melkanoff provided a very rich interaction and helped me to achieve a clear understanding of my research.

The members of the SARA group enriched my work in long hours of discussion and made great contributions to my research. All of them have been great friends and I thank all of them for their encouragement.

The SARA group implemented a prototype system that exhibits the results of my research. Steve Berson worked on the object world, the OREL translator and the grapher; Katie Chong did the SARA to DCM translator and participated in the GMB editor; Maneesh Dahgat built the token machine interpreter; Armando Delgado contributed the

control flow analyzer; Yadran Eterovic built a library of contacts, the GMB editor, the zoom tool and the zoom tool example of Chapter 5; Silvio Grilo joined forces with Delgado in building the control flow analyzer; Carmem Hara has worked on implementing a performance analysis tool; Poman Leung together with Berson did the object world and a TROLL interface to CommonLisp; Dorab Patel acted as a general consultant; Ivan Tou helped organize the source code and provided thoughtful comments on my writing; Elsie Wu did many things, she built a library of geometric figures, a graphic editor a selection tool helped to build the OREL graphic editor and maintained the computer system of our lab.

I also wish to thank all my friends who helped make this years enjoyable. In particular I would like to mention Yadran Eterovic, my friend of many, many years and the father of my only godson.

And of course I am greatly indebted to my wife Estela, and my children Macarena and Sergio Alfonso. They mostly gave up having a husband and father for the past five years.

VITA

1976	Graduated from Pontificia Universidad Catolica de Chile
1978	M.S., University of California, Los Angeles
1978	Teaching Assistant, University of California, Los Angeles
1978-1980	Assistant Professor, Pontificia Universidad Catolica de Chile
1980-1981	Associate Professor, and Chair of Computer Science, Pontificia Universidad Catolica de Chile
1981	Distinguished Service Award, IEEE Section Chile
1980-1982	Member of the Board of Directors, IEEE Section Chile
1982-	Associate Professor, Universidad de Santiago de Chile
1982-1984	Chair, Computer Engineering Area, Universidad de Santiago de Chile
1985-1987	Post-graduate Research Engineer, University of California, Los Angeles

1987-1990	Research Associate, University of California, Los Angeles
1990-	Sr. Computer Scientist, Perceptronics, Woodland Hills, California

ABSTRACT OF THE DISSERTATION

A Computer-based Environment
for Collaborative Design

by

Sergio Tadeo Mujica

Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1991
Professor Gerald Estrin, Chair

This research explores the extent to which computer systems can support teamwork by small groups involved in the design and realization of complex systems. Our attention is on groups such as: engineers collaborating on a system design; customers, users and developers seeking agreement on system requirements; and other groups which may be able to use the computer support to expose and reconcile differences in viewpoint. Toward those goals, this research has created a laboratory, containing software and hardware systems that encourage fundamental and systematic experimentation. The driving force, and main contribution, of this research is the creation of such an environment, to establish a proof of concept for feasibility of effective collaborative design environments.

This research has also produced an object-oriented system for distributed, interactive sharing of data that includes: a graphic

language for modeling using concepts of objects and relations, a programming system for building programs that manipulate data, support for distributed interactive sharing of data, and persistent storage of data; and a tool modeling and integration method that includes a tool model for environment extensibility and interactive tool sharing that allows partial and full integration of tools, enables tools to operate on interactively shared objects, and supports the incremental extension of the environment. The tool modeling and integration method 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.

CHAPTER 1

Introduction

Exploration of computer-supported cooperative work (Greif [1988]), and investigation of *groupware* technologies (Groupware Technology Workshop [1989]) have accelerated during recent years. This research explores the extent to which computer systems can support teamwork by small groups involved in the design and realization of complex systems. Currently we are overwhelmed by the dearth of support for reasoning and arguing about the validity of design models and design decisions *before* they display themselves in unpredicted behavior of working systems. Our attention is on groups such as: engineers collaborating on a system design; customers, users and developers seeking agreement on system requirements; and other groups which may be able to use the computer support to expose and reconcile differences in viewpoint. Toward those goals, this research has created a laboratory, containing software and hardware systems that encourage fundamental and systematic experimentation. Problems which we have investigated in order to create an effective collaborative design laboratory are:

- How to enable *interactive sharing* of design objects by a small group of designers, such that all collaborators have up-to-date knowledge about changes taking place.

- How to integrate new methods and tools which can be used by collaborators without excessive artifact.

- How to manage display which is common to all collaborators.

- How to capture the essence of face-to-face discussions or remote computer-based teleconference sessions.

- How to expose feasible solution spaces when there are conflicts in the goals of the collaborators?

This research effort has created an environment for collaborative design, called coSARA, whose development has been motivated and focused by previous work on design methods and tools to support design of computer based systems in a UCLA project called SARA (System ARchitect's Apprentice) (Estrin et al. [1986]). coSARA contains software and hardware systems, that will encourage fundamental and systematic experimentation towards understanding of those problems outlined above. It is necessary to uncover the nature of computer support that would prove to be effective in augmenting group capabilities, exposing weaknesses and infertile avenues, and testing the suitability of enabling computer technology used in its implementation.

The driving force, and main contribution, of this research is the creation of such an environment, to establish a proof of concept for feasibility of effective collaborative design environments.

This research has also produced two other significant contributions that provide a foundation for building computer-based collaborative environments:

- *An object-oriented system for distributed, interactive sharing of data.* This system includes: a graphic language for modeling

data using concepts of objects and relations, a programming system for building programs that manipulate data, support for distributed interactive sharing of data, and persistent storage of data.

- *A tool modeling and integration method.* This method: includes a tool model for environment extensibility and interactive tool sharing that allows partial and full integration of tools; enables tools to operate on interactively 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.

This dissertation is organized in 6 chapters:

Chapter 1 is this introduction.

Chapter 2 contains a survey of related work and presents a number of concepts that will be used in the following chapters.

Chapter 3 defines a set of functional requirements that characterize the coSARA environment for collaborative design.

Chapter 4 describes OREL, an object oriented system for distributed interactive sharing of data.

Chapter 5 discusses the coSARA tool modeling and integration methodology.

Chapter 6 presents conclusions and directions for future research.

CHAPTER 2

Related Work

Computer-Supported Cooperative Work (CSCW) is a relatively new area of research and development which deals with the role of computers in group work, exposing ways in which usage of computers can enhance the ability to do effective work by groups of people who are cooperatively pursuing a goal (Greif [1988]).

Work in CSCW encompasses a broad area of knowledge. Computer science foundations are essential in building the complex hardware and software systems that are required. Social sciences are of great relevance, because the proper understanding of human behavior will determine, to a significant extent, the impact of introducing computer technology for group work.

In this chapter we review research by others who have been dealing with problems closely related to this study. The areas covered include computer based conferencing, design environments, user interfaces, concurency control in database systems, object oriented systems, and very recent results reported at an August 89 workshop on groupware technology.

2.1. Computer Based Conferencing

A conference is the interaction of two or more people who do it to accomplish a defined goal. When computer-based tools are used to support a conference, we talk of a computer-based conference. The key issue in this area is that computer-based tools have the potential to provide support beyond that of a simple communication medium.

An example of using the computer as a communication medium is the commonly used Unix™[1] "talk" program that allows a pair of users to exchange messages in real-time. It does not make a record of the conversation, nor provide access to other processing abilities. We regard the kind of support provided by ``talk'' as a weak example of a computer-based conference, because of the absolute lack of processing capability. We require that a computer-based conference offers some sort of processing capability that can be applied in useful ways to the contents of the conference.

Computer-based conferences can be classified according to two coordinates: location and timing. They can occur at one place in face-to-face settings or they can occur with participants in geographically remote locations. Conferences can occur:

- In real-time when all participants are required to be present and act in the conference simultaneously (for example in a face-to-face meeting), in synchronization with each other's actions (not everyone talks at the same time.)
- In a non-synchronized way when the simultaneous presence of participants is not required; participants can review the progress of the conference and provide input as desired (for

[1] Unix is registered trademark of AT&T.

example, a discussion that uses electronic mail or a bulletin board).

In the following sections we describe systems that have been built or proposed to support computer-based conferences.

2.1.1. NLS and AUGMENT

An early project that explored the area of CSCW was led by Douglas Engelbart at the Stanford Research Institute (Engelbart & English [1968]). This research was aimed at developing principles and techniques for designing an "augmentation system." Engelbart and English describes this notion as follows:

> "This includes concern not only for the technology of providing interactive computer service, but also for changes both in ways of conceptualizing, visualizing and organizing working material, and in procedures and methods for working individually and cooperatively...
>
> Among the special activities of the group, are the evolutionary development of a complex hardware-software system, the design of new task procedures for the systems' users, and careful documentation of the evolving system design and user procedures.

> The group also has the usual activities of managing its activities, keeping up with outside developments, publishing reports, etc.
>
> Hence, the particulars of the augmentation system evolving here will reflect the nature of these tasks—i.e., the system is aimed at augmenting a system-development project team."

The SRI project developed an experimental laboratory centered around an "interactive, multiconsole computer-display system." The system included a conference room equipped with six displays, meant to be used by up to twenty participants in face-to-face to meetings. One participant would control the system and all displays would show the same view. The capability of true collaboration, giving each participant the possibility of controlling a workstation in significant ways, was not present in this early work. It was viewed as a needed improvement:

> "We are anxious to see what special conventions and procedures will evolve to allow us to harness a number of independent consoles within a conference group. This obviously has considerable potential."

The software system underlying this development effort, called the On-Line System (NLS) was built using a set of translators that would generate the internal parts of NLS given a specification of a control processor and of a library of subroutines. The control processor

receives and processes successive user actions and calls upon subroutines to provide services such as: displaying feedback on the screen, locating data in files, manipulating working data, constructing a display view of specified data according to given viewing parameters, etc.

More recently, support for multi-party collaboration, using the principles of NLS has been implemented in AUGMENT (Engelbart [1984]), a text processing system marketed by Tymeshare Inc.[2] for a multi-user, network environment.
This support amounts to:

- A mail system for asynchronous communication, in which links among messages can be made, in a hypertext style (Conklin [1987]). Engelbart highlights this functionality as useful for recording and managing discussions.

- Shared-screen teleconferencing, that allows two users to operate as if they were sitting in front of the same terminal, seeing the same screen and with the ability to alternatively take the controls. When two users enter this mode of operation from remotely located workstations, their screen will look exactly the same and there are provisions for passing control back and forth between workers. There are

[2] Cupertino, California.

also provisions for the subsequent entry and departure of other conference participants.

2.1.2. RTCAL and Mblink, real-time conferences

Irene Greif and Sunil Sarin have done extensive research in the nature of computer-based tools to support real-time conferences and have identified important issues in the design of such (Sarin & Greif [1985]). They have developed two prototype systems that demonstrate two ends of the spectrum of possible implementation strategies. The first, RTCAL, supports a single application activity in a real-time conference. The second, Mblink, is an example of a generic facility that can be used for multiple applications.

2.1.2.1. Real-Time Conferences for Scheduling Meetings

The first prototype, RTCAL, supports meeting scheduling by building a shared workspace of information form participants' on-line calendars. Some of the most notable features of RTCAL are:

- Provision of shared and private spaces to the users. The common calendar view is shared and visible by all the users; each user has also a view of his or her own private calendar.

- Separation of application and conference control. It is intended that different applications that can run in a conference setting, have a uniform set of commands for conference control.

- Definition of conference roles. There is a chairperson who oversees all the activity of the conference and determines who has control at any given time, and is the only person who can terminate the conference.

- Alignment of information in the shared and private spaces. Windows are automatically scrolled so both display the same range of data and time.

- Voting as a means to support group decision processes is incorporated in the system.

- Participants can leave the conference at any time and re-enter it at a later time. The shared space is automatically refreshed upon re-entry. The participants however do not have autonomy to individually browse private calendars.

2.1.2.2. A Shared Bitmap System

Mblink is a shared bitmap system that is an example of a general facility usable for multiple different applications. Some characteristics of this system are:

- Applications run in a central mainframe and display in workstations. Each workstation reports the position of its mouse and the state of the mouse buttons to the bit-map module on the the mainframe.

- Each participant can see, on his workstation screen, the position of every participant's mouse.

- A participant has two different views of his or her own cursor: the first is tracked by the local system and the second is echoed by the mainframe. When the mouse is moved the echoed version lags a small amount of time (Sarin & Greif [1985] report about half a second), The intended application of this feature is to let users know what other users are viewing in their screen, as indicated by the cursor image echoed by the mainframe.

Sarin & Greif [1985] report that Mblink has been successfully used in implementing an application for experimenting with different internetwork congestion-control algorithms. They claim the implementation effort was small.

2.1.2.3. Design Issues in Computer-Based Conference Systems

Greif and Sarin identify the following design issues:

- User interface uniformity.

  Provision of a consistent set of commands to support conference control across the complete range of applications, and a consistent user interface for conference work as well as for individual work—i.e., the editing commands should be

the same regardless of whether the user is editing in a conference session or is working individually.

- Shared vs. individual views.

  Indications of special meanings that an object may have to a user are necessary, For example a user needs to know that he or she owns a lock on a particular object or that another user is locking some object.

- Access control.

  It is often appropriate to control who can gain access to a particular conference and shared data. Existing access control techniques can be applied to decide which users will be granted those privileges. For example a person could find out about a conference in progress and submit a request to join it. The request could then be either approved or denied by a participant who is empowered to do so.

- Concurrency control.

  If two or more users are working simultaneously on the same shared data space, it is likely that conflicting updates occur. Concurrency control mechanisms to prevent this are necessary. Two key issues are the granularity at which data objects are locked and the policies for passing control from one participant to another.

- Constraints on real-time conference designs.

  Because of the demand on processing and communication originated by real-time conferences, a limit in the range of conference sizes may be necessary. Also, conferences should run at the level of support that is the least common denominator of the workstations involved in the process.

2.1.3. COLAB: Beyond the Chalkboard

Colab (Stefik et al. [1987]) is a meeting room that includes a workstation for each participant and a large screen at the front of the room with an independent controlling device. All the workstations as well as the large screen are connected in a common local-area network. The large screen makes a shared space of data visible to all the participants at the same time. Colab has been used as a platform to test tools for collaboration. Stefik et al. [1987] report the usage of two tools: the Cognoter, a tool to aid in brainstorming sessions; and Argnoter, a tool to support the evaluation of specific proposals. These tools reflect an important property of Colab: the existence of process models for problem solving.

Colab has a multi-user interface (Foster [1986]) that is claimed to let participants interact with each other easily and immediately through a computer medium. The paradigm on which this interface is based has been called WYSIWIS (What You See Is What I See) referring to

the presentation of consistent images of shared information to all participants.

Stefik et al. [1987] report that the usage of a strict form of WYSIWIS is too restricting. For example it is often necessary to view private information (as is done in RTCAL), action that would violate the strict WYSIWIS principle. The simultaneous display of the pointers of all active participants has also been found to be excessively distracting. These and other experiences with the use of this system are reported by the same group (Stefik et al. [1987], "WYSYWIS Revised:..."), and evaluated by considering constraints on four dimensions of WYSIWIS: display space, time of display, subgroup population, and congruence of view. Possible relaxations of WYSIWIS along these dimensions are discussed.

The usage of the electronic board (the large screen) is interesting, since it provides a tool to focus the attention of the group, giving a common view of data at the same time and in the same place to all participants. It is built to provide functionality similar to a common chalkboard. It has a large drawing area, chalk, eraser, etc. To draw one picks up the chalk clicking on the the chalk icon. Using this device it is possible to make free-hand sketches. These marks can be eliminated with the eraser, that is also represented as an icon.

2.1.3.1. Implementation Issues in COLAB

The software system is implemented on Xerox Lisp Machines using Loops, an object oriented language (Stefik & Bobrow [1983]). Initial goals placed on Colab were: short time to get information, short delay in changing information, the database should converge quickly to a consistent state and the database should not be vulnerable to user mistakes or equipment failures.

A cooperative model of concurrency control is used. In this approach each machine has a copy of the database, and changes are installed broadcasting the changes without any synchronization. This method can of course lead to conflicting updates in data due to race conditions. Since the participants of Colab sessions are aware of this problem, they will use verbal cues to coordinate their behavior (the concurrency control problem is in fact left to the social system of the meeting to solve.) It is also expected that the mechanisms that may later be provided in Colab to coordinate group work activities would help in this respect.

The result of this solution to the problem of concurrency control is reported to be unsatisfactory and an intention to investigate the use of two-phase locking and time stamps is declared.[3]

[3] Concurrency control techniques such as two-phase locking and time stamps are discussed thoroughly in (Bernstein, Hadzilacos & Goodman [1987])

In Colab, the term conversation refers to a set of machines, Colab tools and participants working together to solve a problem. Communication in conversations is done by several layers of protocols over an Ethernet. Communication among Colab tools is accomplished via a programming abstraction that they call broadcast methods. When a broadcast method is invoked on one machine, it is also executed in all the machines that are part of the conversation.

2.1.3.2. Observations

One starting premise in Colab was that ideally equal participation characterizes collaboration (stefik et al. [1987]):

> "...By equalizing access of all participants to displays and shared data, The Colab's interface enhances flexibility as to roles and discourages control over the activity by one participant."

However limitations in this regard imposed by current technology turned out to be useful. For example, the fact that a particular writing technology allows only one person to write at a time, forces a shared focus on that person's actions, maintaining a common context for the group and helping to expose roles.

It has been observed that the meetings using the Colab's Cognoter tool are composed of cycles, in which unstructured verbal interaction is followed by short periods of time when each participant works on the computer to join again in verbal exchange of ideas. The verbal

exchanges are used by participants to set up immediate goals and short term plans of action.

It also seems to be a valid conclusion of the experimentation done in the Colab environment that meeting participants not only need access to the finished product of each other, but also to the production process itself.

Some relevant design issues discussed in this report are:

- The WYSIWIS display of cursors from multiple users is unacceptably distracting.
- Small grain size of data allows small grain size collaboration but is computationally too expensive.
- Multiple shared spaces of data are often needed simultaneously. but current screen technology may not provide enough space to display all of them.
- The screen may become crowded with screens used by other participants.
- When all the shared data can not be viewed (most likely because of the display size) it is not possible to quickly assess what is changing and by how much.

2.1.4. Data Sharing for Collaboration

One central concern in CSCW is coordinated access to shared data by collaborators. Greif & Sarin [1987] discuss data management requirements of CSCW applications and identify areas requiring further development, as presented below.

A major bottleneck in the development process and in performance as well, is the need to manage data outside of the address space of an executing program; including long-lived data that persists between program invocations and data that are communicated between program address spaces in real time. These issues are addressed in later sections.

The dissociation between data models used in programming languages and database systems is also a problem in utilizing the latter to support data management in CSCW applications. There is currently a trend in development of object oriented database systems that provide a point of convergence for programming languages and database technology.

RTCAL (Sarin & Greif [1985]), a real-time conferencing system for scheduling meetings exhibits the need for display of shared and private information as well as the use of roles to control access to the information.

CES (Seliger [1985], a system for collaborative editing of hierarchically arranged documents, exemplifies locking requirements. It promotes the use of unobtrusive lock operations. When several writers attempt to edit the same section of a document one is granted a lock and the others are informed of who holds the lock. Locks are obtained implicitly when editing starts and is held as long as some editing activity continues. The lock will be released to a new author after some idle period. This requires frequent updates of the information in permanent storage while an item is locked. If the lock is released without the explicit consent of the author, the committed changes will still be incorporated in the document.

CES also incorporates the concept of dirty reading, allowing users to read text that is being modified. The readers view will be refreshed periodically as the section is updated.

Access and concurrency control are critical issues in managing shared data for CSCW applications. A problem with current technology for access control is that the facilities usually provided by system software operate at the file level as it happens in UNIX. Access rights to files usually depend on the user and file that are involved in the access operation.

On the other hand it is usual that for collaboration purposes, access control is needed at finer grain. For example, in RTCAL it is necessary to control access to the operations for creating, viewing and modifying calendar entries. To accomplish this finer level of access

control, in a system that supports access control at the level of files, either the files are left unprotected, making the system vulnerable to accidental or malicious modification, or the programs that enforce the access rights to data are run in a privileged mode.

The typical UNIX-like access criteria based on owner, group and public is often insufficient to implement the richness of control required in a cooperative application like RTCAL. Other properties may be used to determine access: Users may play different roles and therefore have different access privileges depending on their current role; access may depend on the creator of a data item For example, in the calendar application, only the creator of a meeting may be allowed to cancel the meeting. Different users may be allowed to violate particular database integrity constraints, for example a trusted group of users may enter conflicting entries in a calendar, but the public may not be allowed to do so.

Long-lived transactions take a long elapsed time to complete; for example a document editing session can be considered to be a long lived transaction, a cash withdrawal at an automatic teller machine would not be a long-lived transaction. Long transactions are more vulnerable than short ones to system failures. This can be alleviated by supporting saving points within a transaction. A transaction in this domain need not be limited to a single user session. The user who is operating on the system may choose to interrupt the work and resume it at a later time. In other scenarios, the actions that need to

be performed may be known in advance and can be recorded in a script to guarantee its execution.

Transaction schedulers are often quite rigid in their ways to determine which transaction should wait or abort in case of conflict. More flexible ways to resolve conflicts are required by cooperative work. The CES lock timeout exemplifies this notion, providing a measure of fairness to the co-authors of a document. In general, it is useful to provide information regarding the holders of locks when conflict arises, allowing informal negotiation of schedules.

2.1.5. Design of Cooperative Work

Winograd and Flores present a perspective on computer-based systems, that takes language as the primary dimension of human cooperative activity , (Winograd [1986]; Winograd & Flores 1986]). This approach has been the basis for The Coordinator[4], a commercial system, based on the theoretical work done by Winograd and Flores.

The concept ``People act through language," is exemplified by Winograd (Winograd [1986]; section 2) as follows:

> "Consider a situation in which a hospital nurse calls the pharmacy, finds out what drugs are available and orders

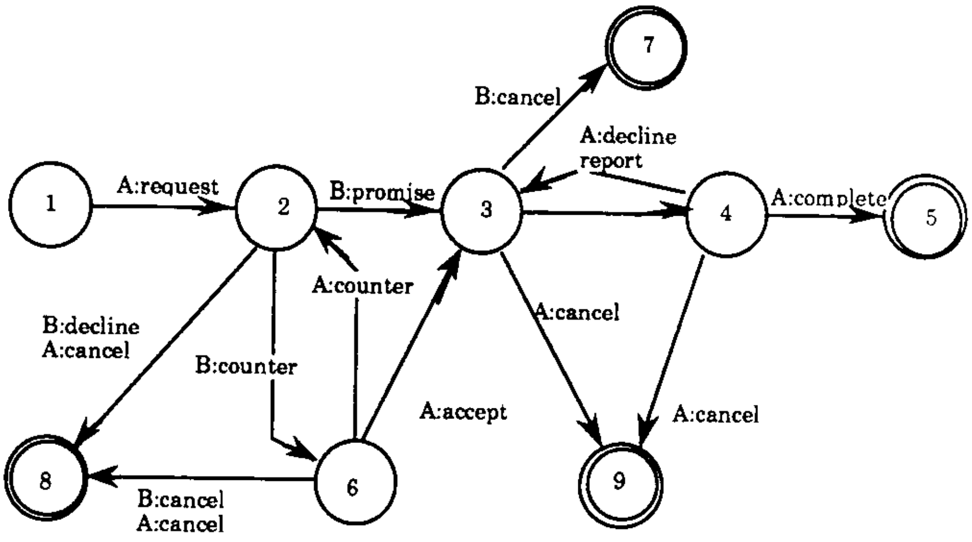
[4] The Coordinator is asystem created by Action Technologies, Inc. The coordinator is a registered trademark of Action Technologies.

> one of them for a patient. From an information-processing perspective, we could focus on the database of information about the drugs and the rules for deciding what drug to order. From a language/action perspective, we focus on the act of ordering and on the patterns of interaction in related conversations, such as the preliminary conversation about drug availability and the subsequent conversation that unfolds in the process of fulfilling the orders. From other perspectives we might consider such things as the personal relationship between nurse and pharmacist, the cost-effectiveness of making the communication over a phone, or the legal status of orders placed by a nurse."

2.1.6. Conversations for Action

Speech acts are not unrelated events, they participate in larger conversation structures. An important example is the ``conversation for action," in which one party A makes a request to another party, B. This conversation can be modeled using the state transition diagram of Figure 2.1. After A's initial request, interpreted by B as having certain conditions for satisfaction, B can accept and thereby promise to satisfy those conditions; decline, ending the conversation; or counter with alternative conditions, etc.

The diagram intends to show how the conversation proceeds towards mutual recognition that the requested action has been done or that the the conversation is complete without it having been done.



State transition network representing a conversation for action initiated by a request from speaker A to speaker B. The circles represent conversation states and the labeled line represent speech acts. Double-line circles represent states of completion. (This figure is adapted from (Winograd & Flores [1986], page 65.)

Figure 2.1: Conversation for action model

2.1.6.1. Other Types of Conversation

There are several kinds of language acts that do not participate directly in conversations for action. For example, a remark such as "They are planning to remodel Boelter Hall next year" need not relate directly to any specific future action of speaker or hearer. The following classify different kinds of conversation:

- Conversations for clarification.

  This type of conversation occurs when there is a background that is partially shared or that needs to be negotiated. For example, the request "Give me a glass of water" might evoke a response like "Large or small?."

- Conversations for possibilities.

  In this type of conversation the principal activity is speculation anticipating the subsequent generation of conversations for action. Brainstorming sessions such as those supported by Cognoter in the Colab are an example of conversations for possibilities.

- Conversations for orientation.

  These conversations are for creation of a shared background as a basis for future interpretation of other conversations.

2.1.6.2. Analysis and conclusions

Winograd and Flores consider conversations for action to be the central coordinating structure of human organizations. The emphasis is on language as an activity, not as the transmission of information or expression of thought. Only the control structure of the conversation is modeled, specifying how control is passed from one speaker to the other by speech acts.

The Coordinator is an example of basing a system on language theories without attempting to program understanding of natural language. All interpretations are made by the people that use the system. The only model of conversation known by The Coordinator is that of conversations for actions, and its main function is to monitor the completion of such conversations. It has no capability to introduce new conversation models.

2.2. Design Methods and Design Environments

This section reviews several existing and proposed environments for design as well as current research projects in the area.

2.2.1. The SARA Design Method and Tools

SARA (System ARchitects' Apprentice) promotes an interactive design philosophy and a framework within which computer-based tools can extend the capabilities of computer system designers and analysts (Estrin et al. [1986]).

This is a requirement driven design method, that prescribes a combination of top-down partition and bottom-up composition of systems with particular focus on interface between modules. The essence of the method is illustrated in Figure 2.2.

The steps in the process of design as prescribed in the method

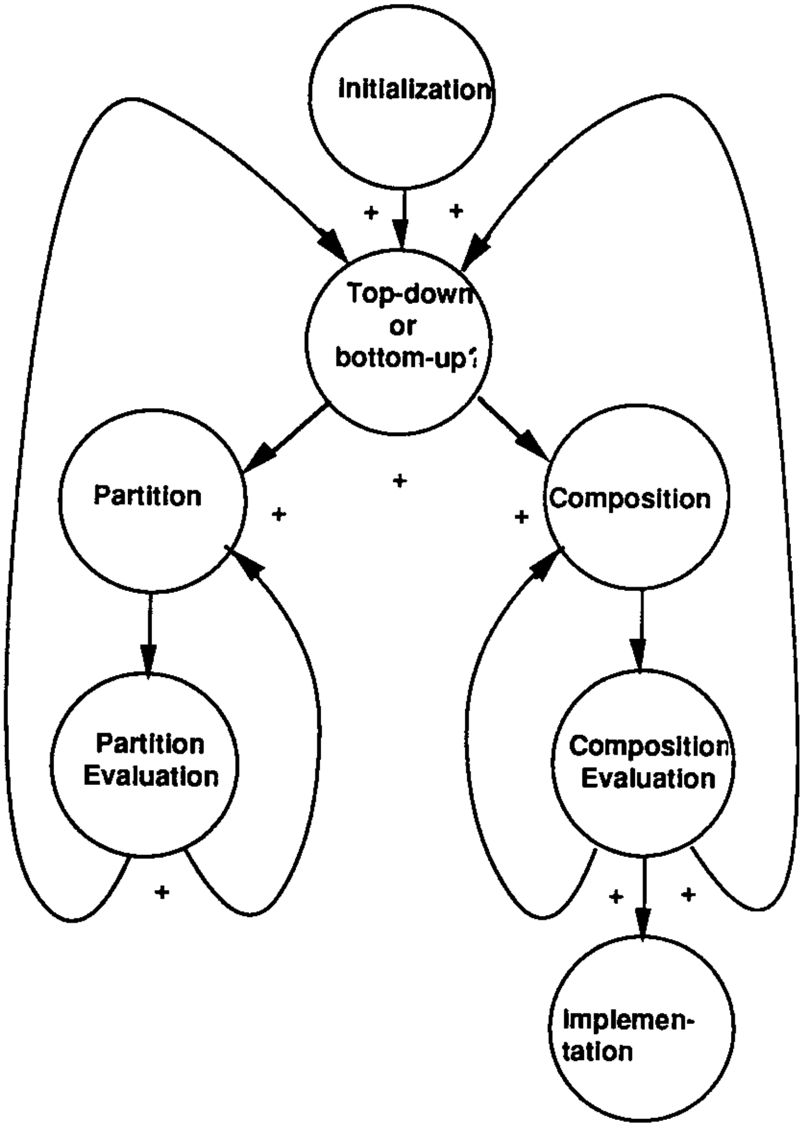


Figure 2.2: SARA Design Method

supported by SARA are:

- Initialization of design for any system or subsystem: establish requirements and constraints and their corresponding evaluation criteria; define assumptions about the behavior of the environment to which the system under design will interface; define behavioral models of the system and its environment; define a model of a test environment. No formal methods exist currently to support the definition of environment assumptions nor system requirements.

- Decide to proceed bottom-up or top-down: if there are predefined building blocks which can be composed to meet requirements, then a composition step can be taken. Otherwise the design continues with a partition of the system.

- Partition: the current structural model into submodules and initialize the design for each one of the resulting submodules. Build behavioral models for the resulting structure.

- Evaluation of partition: test the system applying evaluation criteria for requirements and constraints.

- Composition: build a composite behavioral and structural model from models of predefined building blocks.

- Evaluation of composition: test the composed model. If the test is successful and all subsystems have already been composed proceed with the implementation, otherwise continue the composition. If the test of the system fails then step back in the design and decide whether to proceed top-down or bottom-up.

- Implementation: Prepare documentation, manufacture, packaging.

SARA supports modeling of hierarchical modular structures and of behavior in three domains: control, data and interpretation. Graphic modeling primitives are discussed in Appendix I.

The SARA design method is supported by a collection of tools which are organized in the tree hierarchy shown in Figure 2.3. The role of each one of these tools is explained below:

- The SM editor allows the user to prepare a structural model of a system, that lays down the modules into which the system has been partitioned and the interconnections that model their relationships.

- The GMB editor supports the creation and modification of behavioral models of modules, that include models of control flow, dataflow and the interpretation of the transformations done in the data flow model.

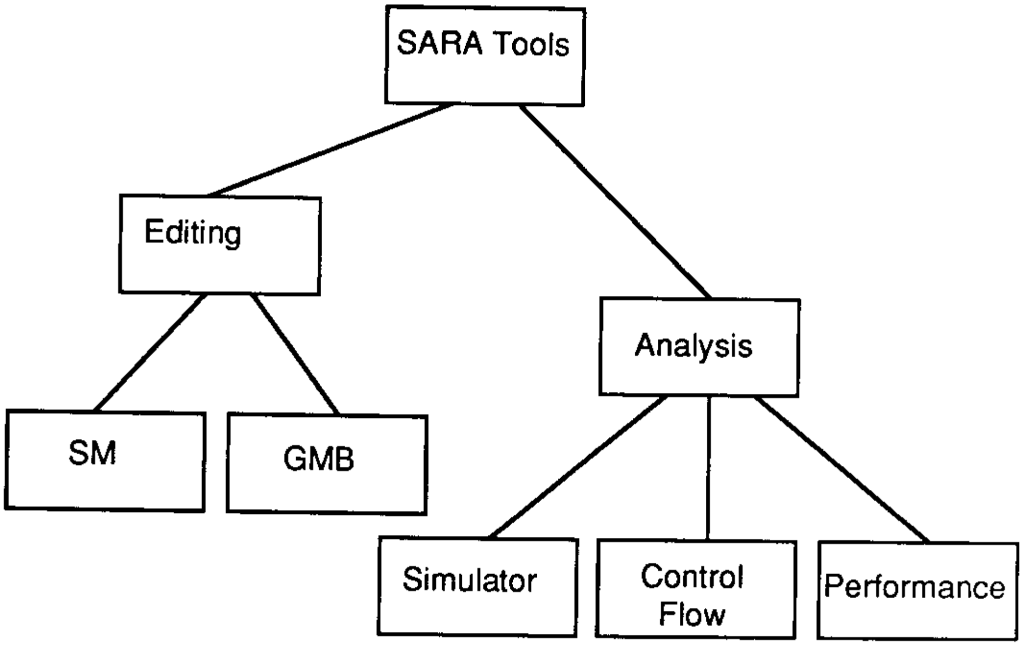


FIgure 2.3: SARA tools tree

- The GMB simulator is a tool for observing the behavior of systems and for interactive debugging of models. It provides animated graphical output and breakpoint and trace facilities.

- The control flow analyzer applies analysis algorithms to the control flow model of a system for detecting deadlocks and possible nonterminating states.

- The performance analyzer is a tool that allows queueing network modeling to be integrated with the behavioral model of a system. It uses the GMB simulator to produce performance measurements.

2.2.2. StP: Software through Pictures

StP (Wasserman & Pircher [1987]) is an environment that embodies a multitude of tools for software engineering based on structured design (Stevens & Myers [1974]). The system utilizes a powerful graphical interface that allows graphics editing of models of the system being analyzed or designed in data and control domains. The system is aimed at generation of substantial amounts of code from the models of the system built using the tools.

StP operates using the TROLL relational database system as a basis. It supports the rapid prototyping of user interfaces using RAPID/USE that is described in the section on user interfaces and provides facilities for integration of new tools.

Recently Wasserman, Pircher & Muller [1989] presented an object oriented, structured design method for code generation, whose integration in the StP environment is claimed to be in progress.

The method leans on a graphical language to specify classes and objects, in which operations are denoted explicitly. The notion of information cluster is introduced, matching the classical concept of object and class, as a module that encapsulates data and behavior. Use of a cluster by some other module is then represented by a connection between the user module and the representation of the operation that is used in the cluster. Only those operations that are actually used are represented graphically.

The model includes primitives to support features such as separate compilation of lexical units, clearly targeted in Ada Department of Defense [1983]).

Inheritance is treated from the point of view of data generalization, resembling generic packages in Ada.

The design method also accounts for the asynchronous activation of a module, supporting fork/join synchronization and message passing, as well as Ada's rendezvous.

2.2.3. STATEMATE

STATEMATE (Harel et al. 1988) is the implementation of a set of tools to model and analyze "reactive systems." Reactive systems cannot be adequately described by a statement of the functions from inputs to outputs, but involve the allowable combinations of inputs and outputs in time. The systems under design are viewed in three different ways, a structural view, a functional view, and a behavioral view. The structural view is similar to the SM, but it distinguishes modules that are intended to serve as data stores. It also distinguishes the system under design from its environment. The functional model is like the dataflow model, and it incorporates the control mapping. The behavioral model is a control model. The resulting graphs are not too messy due to the AND/OR decomposition of nodes.

2.2.4. Arcadia

The goals of the Arcadia project are: the discovery and development of environment architecture principles and creation of novel software development tools (Taylor et al. [1986]).

The term architecture is used to denote the set of rules and support infrastructure which characterize, bind together and enable utilization of the software development support tools existing within an environment. Object managers, user interface tools and tool activation managers may all be elements of an environment architecture.

Two principal qualities sought in the development of Arcadia are: extensibility, which refers to the ease of adding new capabilities to the environment; and integration, which refers to consistent user interfaces, easy context switching, efficient communication between tools.

The tools and objects manipulated in Arcadia are classified in three broad categories:

Basic components: these include the internal representation for programs, suitable for compilation, interpretation, analysis and program transformation.

Tool-building tools: these include tools such as lexer and parser generators.

Analysis tools: These include testing and debugging tools, design analysis tools and other tools applicable in pre-implementation stages of software development.

The process of developing software in Arcadia, depends on the process of creating, organizing, augmenting and exploiting a collection of persistent objects and aggregates of information. Arcadia users are encouraged to think of their work in these terms. Taylor et al. [1986] give the following example:

> "...rather than requesting the execution of a program, the user will request the display of the output of a program as applied to a specified set of data."

Objects in Arcadia are typed and are organized in a structure known as the Object Derivation Graph (ODG), which defines how objects are derived from other objects. This organization is compared by Taylor et al (1986) to that of RCS (Tichy [1982]), a system for version control. Hierarchy is also used to organize the object store. Users may define arrays or structures of objects that may be in turn organized as arrays and structures.

Arcadia is designed to support cooperative activities of teams of software developers and maintainers, working in separate workstations connected in a network. Each worker has a separate store of persistent software objects in his/her own workstation and also has access to objects stored in different workstations. Sharing of

objects is done using the principles of Software Federation (Heimbigner & McLeod [1985]).

In Arcadia a tool is a collection of tool fragments, temporarily allied, under the control of the environment, to complete some activity. For example, the tool fragments may be connected in a Unix pipeline. This notion contrasts with the more conventional concept in which a tool is more or less equivalent to a single program. Arcadia tools may be either passive or active. Active tools are executed without direct invocation by users. They perform according to predefined plans, and are invoked by Arcadia using devices such as timers and daemons that watch for relevant changes in the object store.

Creating larger tool capabilities out of smaller more general tool fragments is beneficial because, if the tool fragments are well chosen they will be usable for composing a variety of larger tools, at a lower cost. For example a pretty-printer can be composed using pieces such as a parser, a lexer and a formatter.

In many cases it is possible to determine which tool fragments will have to be invoked and in which order to accomplish a given task. However, there are cases when this is not possible. For example, consider a two-pass dataflow analyzer, that determines the order of analysis of the second pass during the first pass. The design of Arcadia includes a planning tool fragment whose job is to dynamically create tool fragment invocation sequences.

Arcadia tools are objects too, and there are tools to manipulate them. Using this approach a method for incorporating new tools is defined. It is required that at least one tool to create instances of new tools be written initially.

Inter-tool communication is done by remote procedure calls. When tight integration is desired, tools must share a set of common data structures, which are implemented as monitors.

2.2.5. The Programmer's Apprentice

The near-term goal of the Programmer's Apprentice (Rich & Waters [1988]) is to develop a system to provide "...intelligent assistance in all phases of the programming task." In the long-term the goals of this project are to develop a theory of how expert programmers analyze, synthesize, modify, explain, specify, verify and document programs.

The two basic principles underlying the Programmer's Apprentice are the assistant approach and inspection methods; it is realized using a Plan Calculus and a hybrid reasoning system.

2.2.5.1. The Assistant Approach

It is recognized that complete automation of the programming task is not a realistic near-term goal. A fundamental difficulty with a complete automation approach is the trade-off between the generality of a specification language and its ease of use and compilation

(writing a complete specification in first-order logic may be much harder than writing a program.)

An alternative approach is to assist programmers rather than replacing them. This was proposed by Harlan Mills in the form of chief-programmer teams in the early 70s. The goal is to provide each programmer with an computer-based assistant, called the Programmer's Apprentice. The Apprentice is regarded as a new agent in the software process rather than as a tool. It should interact with the programmer and have access to the tools that exist in the programming environment. The communication between the programmer and the Apprentice must be based on a substantial amount of shared knowledge.

2.2.5.2. Inspection Methods

Human programmers usually think in terms of commonly used combinations of elements with familiar names (called cliches.) For example, "device driver" is a cliche. An essential property of cliches is their relation to one another. For example a cliche may be a specific case or an extension of another cliche. Algorithm and data structure cliches may be related as possible implementations of specification cliches.

Given a library of cliches it may be possible to perform many programming tasks by inspection rather than by reasoning. In analysis by inspection, properties of a program are deduced

recognizing cliches. In synthesis by inspection, implementation decisions are made by recognizing specification cliches.

The Programmer's Apprentice focuses on the use of inspection methods to automate programming, rather than methods like deductive synthesis or program transformations.

2.2.5.3. The Plan Calculus

To apply inspection methods the cliches are represented in a concrete, machine-processable form: the Plan Calculus. This is essentially a hierarchical graph structure made up of different kinds of boxes representing operations and tests, and arrows representing flow of data and control. The representation has a graphical notation (Rich & Waters [1988, sidebar in page 13]) and a formal semantics for reasoning.

Taxonomic relationships between cliches, like specialization, are handled by special-purpose mechanisms in the cliche library. The relation between a specification and an implementation is represented by an "overlay", which defines a mapping from a set of instances of the implementation plan to the set of instances of the specifications.

2.2.5.4. The Hybrid Reasoning System

The approach used to reasoning about structured objects (programs, specifications, requirements) and their properties, is to use a combination of special-purpose techniques and general logic reasoning. Special-purpose representations and algorithms are used to avoid the combinatorial explosion of general logical reasoning systems. logic-based reasoning is used as the "glue" between inferences made in special purpose representations. This is implemented in a system called Cake, that is being used as base for the development of the Programmer's Apprentice.

Cake is structured in four layers:

1. The Propositional Logic Layer performs simple one-step deductions, records dependencies, and detects contradictions.

2. The Algebraic Layer contains special-purpose decision procedures for congruence closure, common algebraic properties of operators (commutativity, etc.) partial functions and the algebra of sets.

3. The Frames Layer of Cake support the notions of inheritance and instantiation.

4. The Plan Calculus Layer supports graph-theoretic manipulations of plan and overlay diagrams, such as

following arcs. It also implements the formal semantics of the Plan Calculus.

2.2.5.5. Programmer's Apprentice Scenarios

Rich & Waters [1988] describe three target scenarios for the usage of the Programmer's Apprentice, two in the area of implementation and design and one in the area of requirements.

In the first case, KBEmacs (knowledge-based editor in Emacs (Stallman [1981]) is used to demonstrate how implementation cliches aid in the construction of programs. KBEmacs adds a higher level of editing commands to the existing text and syntax based commands of Emacs. Changes in the algorithmic structure of a program can be achieved with a single command and may involve widespread textual modifications.

Two important capabilities of KBEmacs are the generation of program documentation, explaining the functionality in terms of the cliches used, and programming language independence. KBEmacs was originally written to handle Lisp programs and has been extended to operate on Ada programs with little effort.

KBEmacs was implemented before Cake was available, and the Plan Calculus is essentially the graph formalism without the associated logical reasoning.

The second case is the Design Apprentice, which extends the capabilities of KBEmacs into the realms of design. It is based on: a declarative input language, detection and explanation of programmer errors, and automatic selection of possible implementation choices. The Design Apprentice emphasizes the use of Cake to detect and explain programmer errors.

The third case is the Requirements Apprentice, that uses Cake as the underlying knowledge representation and reasoning system. Work on this prototype is at a very early stage of development. The Requirements Apprentice is expected to be of use in several stages of the requirements acquisition process, like achieving consensus among a group of end-users about what they want and in the transition from informal to formal requirements.

2.3. User Interfaces

The user interface of a system is that component of a system that collects input data from users of the system, passes collected data on to the system for processing and presents output data to users of the system. For example the user interface of a text editor collects text and editing commands form a user and displays the current state of the file text that is being edited. These concepts are discussed in detail in Chapter 5.

2.4. Concurrency Control And Recovery in Database Systems.

A typical example that illustrates the nature and need for concurrency control, is the airline reservation system. The problem is that many agents may be using the same data, and if this is not done "carefully", the same seat may be reserved twice. Processes that read and write the same data, should not run concurrently in conflicting ways. We regard operations that use shared data occurring atomically, as a transaction. The concept of serializability of transaction provides a basis for correctness. It is also necessary to consider the case when transaction fail for any reason; actions should be taken to prevent failed transactions from corrupting the database. A thorough discussion of these issues can be found in (Bernstein, Hadzilacos & Goodman [1987]; Ullman [1988]).

2.4.1. Transactions

A transaction is a single execution of a program that reads and writes data. Several independent executions of the same program may be in progress simultaneously; each is an independent transaction. It may also be the case that a number of different programs are being executed concurrently; each is an independent transaction as well.

A component of database systems is the transaction manager. Transaction management makes complex operations that read and write data appear atomic. That is:

- They act on shared data without interfering with other transactions.
- If they are completed normally, their effects are made permanent.
- If they are not completed normally, they have no effect on the data.

When a transaction execution is completed, its effects are made permanent in the database and it is said to be committed. If a transaction fails, and can not be brought to completion satisfactorily, it is said to abort. A transaction that has not yet committed or aborted is said to be active. A transaction that is uncommitted if it is either aborted or active.

A transaction may abort for a number of different reasons:

- The transaction itself issues an abort in face of errors from which it can not recover (you may be withdrawing money in an ATM, and do not have enough funds.)
- A system failure may interrupt the execution of the transaction (somebody accidentally unplugs the computer.)
- The database may abort a transaction because it detects that it has returned a wrong value to the transaction in response to a read operation (maybe, because another transaction has

been aborted, and we were given an uncommitted value written by that transaction.)

Commitment of a transaction guarantees that the effects of the transaction will be made permanent. (You do not want to leave the bank until the teller commits your deposit!)

For example, we consider an airline reservation system, we can define a "reservation" procedure $R(C)$ for a customer C as the sequence of operations:

1. $S \leftarrow$ read next available seat

2. mark S as reserved for C

3. write S

If two transactions that execute the reservation procedure are issued concurrently, say $R_1(C_1)$ and $R_2(C_2)$ it is possible that the same seat S_0 be reserved for both customers, C_1 and C_2. Table 2.1 shows a possible sequence of execution of steps in R that would lead to this situation, assuming that the list of next available seats is: $<S_0, S_1,...>$ and that the seat is no longer available for reservation when it is marked reserved for some customer.

	R_1	R_2
1	S ← read (= S_0)	
2		S ← read (= S_0)
3	mark S reserved for C_1	
4	write S	
5		mark S reserved for C_1
6		write S

Table 2.1: Execution of reservation transactions

It is easy to see how C_1 and C_2 ended having the same seat reserved; if the transactions had not been executed concurrently, but serially one after the other, different seats were reserved. Selling exactly the same seat twice is incorrect; assigning each seat only once would be correct but it would not be necessary that you get exactly the same seat under every possible order of execution of the transactions. If the transactions are executed as *(R_1;R_2)* the assignment of seats will be different than if they are executed in the order (*R_2;R_1*).

2.4.2. Serializability

An execution is serial if for every pair of transactions all the operations of one of them are done before any of the operations of the other transaction. Intuitively we can say that serial executions are correct because each transaction is individually correct by assumption, and transactions that execute serially cannot interfere with each other.

It is clear that a strictly serial execution would make poor usage of resources. In general, it is desired to interleave the execution of transactions. We say that a concurrent execution of a set of transactions is serializable if and only if it produces the same output and has the same effect on the database as a serial one. Since serial executions are assumed to be correct, serializability implies correctness.

All serial executions are equally correct according to this criterion. Therefore a set of transactions may execute in any order as long as the effect is the same as that of some serial order. It is of course possible that not all serial executions produce the same effect. If a particular order is preferred, it is the user's responsibility to ensure that the preferred order actually occurs. For example, in the case of reservation transactions discussed earlier, if two reservations are run for two different customers, they may get different seats depending on the actual order in which the transactions' operations are interleaved. If a particular assignment is desired, the burden of producing it is left to the user.

2.4.3. Recoverability

Database systems usually include a recovery subsystem (or recovery manager) whose mission is to make the database system behave as if the database contains all the effects of committed transactions and none of the uncommitted transactions.

If transactions never abort, then recovery is trivial because all transactions eventually commit. The recovery problem appears when transactions potentially abort.

When a transaction *T* aborts, the database system must remove its effects by restoring each data item written by *T* to a value it it would have if *T* had not taken place. Undoing a transaction is usually called ``roll-back''. Rolling back one transaction may give rise to cascading roll-backs that are often very expensive. This happens when transactions read uncommitted data. Let us consider the following example from (Bernstein, Gooman & Hadzilacos [1987]):

> Suppose that the initial values of x and y are *1*, and suppose T_1 and T_2 issue operations that the database system executes in the following order:
>
> $$\text{Write}_1(x, 2);\ \text{Read}_2(x);\ \text{Write}_2(y, 3);$$
>
> The subscript on each Read and Write denotes the transaction that issued it. Suppose that T_1 aborts and

Write$_1$(x, *2*) is undone restoring x to the value *1*. Since T_2 reads the value of x written by T_1, T_2 must be aborted too.

Now, let us build the notion of recoverability. We say that a transaction T_j reads x from transaction T_i if:

1. T_j reads x after it has been written by T_i;
2. T_i does not abort before T_j reads x; and
3. every transaction that writes x between the time when T_i writes x and the time when T_j reads x, aborts before T_j reads x

A transaction T_j reads from T_i if it reads some data item from T_i. An execution is recoverable if for every transaction T that commits, it commits after every other transaction from which it reads. Recoverability is required to ensure that an already committed transaction is not rolled-back.

Recoverability, however does not guarantee that cascading roll-backs will not occur. To avoid them we must ensure that every transaction reads only values written by committed transactions.

2.4.4. Concurrency control with locks

We previously said that transactions read and write data. To be more precise we will define the notion of item, as those units of data to which access is controlled. In general, the nature and size of items is for the designer to choose. For example, in relational databases, items could be entire relations, tuples or elements of tuples; in an object oriented database items could be classes, objects or slots (attributes) of objects.

The most common way to implement concurrency control is the use of locks. A lock manager is that part of the database system that records for each item *I*, whether one or more transactions are reading or writing any part of *I*. If so the lock manager will not grant locks to other transactions.

Choosing a large granularity of data items reduces the system overhead needed to maintain locks but may cut down allowable concurrency.

Concurrency control can also be implemented using other methods (timestamps, serializability, graph testing and certificators) to decide the legality of read and write operations on shared data.

We can define lock and unlock operations on data items, such that when a transaction locks a data item, that item cannot be used by any other transaction. In actual implementations there may be many different types of locks, permitting varying degrees of compatibility.

We will model the transactions as being composed of the following operations: start, commit, abort, read, write, lock and unlock which have the obvious meanings. We will also restrict the possible sequences to: start (read | write | lock | unlock)* (commit | abort), using regular expression notation.

Sequences of these operations will have an effect on the properties of the transactions, such as serializability, recoverability, deadlocks and cascading roll-backs. Below we explain such effects.

2.4.5. Deadlocks and Livelocks

A deadlock is a situation in which two or more tasks can not proceed because each one of them is holding a lock on some data item that the other transaction needs. Not being able to proceed, they are not able to release the locks either.

For example assume that we have transactions T_1 and T_2 which are executions of:

```
(progn
        (lock A)
        (lock B)
        ...)
```

and

```
(progn
        (lock B)
        (lock A)
        ...)
```

respectively.

Also assume that the computation proceeds as shown in Table 2.2. At time 3, none of the transactions can proceed since it will be impossible to grant the requested lock to one transaction without aborting the other one.

	T_1	T_2
1	lock A	
2		lock B

Table 2.2: Deadlock situation

Corrective actions include: detecting that deadlocks exist and aborting or restarting the transactions involved; granting all locks in a block, either all locks requested by a transaction are granted or none is granted; linearly ordering the resources to lock, and requiring the requests to be made in the predefined linear order. The first method (abort/restart) lets deadlocks occur. The last two avoid deadlocks.

Another situation, called livelock, may happen if a many transactions are requesting a lock on the same data item, let us say A, and they either get the lock or they have to re-issue the request. It is possible that every time some transaction T issues a request for A the lock is not granted. T will never be executed. A solution to this problem is to schedule the execution of the transaction in a FCFS manner (a scheduler with priorities and aging could also be used.)

2.4.6. Two-Phase Locking

Another problem that must be faced when designing a transaction system is how to guarantee correctness of the concurrent execution of sets of transactions. One solution is to use a protocol called two-phase locking (2PL for short.) In this case the transactions are divided in two consecutive phases. In the first lock are acquired and in the second locks are released. That is, no lock is released before all locks have been acquired. It has been proven that 2PL guarantees serializability of transactions (Bernstaein, Goodman & Hadzilacos [1987, Section 3.3]).

There are several variations of the 2PL protocol: conservative 2PL, and strict 2PL, which are of importance in different cases. Conservative 2PL requires that transactions request all locks before any other operation of the transaction is submitted for execution; the purpose of this scheme is to avoid aborting transactions because of deadlocks. Strict 2PL requires that transactions release all locks together when the transaction ends, after it is committed (or aborted); the purpose is to guarantee recoverability and avoid cascading roll-backs.

2.4.7. Locking Hierarchical Items

In many cases the data items are organized as hierarchical items. For example the modules in SARA models are hierarchically organized by the module-submodule relation. It is therefore convenient to devise locking mechanisms to handle these cases.

There are two basic approaches to this problem. The first is to lock individual objects in the hierarchy, the second is to lock complete subtrees.

To implement the first one, we use a locking protocol called tree-locking, that proceeds according to the following rules:

- The transaction locks a first item, which must not be locked by any other transaction.
- To lock a data item other than the first one, the transaction must hold a lock on the parent of the data item.
- When a transaction holds a lock in an item, it can release the lock of its parent.

Tree-locking guarantees serializability.

To implement locking of complete subtrees, a protocol called tree-warning, that works according to the following rules.

- To obtain a lock on a data item, a warning must be placed on every data item in the path from the root of the hierarchy to the data item.
- A lock on a data item implies a lock on all its descendants.
- The compatibility of locks and warnings is: warnings coexist with other warnings and exclude locks. Locks exclude warnings and locks.

Tree-warning locking requires 2PL to ensure serializability.

2.4.8. Long Transactions

It has been suggested by some authors (Greif &Sarin [1987]; Korth, Kim &Bancilhon [1987]) that the enforcement of serializability is not appropriate for long-lived interactive transactions, beacause it hinders concurrency and because it may be necessary to undo significant amounts of work. It is also pointed out that long-lived transactions are vulnerable to failures recurring during the transactions' life.

Proposals to alleviate these problems include the use of frequent saving points to ease recovery, human-handled cooperative scheduling for resolution of conflicts, and use of nested transactions that refine to fine-grain short transactions.

2.5. Object-Oriented Systems

Object-oriented systems have their roots in programming languages such as Simula (Dahl & Nygaard [1966]) and Smalltalk (Goldberg & Robson [1984]). Lochovsky [1989] contains a set of articles that review the state of the art in object-oriented systems. Concepts of object-oriented systems are discussed in detail in Chapter 4.

2.6. Recent Research Results

In this section we review current research closely related to our own work on collaborative design. This section is organized to cover recent work on: software and hardware environments for computer supported cooperative work, object oriented programming systems and databases, integrated design environments and computer system design methodology. Most of the material discussed here was presented at the Groupware Technology Workshop (IFIP Groupware Technology Workshop [1989]).

2.6.1. Software and Hardware Environments

There is a significant number of projects under development to produce software and hardware environments to support cooperative work. All known projects are aimed at providing basic and general support for cooperative activity, usually providing simple functionality for conferencing and editing.

The focus of many research projects is on dealing with multiple media, supporting voice, video and data, to enhance remote meetings. We describe below system that were discussed in the Groupware Technology Workshop: Rapport, Mermaid, the TeleCollaboration project, and ISI's Multimedia Conferencing project. Other current projects that deal with integration of video, audio and computer technologies in multimedia systems to support collaboration are: MM5 at Olivetti Research Center, VIDEO at Xerox PARC, MMConf at BBN and Bellcore's Integrated Media Architecture Laboratory.

Other systems focus more strongly on the mechanisms used to collaborate, such as concurrency control, viewing of shared data, coordination mechanisms, and others. The GROVE editor being developped at MCC focuses on concurrency control issues; the product under development at Brainstorm Inc. is intended to provide support for group decisions; work done at IBM T.J. Watson Research Center focuses on the communication of user actions among workstations, using animation techniques to smoothly reveal screen activities of other participants in meeting.

There are projects that have made efforts to deal with meeting room layouts to enhance collaboration, such as the University of Arizona's project and the Capture Lab of the Center for Machine Intelligence, Michigan. Both are described below.

2.6.1.1. The Rapport Conference System

The Rapport multimedia conferencing system has been developed at AT&T Bell Laboratories. It allows a group of people to hold real-time conferences sharing data, voice and images.

During a conference, the participants interact to produce and edit identical screens on their workstation screens, that contain information produced by conventional, single-user applications.

Rapport software provides two basic functions: support for interaction among conference participants, permitting initiation, conduct and termination of meetings; and support for sharing of displays among conference participants, coordinating the execution of application programs and maintaining consistency.

The original implementations of Rapport worked on the basis of single-user (collaboration-transparent applications. The attention of the implementors is shifting now to multi-user (collaboration-aware) applications. This is being explored with a program that allows its users to have both private and shared annotations, with both private and shared views of an underlying document. Private and shared annotations and views are implemented using private and shared workspaces.

Rapport is geared for remote, non face to face communication, and its suite of tools include the ability to display pictures of the conference participants, associating them with different colors for easy identification of telepointers.

In the case of Rapport the use of single-user, collaboration transparent tools, and the strict WYSIWIS approach, suggests that sharing is done mainly at the display level. It is not clear how data consistency is maintained.

2.6.1.2. Mermaid: a Distributed, Multimedia conference system

Mermaid, a system under development at the C&C System Research Laboratories of NEC Corporation, is a multimedia communication system to support collaborative work among multiple participants in a distributed office environment. The architecture of the system defines protocols for exchange of multimedia documents and conference coordination. Both, synchronous and asynchronous communication are supported in Mermaid.

The system operates using a client-server model. The clients supply functionality for preparing conferences, opening and closing conferences and other operations. Servers are classified into master, conference and domain servers.

The system contains a single master server, that supervises all the conferences that are conducted and replies to client inquiries about conference details.

One conference server is allocated per existing conference. These are responsible for conference control operations such as: convening, opening, closing, dynamic joining and leaving, presentation and floor passing.

A domain corresponds roughly to a set of workstations connected in a local area network. A domain server is responsible for data communication within its domain and between domains.

2.6.1.3. The TeleCollaboration Project

The TeleCollaboration Project project, being done at U.S. WEST Advanced Technologies, uses an iterative approach to designing and building collaborative systems, using a cycle of prototype development followed by observations via behavioral research methods.

The current prototype is used in support of a research organization that is intentionally dispersed geographically. The multimedia environment provides functionality for communication via video, audio, facsimile, remote control, data, text, shared computing and various combinations of these.

Communication is supported by several metaphors, such as phone calls and hallway wandering via video. There is a major concern about social issues in this project, as is demonstrated by the research approach used and the participation of social scientists in the research group.

2.6.1.4. The ISI system

The Multimedia Conferencing project has developed an ``experimental facility for realtime multisite conferencing''. It is aimed at geographically dispersed sites, and the emphasis is on multimedia technology and realtime communication.

This system can support up to four sites, each of which contributes video images that are displayed in a quadrant of a video screen located in the conference room.

MMConf is used as basic software to share single-user applications in a conference. There is no integration of the MMConf software with the conference room setup. Video and voice control software operates independently of MMConf.

Problems that have been perceived with usage of this system are: floor negotiation, which has to be done verbally due to delayed response of the software system; screen space management for video images, to accommodate more than four participants; resolution quality of video, transmitted using a packet switching mechanism in which packet damage or loss is common.

Usage of headphones has been perceived as better than loudspeakers. Using loudspeakers makes the conversations feel farther away and people tend to shout. Communication is more relaxed when headphones are used.

2.6.1.5. GROVE

GROVE is a simple outline editor, built at MCC, specifically designed for use by a group of people interacting synchronously. It is intended to support both, face to face and remote operation.

GROVE has been used as a prototype to explore implementation alternatives and problems of multi-user tools, and to collect informal observations on its use.

The main functionality of GROVE is to provide fine-grained concurrent editing capability. Several people could type into the same sentence at the same time and see each other's changes immediately. The main concepts embodied by GROVE are:

Session: A session is a set of GROVE processes engaged in editing the same outline. Each process is associated with a user who can enter and leave the session at any time. A session ends when there are no GROVE processes running.

Group window: A group window is a collection of individual windows, that may appear on different display surface (e.g., different workstations). All the windows that comprise a group window are maintained in identical state. For example scrolling one individual window, causes all the individual windows in the group to be scrolled identically.

View: A view is a portion of the outline being edited. There are three types of view: private, which contains items that only a particular user can access; shared, that contains items that a subset of all the users can access; and public, that contains items accessible by all the users.

This system maintains replicated copies of the data in each participant's workstation. The concurrency control algorithm used guarantees consistency of copies.

2.6.1.6. Arizona Meeting Room

The University of Arizona's Management Information Systems Department has created a relatively large meeting room, equipped with audio and video facilities.

The layout of the room is a U-shaped plan, with a large screen located at the open end of the U. It is claimed that the introduction of computer hardware does not affect the line of sight, so the group members have a good sight of each other.

The software provides explicit support for facilitation, and is aimed at brainstorming processes in which each user may add items anonymously.

The authors place a great amount of attention on the planning necessary for the introduction of this kind of meeting rooms in existing organizations.

2.6.1.7. The Capture Lab

The Capture Lab at the Center for Machine Intelligence, is a meeting room with software to support face to face meetings. As well as the Arizona meeting facility, this room makes an issue of not obstructing the participants line of sight and using a non-intrusive workstation plan.

The software system provides functionality to handle one large screen as a shared, WYSIWIS space, and treats the workstation's screens as private workspaces. The system is intended to support collaboration-transparent applications, so as to provide flexibility in the group's choice of tools.

2.6.2. Object-oriented Systems

There is agreement that object oriented systems constitute a solid foundation for the implementation of collaborative systems.

We describe below two important research efforts in the area of object oriented systems, that have as a goal to provide efficient management of shared persistent objects in distributed environments. Decouchant (Decouchant [1989]) focuses on extending Smalltalk with these features, whereas Moss (Moss [1989]) is trying to smoothly integrate database and programming languages concepts. The latter is a particularly promising approach, that seems to fit well with our own interest, but there is yet little information as to how that goal is accomplished.

2.6.2.1. Distributed Objects in Smalltalk-80

The distributed object manager for Smalltalk-80 [.decouchant 1989.] is based on: location transparency and uniform object naming, unique object representation and use of symbolic links for remote access, possibility of object migration, and distributed garbage collection.

The object manager is implelmented as a collection of cooperating local object managers, which run on each workstation. Local object managers provide a set of primitives to share and name objects without programmer awareness of the object's actual location. Other functionality of object managers include migration, garbage collection, connection and disconnection of sites.

In this system, an object name is always a reference to a local object. For remote references, there is a particular type of object called a ``proxy'' that locally represents a remote object.

The structure of the object manager consists of three processes: the network manager, the main memory manager and the secondary memory manager. These processes are used by client Smalltalk processes.

The network manager executes remote accesses on remote objects, serves requests from other network managers and controls the local processes. The main memory manager relocates objects, frees space, collects garbage, and resolves objects faults. The secondary memory manager stores and retrieves objects in secondary memory.

2.6.2.2. The Mneme Object System

The goal of the Mneme system (Moss [1989]) is to support cooperative, concurrent and reliable use of large , distributed collections of objects. The need for and benefits of independent collections of objects are described and implications on addressing issues are examined.

2.6.2.3. Integrated Design Systems

Collaboration transparency is accepted as an important problem in the design and implementation of environments for collaboration, because it allows existing tools, such as editors and spreadsheets, to be used within collaboration sessions.

A simple approach to attack this problem is to use shared window systems. In this case an application interacts with a shared window server, via a virtual terminal protocol. The shared window server routes the requests to individual window servers, to produce exactly the same window presentation in all the workstations involved.

The idea of a virtual terminal protocol can be traced back to the NLS project (Engelbart & English [1968]).

Shared window systems of this class help in integrating arbitrary tools into environments for collaboration, but bring sharing to a very low level of abstraction and do not contribute to solve problems of data sharing that are critical in collaboration.

CHAPTER 3

Functional Requirements and High Level Structural Design

Functional requirements to support collaborative design are described in two dimensions, requirements to support collaborative design activities, and requirements to support interaction and management of the software environment used for collaborative design.

A high level structural model of the collaborative design environment is described as a response to functional requirements. This high-level design motivates the introduction of an object oriented data model and a user interface development system in the following chapters.

3.1. Functionality to Support Collaborative Design

The functionality to support collaborative design should enable small groups of designers to share a common set of design objects, having a common view of the design process, established by agreement on a sharable design paradigm.

Sharing of design objects and cooperating in following a common design paradigm require that the designers communicate with each other and coordinate joint activities. Communication refers to the functionality needed to support exchange of information among members of a group, and coordination refers to the functionality needed for the group work to progress towards mutually agreed upon goals.

A design paradigm is sharable by a group of designers if it can be used as a basis to discuss and record design issues, make and record group decisions, perform and record design operations, and compile and trace design history.

3.1.1. Functionality to Share Design Objects

The design objects are *design representations*, *design history*, and *design tools*. Design representations encode models of a system that is being designed or has been designed. Design history contains information about prior design activity. Design tools are used to create and change design representations, to analyze and evaluate designs, to record and browse the history of the design, and to coordinate joint design activity.

As an example of sharing a design representation, consider a design model that has been partitioned into several modules, two of which are A and B. Ferruccio is working on A and Petroushka is working on B. It should be possible for both Ferruccio and Petroushka to see modules A and B on their respective workstations at the same time, for purposes of design discussion, e.g., agreement on the interface between A and B. Also, Ferruccio should not be allowed to make changes in B at the same time that Petroushka is changing B, unless both of them agree on a partition of B into two or more submodules, and work independently on each of them.

As an example of sharing design history, consider a case in which the two designers are making a design decision. Ferruccio and Petroushka may be examining the design history to determine possible interaction between the decision they are about to make and previous decisions. For this purpose they share the design history object, and share a common focus on the history object, i.e., the part of the history that is being observed. Once they have reached a decision, the design history must be updated by one of them, who will record the decision and its rationale. Sharing of design history is necessary for both communication and coordination.

As an example of sharing design tools, consider a tool for establishing the focus of a session, i.e., selecting a set of design objects to work with. There are two cases, the designers work with independent session foci wherein each uses a different instance of the tool, and the designers share a session focus by sharing the same instance of the tool.

It is required that design objects be sharable according to the following rules:

- The existence of all objects must be knowable by all of the collaborating designers at all times. The designers should know at least the name of the object and the name of the designers who are using it.

- A design object can be modified by one designer at a time; it can be viewed and used, but not modified, by more than one designer at a time.

- Changes, made to an object that is shared by two or more designers, can be seen by designers as they are made, unless the designer who is modifying the object has chosen to restrict access to it in order to prevent other designers from seeing changes until they are done.

3.1.2. Functionality for Communication

There are two modes of communication that must be supported by the system, synchronous and asynchronous communication. Communication may be either face to face or remote (Figure 3.1).

	Face-to-face	Remote
Synchronous	**•Rapid response** **•Social interaction**	•Rapid response, •Restricted communication
Asynchronous	•Delayed response •Social interaction	•Delayed response •Restricted communication

Figure 3.1: Location/Timing diagram for communication

Synchronous communication occurs when messages are exchanged between designers interactively, and each message requires a response, which is expected to occur in a short term. To enable synchronous communication among designers, the system must provide support to work on a common space in such a way that the action of one designer can be observed by other designers and to exchange and record short unstructured messages.

Asynchronous communication occurs when messages are exchanged between designers non-interactively. A response is not always required by senders of messages and if it is, it is not expected to occur in a short term. This form of communication does not require the simultaneous presence in the environment of designers other than the sender of the message.

It is necessary to distinguish between face-to-face and remote communication, because functionality that may be acceptable in one case may not be appropriate in the other. A tool that provides a highly structured communication may be appropriate for a remote conference, but provide too low a bandwidth for face-to-face interaction. On the other hand, in a face-to-face setting, it may be possible to rely on socially unstructured communication among designers to quickly achieve agreement on some coordination issue without support from the computer; this may not be realizable in a setting in which designers are interacting remotely, but may become feasible in a teleconferencing setting with multimedia support.

3.1.3. Functionality for Coordination

The functionality for coordination should enable group work to progress towards mutually agreed goals. Functionality is required for management of group focus, management of coordination milestones, group organization, and facilitation.

The system should provide a way to focus the attention of the group or any sub-group on a common set of design objects, for presentation and discussion purposes. The mechanism for setting the group focus should be able to operate in a strict WYSIWIS form(What You See Is What I See; section 2.1.1.2), displaying exactly the same picture either on a common large screen or in a designated window on each designers screen. It should also be able to operate in a relaxed WYSIWIS form, selecting a set of objects that would be displayed on designers' private screens following whatever preferences for graphical display each designer has.

Group coordination milestones are actions taken by the group as a whole, when deliverables are required to proceed with the design. For example, assume that Ferruccio and Petroushka are designing a system X that has been partitioned into modules A and B that interact with each other. To proceed to test the integration of modules A and B it is necessary that both modules have been designed and tested previously. The integration testing would therefore be a group coordination milestone. The functionality required for management of coordination milestones is: definition of coordination milestones, recording of milestones in the design history, scheduling milestones, review of pending milestones and recording completion of milestones.

The functionality required to organize the design group in useful ways is: admitting and deleting group members, passing or relinquishing control of design discussion, partitioning the group into subgroups, and merging subgroups.

The functionality required to support facilitation of group work is: provision of support for group interaction based on established group design protocols, provision of advice about following the design paradigm, assistance for recording discussion and design decisions in the design history and assistance for tracing the design history.

3.1.4. Functionality for Management and Use of the Design Environment

Functionality is required to support the following areas of the software environment: user interface, initialization, recovery and extension.

3.1.4.1. User Interface

The design environment should provide a user interface that the designers will use to manage the design objects. It is required that the user interface:

- Be able to work with graphical representations of design objects.
- Provide mechanisms to handle the graphical complexity of large models within the limited space of current workstations' screens.
- Give access to system documentation and help information.
- Offer direct manipulation of objects (Schneiderman [1983]).
- Allow the user to maintain multiple threads of control (Hill [1987.]).

3.1.4.2. Functionality for Initialization and Recovery

Functionality should be provided to initialize the system under four different circumstances system generation and configuration, initialization of work on a new design, initialization of a group session, and initialization of an individual designer's session.

Functionality should be provided to recover gracefully from either hardware, software and user errors. It should be possible to bring the system to the most recent consistent state, after operation has been aborted while objects are in an uncommitted state.

A recovery procedure should reinitialize the system to the same mode of operation that it had before the failure. For example, if a designer aborts a transaction his or her session should be brought to the state it had before the transaction was initiated, and no one else's session should be affected at all. If recovery is started after a failure that encompasses the whole system. For example, for a network failure, all existing sessions should be reinitialized and brought to a consistent state.

3.1.4.3. Functionality for Extension

There is a growing literature on CADIS (Computer Aided Design Information Systems) and CASE (Computer Aided Software Engineering.) In this work, it is assumed that demonstrable effective design methods and supporting tools will become candidates for integration into the Collaborative Design Environment (CDE).

Functionality must be provided to enable access to existing classes of design objects, and to create new classes of design objects and relate them to other, possibly existing, classes of objects. At later stages, it will be desirable to introduce new design tools and design paradigms. Three scenarios will be considered for the introduction of new tools:

Full integration: In this case the tool is built to operate on data that obeys the CDE's model of data. This is usually the case of tools built from scratch or of foreign tools that are built in such a way that it is possible to tailor them to the environments' model of data.

Partial Integration: In this case the tool is treated as an atomic piece. It should be possible to implement translation procedures that map data in the CDE's data model to whatever format is required by the tool and vice-versa. These translation procedures are specific for each foreign tool that is partially integrated in the CDE. It is an open question whether this transslation process can be generalized. The invocation of the foreign tool would occur by means of a method that is installed in the CDE that would spawn a UNIX process running the foreign tool and would instantiate the translation procedures. In this case the partially integrated tool would operate under its own user interface.

Unintegrated: If it is not feasible to provide some partial integration of a foreign tool into the CDE, it would still be possible for any foreign tool to be invoked from the CDE, creating a method that would start a process running the tool. Even though the foreign tool is not integrated in the sense that its output cannot be the input of another integrated tool, it should be possible to display the output on the screen and a designer should be able to inject the results of unintegrated tool usage into the design decision porcess.

3.2. High Level Structural Design

This section describes a high level structural model of the collaborative design environment, which is shown in Figure 3.2 as a SARA structural model (Estrin et al. 1986)[1] and a number of preliminary design decisions made regarding the implementation of the collaborative design environment.

Figure 3.2 displays a SARA model of the coSARA design. The two top modules in this design are the System module and the Environment module. The following assumptions are made about Environment:

- At most *K* designers may exist in Environment simultaneously, where *K* is a small integer (we expect 2 to 6 to be representative group sizes).
- Each designer uses a workstation to interact with System.
- System is reconfigured to include the newcomer designer.
- Designers issue requests to System using a workstation's mouse and keyboard, and read output from System on a workstation's display.

[1] See section 2.2 and Appendix I for a decription of SARA.

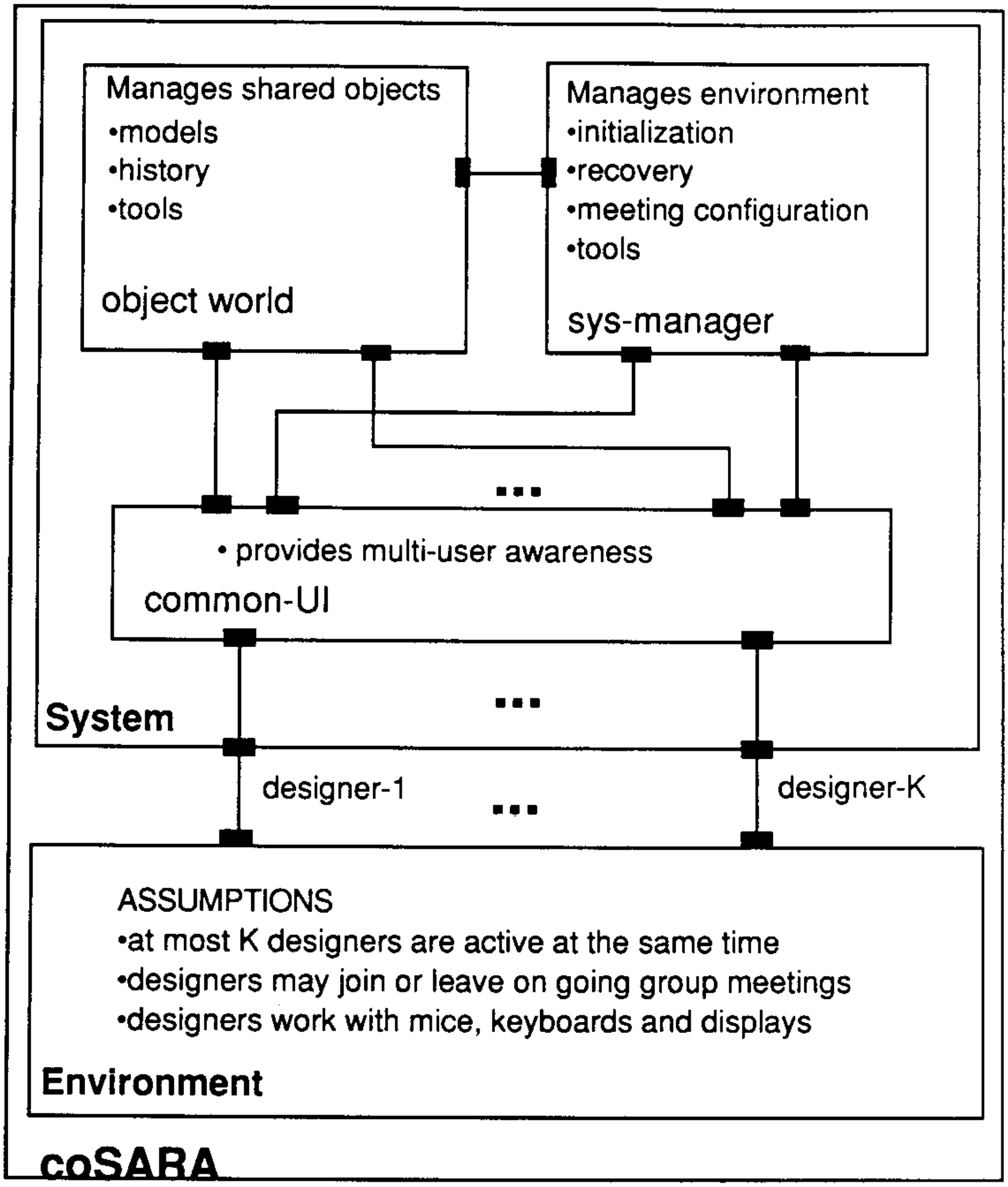


Figure 3.2: Structural model of the Collaborative Design Environment

The first partition of System contains the following modules:

common-UI: this module accepts requests from designers and interprets them, updating information contained in object-world as needed. common-UI also present output data to designers.

object-world: this module provides a common database to all the designers, such that the design objects created and used by designers and those provided by the system can be shared as required in Section 3.1.1 on sharable design objects. The functionality provided by this module includes: storage and retrieval of design objects, and transaction management.

sys-manager: this module takes care of initialization and recovery procedures, addition and removal of meeting participants, partitioning and merging of meeting groups and initiation of tools.

The following preliminary design decisions have been made about the implementations of the design described here.

- The collaborative design environment is initialized with the SARA design tools (Estrin et al. [1986]). There has been a great deal of effort put into the definition of these tools. We are also aware of several other tools which should be added to the set of SARA tools.

- The collaborative design environment will initially provide support for the SARA design paradigm (Estrin et al. [1986]). At a later time we will seek to extend the design paradigm ro deal with constraint management, reliability analysis, aplication of utility functions, etc.

- The collaborative design environment data model will be implemented using OREL (Chapter 4).

- The system is to be written using an object-oriented programming paradigm (Keene [1989]; Bobrow & Stefik [1986]; Goldberg & Robson [1984]), using CLOS (Steele [1990]).

  This decision is based on the following reasoning:

  Modularity: in writing object oriented software, there is a natural modularization guideline in the data structures. The inheritance mechanism also helps to exploit abstraction and generalization when common properties of several classes of objects are factored out.

  Extensibility: the ability to extend the meaning of predefined classes using the inheritance mechanism provides extensibility at a low cost. By manipulating the class structure it is possible to substantially redefine the semantics of the data. The ability to combine collections of methods in different ways is a powerful tool to incrementally develop systems.

Design paradigm: the design paradigms that will be supported are based on the management of design objects. Therefore an object oriented design and implementation should reduce the semantic gap between the end-product and its realization.

- Window management and graphics operation are done using X Windows (Scheiffler & Gettys [1986]) for portability.

- The system is to be written in CommonLisp (Steele [1990]) for the following reasons:

Ease of implementation: usage of Lisp in general (not necessarily CommonLisp) provides an interactive interpretive environment, which allows incremental modification of the code and enables the programmer to work at source code level. These facts ease debugging, (it must be noted however, that the available processors of CommonLisp do not possess particularly strong debuggers.)

Availability of CLOS: CLOS, described in section 4.6.2, is an object system for CommonLisp, which provides needed support for object oriented programming. CLOS however does not provide any support for persistent, shared objects.

Portability: It is clear the CommonLisp is becoming a widely used form of Lisp. Other approaches might have been to use languages as C, C++, Ada, etc. However these do not provide an interpretive environment. Furthermore, C does not have facilities for object oriented programming.

Ability to use off-the-shelf modules: Several components that seem to be essential for our implementation effort have been developed for CommonLisp. These include: PCL (Portable Common Loops) an implementation of the CLOS specification done at Xerox PARC; CLUE (CommonLisp User interface Environment,) an object oriented toolkit for X11 written at Texas Instrument (Kimbrough & Oren [1988]); and CLX, a CommonLisp library to operate the X11 protocol, designed and written at the MIT X Consortium and Texas Instrument (Scheiffler et al. [1989]). There are no such components for other combinations of Lisp (such as T (Slade [1987])) processors with the X Window System. There are of course other environments, such as Interlisp, that have similar modules and more, but they are closed and our portability requirements would not be met.

Figure 3.2 shows the composition of the CommonLisp environment using PCL, CLX and CLUE, in a ``uses-relation'' diagram.

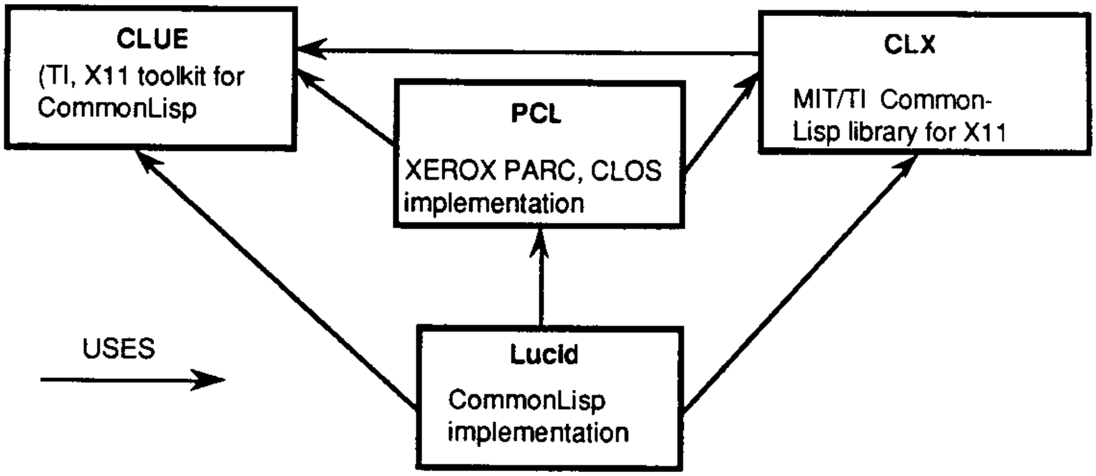


Figure 3.3: The Common Lisp Environment

Two versions of CommonLisp were evaluated: KCL (Kyoto CommonLisp), a public domain implementation that did not offer enough functionality and robustness for our requirements; and Lucid (Lucid CommonLisp manual, Lucid Inc. [1988]), which was found to be quite robust; it also offers good performance and integrates well with the software packages mentioned above. Lucid also offers multitasking, but no IPC (Inter-Process Communication) facility.

CHAPTER 4

OREL: An Object-Oriented System for Distributed, Interactive Sharing of Data

4.1. Introduction

This chapter discusses an object-oriented system for distributed interactive sharing of data, based on a graphic data modeling language that includes objects and relations as its principal elements. The system is named OREL (Object-RELation).

An essential set of OREL data modeling primitives is defined. An interactive graphic editor then supports construction of an OREL data model which can be translated, on the one hand , into DBMS schema and, on the other hand, into CommonLisp Object System (CLOS) code (Steel [1990]). The CLOS code and a set of library functions support the distributed operation of interactively shared data objects.

This research has been motivated and focused by previous work on design methods and tools to support design of computer-based systems in a UCLA project called SARA (System ARchitect's Apprentice) (Estrin et al. [1986]). The challenge to the effort described here has been to create a collaborative design environment which we call coSARA. OREL is a foundation for coSARA

4.1.1. Object Oriented Programming Systems.

Object-oriented systems have their roots in programming languages such as Simula (Dahl & Nygaard [1966]) and Smalltalk (Goldberg &

Robson [1984]). Lochovsky [1989] contains a set of articles that review the state of the art in object-oriented systems.

The most basic concept is that of *object*. Although there is not a generally agreed upon definition of object, we provide here a working definition for the purposes of this paper. An object is an entity that encapsulates *state* and *behavior*. Every object has the capability of storing data, which define the state of the object. For the purpose of data storage, objects have *slots*, each of which can store one data item. The behavior of an object defines the ways in which the object's state can be changed or the possible questions that can be answered about the state of the object.

Objects are grouped in *classes* that denote sets of similar objects, for example, the class "rectangle" denotes all objects whose state and behavior correspond to the common notion of rectangle. Similarly we can have classes stack, queue, hash-table.

An object-based computation proceeds by *messages* sent from one object to another. Message sending is a form of procedure invocation. When the behavior of an object (requester object) requires that another object (target object) be interrogated about its state (for example, "is that window visible on the screen?") or that another object's state be changed (for example, "convert that window into an icon"), a message is sent from the requester object to the target object. The target object will perform a procedure, called a *method*, which is selected according to the name of the message which denotes a function and a set of arguments that are the contents of the message.

A set of related messages defines a *protocol.* The same protocol could be implemented in more than one way. For example a protocol to manage pictures on the screen could be composed of the messages: move, draw, erase, highlight and reshape. One implementation of this protocol to manage pictures on the screen could exist for the X Window System (Scheiffler & Gettys [1986]) while another could be made for the GKS graphic package. The software that is built using this protocol could then run indifferently under both the X Window System and GKS.

The concept of protocol leads naturally to the concept of *polymorphism*. We define polymorphism as the ability of several classes of objects to respond to the same protocol. For example, rectangle and circle could be classes that respond to the protocol to manage pictures on the screen.

Classes and protocols contribute to modularity in the construction of programs. Another basic concept, *inheritance,* contributes largely to re-usability of modules.

Inheritance allows the definition of classes of objects that are almost like objects of another class. Suppose that class A inherits from class B. Then objects of A have all the properties that objects of B have, plus some additional properties that can be specified for A. We say that A is a *subclass* of B and that B is a *superclass* of A.

Several models of inheritance have been proposed and implemented. *Simple inheritance* exists when each class is allowed to have only one direct superclass (Dahl & Nygaard [1966]; Goldberg & Robson [1984]). *Multiple inheritance* occurs when each class may have more than one direct superclass (Stefik & Bobrow [1983]; Bobrow et al. [1988]). In actor languages (Hewitt [1977]; Lieberman [1981]) a form of inheritance called *delegation* is used, in which an object may delegate any other object to handle a received request. This is a very general form of inheritance, that seems too unstructured to be of practical use.

For a long time, object-orientation has been confined to programming languages. Now there is a strong trend to bring database technology and object orientation together to a point of convergence where one of the essential ingredients is the persistence of objects over time.(Lochovsky [1987], editorial note). Several experimental systems and commercial products have been developed and are currently being developed, leaving many research questions open in this field (Nierstrasz [1989.]). We can point to Gemstone from Servio Logic, which is based on Smalltalk (Purdy, Schuchardt & Maier [1987]), IRIS from Hewlett-Packard Labs (Derret [1985]; Fishman [1987]) and Orion, an effort carried out at MCC (Banerjee et al. [1987]).

4.1.2. Computer Supported Cooperative Work (CSCW)

Exploration of computer-supported cooperative work (Greif [1988]) and investigation of *groupware* technologies (Groupware Technology Workshop [1989]) have accelerated during recent years. This research explores the extent to which computer systems can support teamwork by small groups involved in the design and realization of complex systems. Currently we are all overwhelmed by the dearth of support for reasoning and arguing about the validity of design models and design decisions *before* they display themselves in unpredicted behavior of working systems. Our attention is on groups such as: engineers collaborating on a system design; customers, users and developers seeking agreement on system requirements; and other groups which may be able to use the computer support to expose and reconcile differences in viewpoint. Toward those goals, this research has created a laboratory, containing software and hardware systems that encourage fundamental and systematic experimentation. Some technical problems which we have been investigating in order to create an effective collaborative design laboratory are:

- How to enable *interactive sharing* of design objects by a small group of designers, such that all collaborators have up-to-date knowledge about changes taking place.

- How to integrate new methods and tools which can be used by collaborators without excessive artifact.

- How to manage display which is common to all collaborators.

- How to capture the essence of face-to-face discussions or remote computer-based teleconference sessions.

- How to expose feasible solution spaces when there are conflicts in the goals of the collaborators?

The subject of this paper is the first topic listed above: enabling interactive sharing of design objects by a small group of designers having a common view of the design process. That common view is established through agreement on a sharable design paradigm. Interactive sharing, in turn, requires supported communication among designers and supported coordination to help the group progress toward mutually agreed upon goals. Design objects include design representations, design history, and design tools.

4.1.3. Contributions of OREL

The main contributions of our work on OREL are:

- A *distributed system* in support of *interactive sharing of data* objects with *persistent storage.*

- An object-oriented *graphic language for modeling data,* that includes: classes and multiple inheritance, recursive composition of objects, relations as first class objects and integrity constraints on relations.

- A *programming interface* that allows manipulation of distributed, interactively shared data objects..

We define the concept of interactive sharing of data objects by the following rules:

- The *existence of all objects must be knowable* by all of the collaborating designers at all times.
- A design object is *modifiable by only one designer at a time;* it can be used, but not modified, by more than one designer at a time.
- Changes, made to an object that is shared by two or more designers, can be seen by other designers as they are made, *unless the designer who is modifying the object has chosen to restrict access until changes have been completed.*

The classical notion of object defined in the previous subsection, is extended to make possible the implementation of these features. The extensions are:

- objects are accessed from many sites simultaneously,
- objects may contain site-dependent data,
- objects have unique, site-independent identifiers,

- objects may have symbolic names, which are not necessarily unique,
- objects are used within long transactions and are subject to locking.

There are current research efforts aiming at efficient management of shared, persistent objects. Decouchant [1989], focuses on extending Smalltalk with these features, Moss [1989] is working to smoothly integrate database and programming language concepts. None of the current research, however, has reported interactive sharing of data objects. We believe this to be a unique characteristic of our work.

Other researchers have found that the combination of object-oriented techniques and relational database concepts constitutes a powerful data modeling methodology. Chen [1976] uses entities and relations as modeling primitives. Entities denote sets of data objects, similar to classes in object-oriented systems. Relations do not have an explicit correspondence and constitute an enrichment, expressing more information than can be done with class definitions. Landis [1988] proposes a number of augmentations to Chen's entity-relationship model for modeling complex design data. Landis' augmentations were targeted at the modeling and implementation of SARA/IDEAS (Landis [1988]; Worley [1986]) and are discussed in more detail in Section 4.6.4 on related work.

Our data modeling language makes use of relations as a way to increase expressiveness and to enable management of constraints. Unlike previous work, OREL relations are first class objects. Moreover, recursive composition provides an effective way to manage complexity of design data.

We have built a prototype of coSARA using an implementation of OREL based on CLOS. The existence and behavior of that environment is a proof of concept for OREL.

This Chapter is divided in seven sections:

4.1. this introduction,

4.2. the OREL data modeling language,

4.3. description of OREL's distributed operation,

4.4. description of an implementation of OREL based on CLOS,

4.5. demonstration of the expressive power of OREL by applying it to the construction of a hypertext,

4.6. review of related work, and

4.7. Conclusions and directions for future work.

4.2. OREL Data Modeling

In this section we introduce: the essential set of OREL primitives for data modeling, their graphic representations and CommonLisp functionality associated with them. We start by describing the way in which inheritance is defined in OREL. We proceed then to describe the different primitive types of objects that are supported in OREL and their inheritance relations. After establishing an understanding of the primitive types of objects we describe them one by one using examples. We do not attempt to fully define the functionality associated with each OREL primitive here; instead we define the principal functionality. A full description of the functionality associated with OREL primitives can be found in Appendix II (OREL Protocols).

Our examples of OREL primitives are based on modeling data objects which are required for SARA models. A semantic description of all SARA design primitives is found in Appendix I.

4.2.1. Inheritance Network and Primitive Types of Objects

OREL supports multiple inheritance, in which a class may inherit properties from more than one superclass. The subclass relationship is treated as a lattice, in which each class is represented by a node and precedence between classes is represented by arcs. In the graphic representation that we use, superclasses are drawn to the left and subclasses to the right. Arcs

always go from a superclass attached to its right extreme to a subclass attached to its left extreme. Figure 4.1 shows the graphic representation of the basic inheritance network of OREL, that specifies the inheritance relations between the different primitive types of objects.

The inheritance network of figure 4.1 includes six primitive OREL classes: SIMPLE-OBJECT, COMPOSITE-OBJECT, RELATION-OBJECT, PAIR-OBJECT, UNDIRECTED-PAIR-OBJECT and DIRECTED-PAIR-OBJECT. We explain this classes of objects in the following sections. Figures 4.14 and 4.15 contain a brief description of OREL primitives and Figure 4.16 contains a list of the OREL protocols.

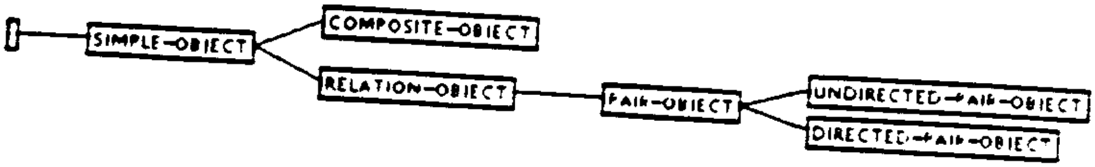


Figure 4.1: Basic inheritance network of OREL primitive classes

4.2.2. Simple Classes and Protocols

Let us begin our discussion of the OREL primitives describing the simple-object class. SARA structural models have three primitives: *modules, sockets and interconnections.* Sockets are objects with no other property than providing connectors for modules. A socket is known inside and outside a module and provides a named place for delivery of either a service provided by the module that contains the socket or a service required from some other module. When a module uses a service provided by some other module, both modules are connected by an interconnection that attaches to the corresponding sockets.

We can model a socket as a *SIMPLE-OBJECT.* A SIMPLE-OBJECT belongs to a *simple class* and has a state represented by a collection of data. The state of a SIMPLE-OBJECT can be altered and queried by sending messages to the object. A set of such messages is called a *protocol,* and is represented graphically as a hexagon. Figure 4.2 displays the representation of the SIMPLE-OBJECT protocol. Each simple class has a set of associated protocols that embody the messages that can be used to operate on the simple class' objects.

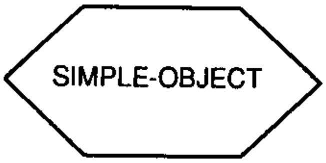


Figure 4.2: SIMPLE-OBJECT protocol

A simple class is drawn as a rectangle with a label that indicates the name of the class. The protocols supported by a simple class are drawn on the border of the class' rectangle. Figure 4.3 shows the class SOCKET.

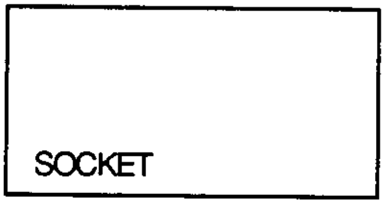


Figure 4.3: The class SOCKET

Figure 4.4 shows the inheritance network of OREL after we have added the simple class SOCKET. We can see in this figure that there is a connection from SIMPLE-OBJECT to SOCKET, that goes left to right. This says that SOCKET is a subclass of SIMPLE-OBJECT and therefore SOCKET objects inherit all the properties of SIMPLE-OBJECTs.

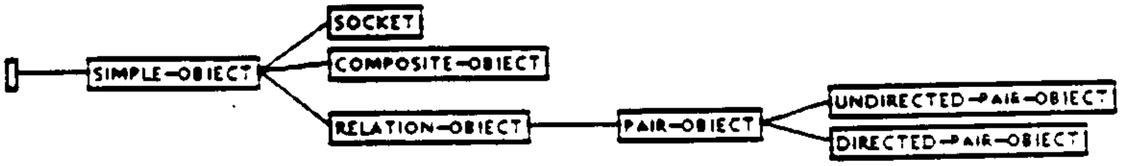


Figure 4.4: Inheritance network after definition of SOCKET

4.2.3. Slots

Each object has a set of *slots* to store state data. Each slot is capable of holding one data item. A data item may be any CommonLisp object or any OREL object. There are two types of slots, *class-allocated* slots and *object-allocated* slots. A class-allocated slot has a common value for all objects of the class. An object-allocated slot has a different value for each object of the class.

Sockets are named. This fact implies that we need a way to store the name of a socket, query a socket for its name and possibly alter the name of the socket. We can model the name of a socket as an instance-allocated slot in the class socket. Figure 4.5 shows the graphic representation of the class socket showing the slot "name".

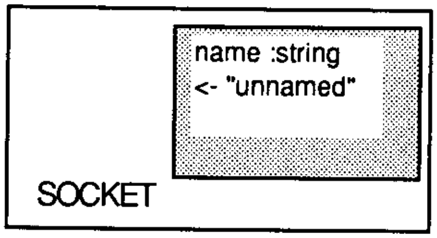


Figure 4.5: Class SOCKET with slot "name"

The graphic representation of slots includes textual information to specify the following properties of slots:

Type: The type of a slot may be any defined class or any defined type of CommonLisp. The type is specified as *:type* following

the slot's name. For example in Figure 4.5 the type of the slot name is "string".

Initial value: The initial value of a slot is specified by ←*value.* For example the slot name of the class socket in Figure 4.5 is specified to have an initial value of "unnamed".

For each slot there is an *initialization argument keyword* that can be used when an object is made. This is a keyword of the form *:slot-name,* where slot-name is the name of the slot. For example there is a keyword :name to optionally initialize the name slot of sockets.

4.2.4. Relations and mapping constraints

A relation is a set of tuples; each tuple represents a relation among a set of objects. The participation of objects in a relation is defined by *mappings,* where a relation has one mapping for each one of the classes it relates. Several types of constraints can be placed on the way in which classes participate in relations. We call these *mapping constraints.* Mapping constraints include: *cardinality* constraints, *completeness* constraints, *direction* constraints and *order* constraints. The precise meaning of mapping constraints will be explained by means of examples below.

There are three types of relation: undirected pairs, directed pairs and general relations. The simplest one is an *UNDIRECTED-PAIR-OBJECT.* An UNDIRECTED-PAIR-OBJECT is a relation in which each tuple is a pair of objects listed in any order. We can use an

UNDIRECTED-PAIR-OBJECT to model interconnections. To represent the class of interconnections we use an *undirected pair class* INTER as shown in Figure 4.6. An UNDIRECTED-PAIR-OBJECT is represented graphically as a multi-segment manhattan-style line.

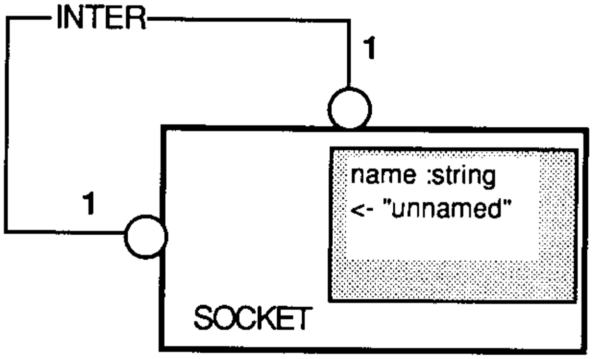


Figure 4.6: The undirected pair class INTER

We have said that an interconnection connects a pair of sockets, and that each socket can participate in only one interconnection outwards from its module and one inwards to its module. The undirected pair class INTER connects the simple class socket to itself. This means that the pair INTER is composed of pairs of sockets. The undirectedness of the pair implies that when two sockets S1 and S2 are related by INTER, it does not matter in which order the sockets are related to each other. The behavior is as if both pairs (S1,S2) and (S2,S1) were in the relation. If S1 and S2 are related by an INTER pair, given S1 we can obtain S2 and given S2 we can obtain S1. This would not be the case if the pair were directed.

Let us turn attention to the constraints that have been placed on the INTER pair. We have placed cardinality mapping constraints **1** on both ends of the arc that represents the undirected pair INTER. A cardinality constraint of **1** means that each socket may participate in only one pair of the relation. Another constraint (or lack thereof) is that we have specified a completeness constraint "partial" (represented in Figure 4.6 as hollow circles). A "partial" constraint means that not all sockets need to participate in an INTER relation. If a "total" completeness constraint had been used instead of "partial", all sockets would have to participate in some interconnection. We will find examples of this type of constraint when we discuss the modeling of the relation that we use to represent the mapping of control nodes to data processors in the GMB.

So far we have stated that each socket may participate in only one INTER pair. Examination of Figure 4.7 reveals that a socket may indeed participate in more than one SARA interconnection. This apparent conflict with the model that we show in Figure 4.6 is not a problem because interconnections are classified in several sets, each set corresponding to one module. Thus there is an instance of the simple class INTER in the module UNIVERSE and a different instance in the module System. Each one of these two INTER objects group a different set of interconnections. Hence pairs of sockets can be related by INTER objects that belong to different modules.

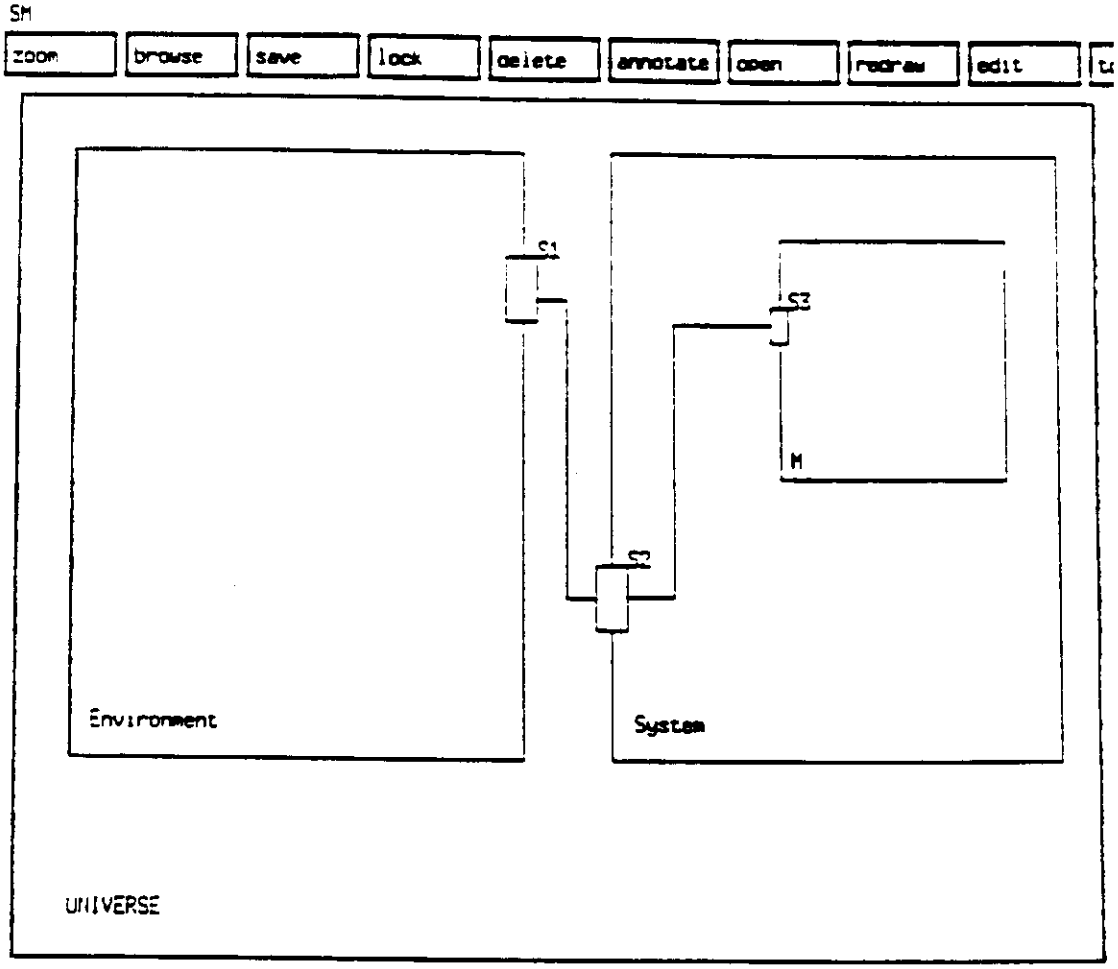


Figure 4.7: Example SM model

A *DIRECTED-PAIR-OBJECT* is similar to an UNDIRECTED-PAIR-OBJECT, except that the order in which pairs of objects are related is important. We can use as an example the modeling of the relation between control arcs and control nodes in the Graph Model of Behavior (GMB)[1]. Control arcs go from one set of control nodes to another set of control nodes. The source nodes are said to be the *tail set* of the control arc and the destination nodes are said to be the *head set* of the control arc. For example in Figure 4.8 {**n1**, **n2**} is the tail set of control arc **a**, and {**n3**, **n4**, **n5**} is the head set of that same control arc **a**.

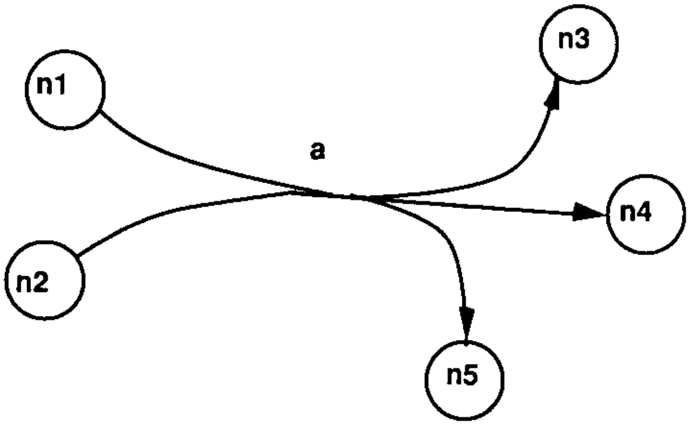


Figure 4.8: A control graph

We can model this relation between control arcs and control nodes using two directed pair classes, TAIL-SET and HEAD-SET, as shown in Figure 4.9 where CONTROL-NODE and CONTROL-ARC are

[1] See Appendix I

simple classes. The direction specifies that given a control arc we can find control nodes in its head-set and control nodes in its tail set. We have placed the cardinality mapping constraints **1** in the side of CONTROL-ARC and **N** in the side of the CONTROL-NODE to represent the fact that one control arc may have zero or more tail control nodes and zero or more head control nodes. In this case given a control arc we can retrieve its head set and its tail set. However, unlike the case of undirected pairs, given a control node we cannot retrieve a control arc because the relation is directed from CONTROL-ARC to CONTROL-NODE.

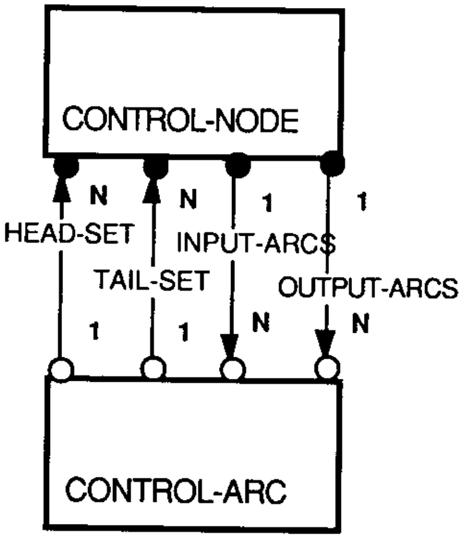


Figure 4.9: OREL model of control nodes and control arcs

In this example we have placed completeness constraints[2] on the pairs: HEAD-SET, TAIL-SET, INPUT-ARCS and OUTPUT-ARCS, meaning:

1. Every control node must have input arcs and output arcs.
2. Every control node must be either in the head set of some arc or in the tail set of some arc.
3. A control arc may have an empty head set and an empty tail set.

In the next paragraphs we will discuss the class of general RELATION-OBJECTs, showing how we can use relation objects to model the association between a GMB and the module that contains it.

We explained in our discussion of the SARA primitives that control arcs and data arcs may be connected to sockets; however in that case the sequence in which they are connected is significant. Figure 4.10 shows an example in which two modules are interconnected through sockets **S1** and **S2**. Assume that the arrangement of control arcs and data arcs is such that:

- Control arcs **a1**, **a2**, and **a3** of module **M1** correspond to control arcs **a4**, **a5** and **a6** in module **M2**, and

[2] See Figure 4.15.

- data arcs **dl** and **d2** of module **Ml** correspond to data arcs **d3** and **d4** of module **M2**.

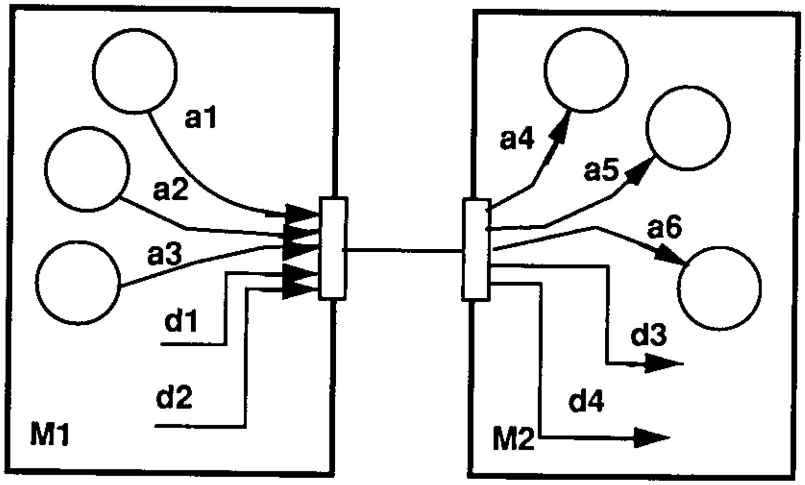


Figure 4.10: Correspondence of control arcs and data arcs across module boundaries

We can model this association between control arcs, data arcs and sockets using a a three-way *RELATION-OBJECT* as is shown in Figure 4.11. RELATION-OBJECTs are members of a *relation class* and are graphically represented by a diamond. RELATION-OBJECTs inherit from SIMPLE-OBJECT as shown in Figure 4.1. A RELATION-OBJECT is a collection of tuples, each of which has one component for each one of the related classes. In this case a relation-object GMB-SOCKET will be a collection of triples of objects of classes CONTROL-ARC, DATA-ARC, and SOCKET.

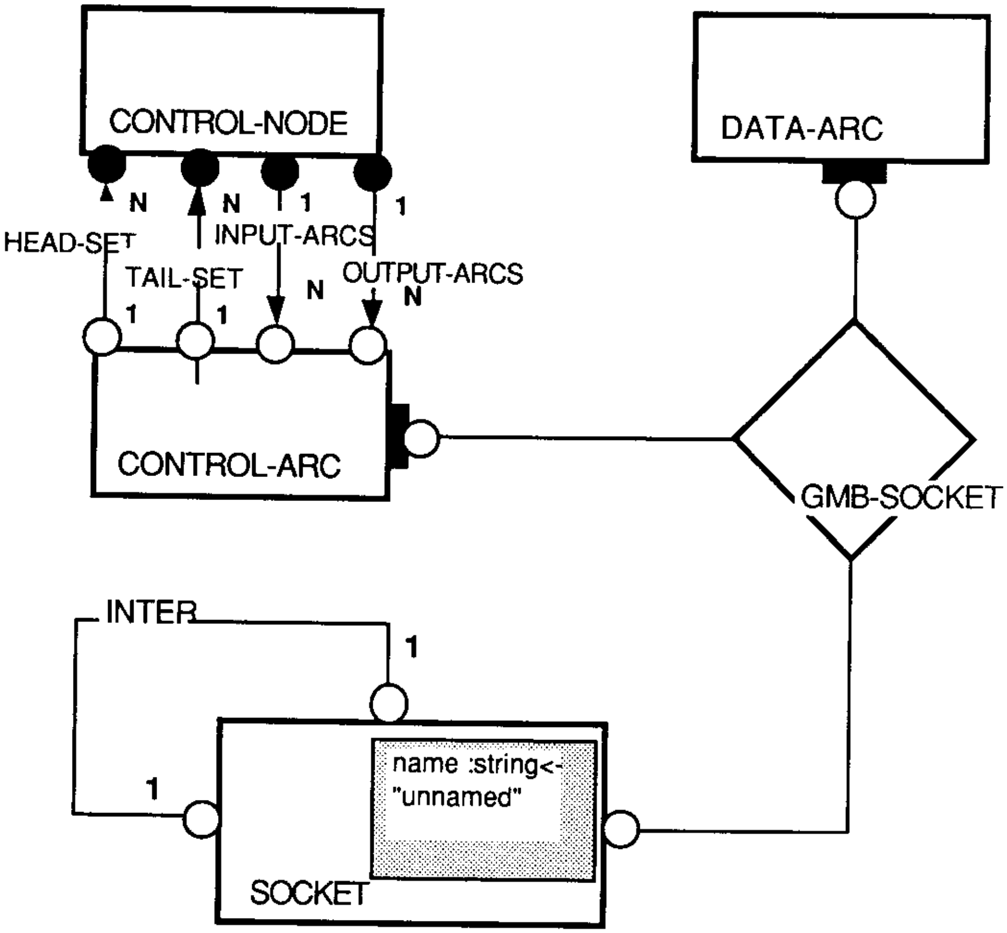


Figure 4.11: GMB-SOCKET relation

In this example we have introduced an order mapping constraint (black rectangle). This constraint, applied in Figure 4.11 to CONTROL-ARC and DATA-ARC, specifies that control arcs and data arcs are ordered in the relation. Then, given that a set of control nodes related to a specific socket is retrieved, it is guaranteed that the control nodes will be ordered.

All relations are subclasses of SIMPLE-OBJECT. Therefore they have the same inheritance properties as simple classes. In the case

of relations, inheritance is interpreted differently. It is not the RELATION-OBJECT which inherits properties from the superclasses. Instead the tuple objects inherit properties from the superclasses of the relation class. In this way we can associate properties with a group of objects that are related by some relation object. For example in Figure 4.11 we show pairs INPUT-ARCS and OUTPUT-ARCS relating the simple classes CONTROL-ARC and CONTROL-NODE. In this case we can specify that the pair relations INPUT-ARCS and OUTPUT-ARCS inherit from a LOGIC-FORMULA class to represent the input and output logic of a control node. We do not show the class LOGIC-FORMULA in the figure for simplicity. The fact that both CONTROL-ARC and CONTROL-NODE are subclasses of LOGIC-FORMULA is expressed in the inheritance network.

4.2.5. Composite objects

COMPOSITE-OBJECTs aggregate a number of other objects. For example SARA modules are composite objects that aggregate sockets, interconnections, a GMB and possibly other modules.

In general a COMPOSITE-OBJECT is composed of any number of SIMPLE-OBJECTs, COMPOSITE-OBJECTs and RELATION-OBJECTs. We define a *composite class* by enumeration of the classes of objects that are aggregated in a COMPOSITE-OBJECT. The graphic representation of a composite class is a rectangle with

rounded corners that contains the representation of all its components (Figure 4.14)

A composite may contain objects of its own class. In this case we say that the composite is recursive. The graphic representation of a recursive COMPOSITE-OBJECT is a shaded rectangle with rounded corners. For example, we need to use recursive composition to model module objects. Figure 4.12 shows a model of SARA objects, which includes the recursive composite class MODULE.

As specified in Appendix I, a module contains other modules, sockets, interconnections and a GMB. A module is restricted to either contain any number of other modules or to contain a single GMB.

In previous examples in this section, which led to discussion of Figures 4.3, 4.5, 4.6, 4.9, 4.10 and 4.11 we have built the model of Figure 4.12 step by step and demonstrated how this model represents the SARA objects described in Appendix I, except for modules which we discuss below.

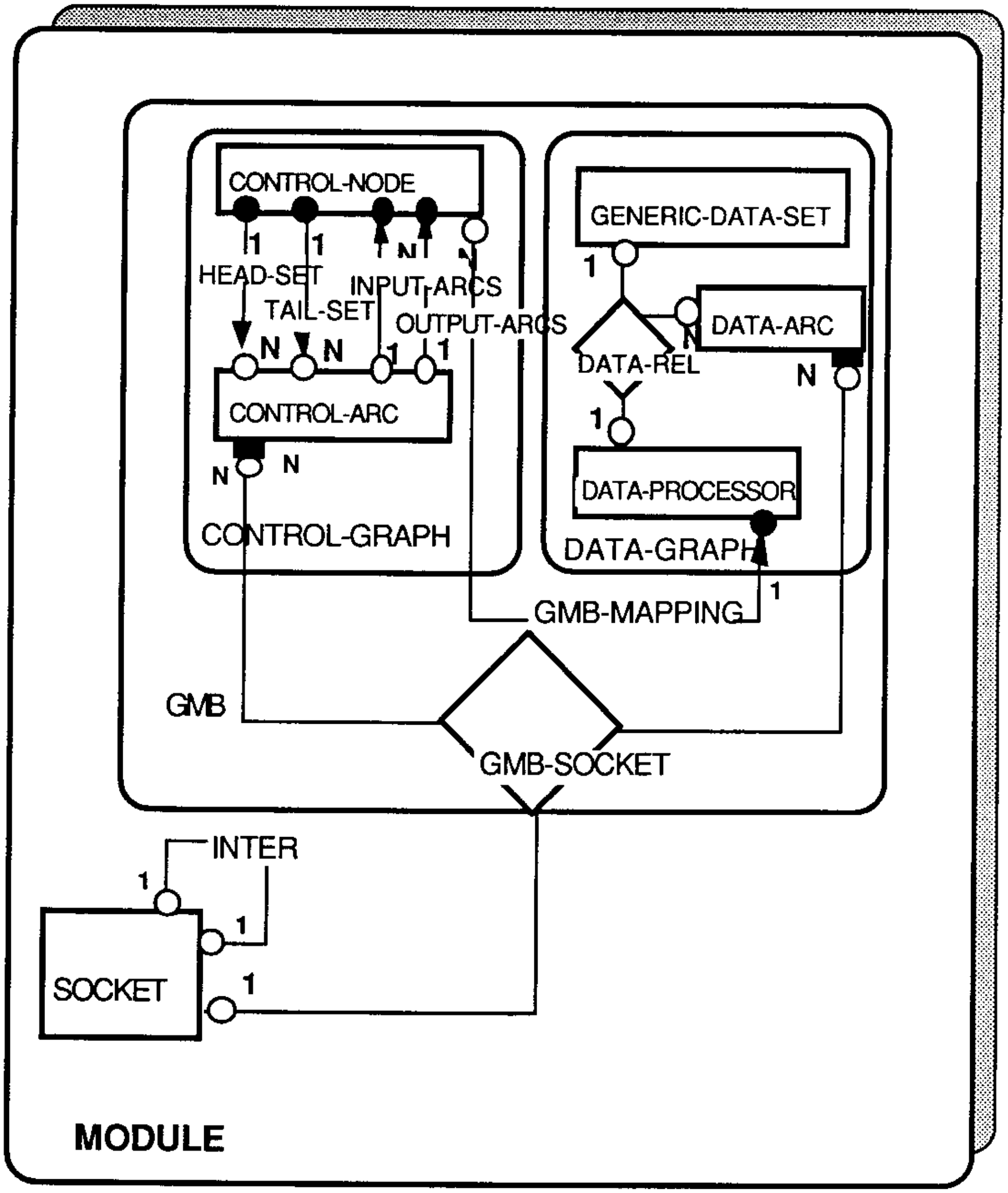


Figure 4.12: OREL model of SARA objects

The model of the composite class MODULE shown in Figure 4.12 specifies SOCKET as one of the MODULE object components. This specifies that any number of SOCKET objects may compose a single MODULE object. There is no provision for constraining the number of objects of a given class that can be part of a composite-object. For example in the case of the GMB component of MODULE we know in advance that there will always be at most one GMB object. However we cannot express this fact. The same is valid for the undirected pair class INTER.

Composite objects are also simple objects as can be seen in the inheritance diagram shown in Figure 4.4. Therefore all the properties that we have discussed for simple objects also apply to composite objects.

Figure 4.13 shows extension of the inheritance network after definition of MODULE.

Figure 4.14 and 4.15 summarize the OREL primitive classes and the relation mappings described before. Figure 4.16 lists the OREL protocols, which are documented in detail in Appendix II. There are three protocols: SIMPLE-OBJECT, COMPOSITE-OBJECT and RELATION-OBJECT protocols. The Figure 4.16 enumerates the messages that compose each of these protocols.

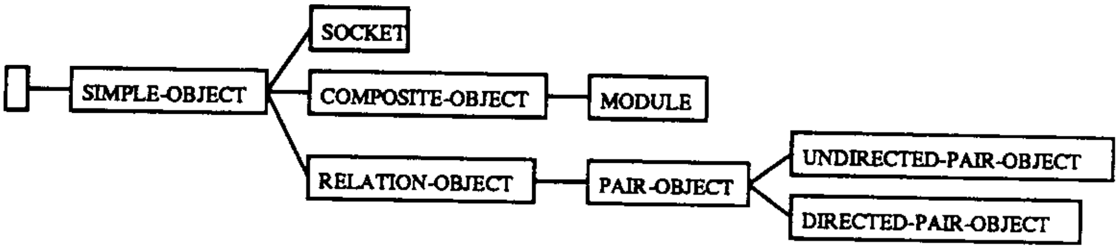


Figure 4.13: Inheritance network after definition of module

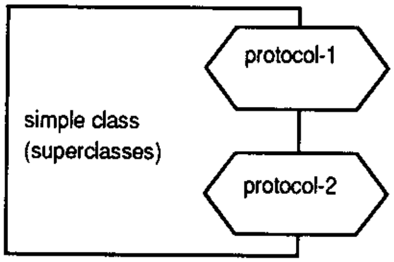
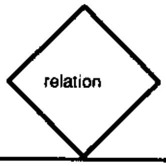
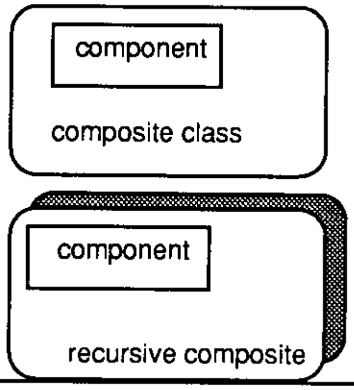
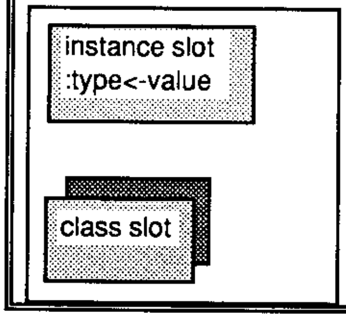
	**Simple class[rectangle]:** Encapsulates the state and behavior of its objects. Each supported **protocol [hexagon]** is specified for the class. A protocol denotes a set of methods that can be applied to the instances of the class. Basic functionality of all OREL objects includes functionality to make and initialize objects.
	**Relation[diamond]:** Relations support a protocol that includes functionality to relate and unrelate objects, select a subset of the tuples in the relation, and apply functions to selected tuples.
 component composite class component recursive composite	**Composite class[shaded round-corners rectangle]:** A composite class aggregates other classes and relations. Each component is a set of objects of the corresponding class or relation. A composite class may be **recursive [shaded rectangle].** In this case the class has components of its own class. Composites support functionality to create composites, add objects to composites, remove objects from composites, and test membership in a composite.
	**Slot[shaded rectangle]:** Is a data attribute that determines the state of an object. Simple classes, composites and relations have slots as holders of state data. Slots have the following attributes: type, initial value and class vs. instance allocation.

Figure 4.14 OREL Primitive Classes

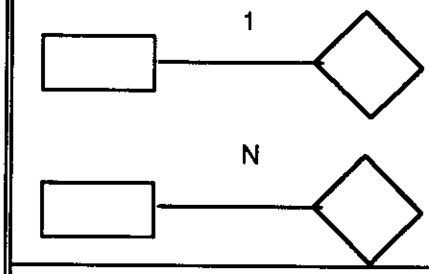
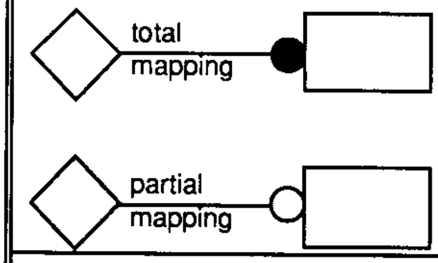
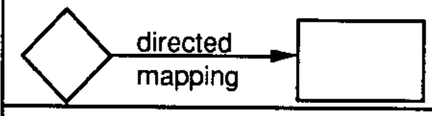
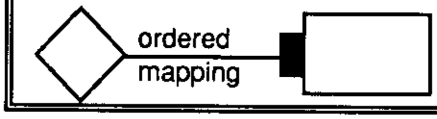
Relation Mapping[arc]: Defines the way in which a class participates in a relation. There are four mapping types that can be combined with each other.	
	**Cardinality:** *N* specifies that many tuples of the relation may contain an instance of the associated class and *1* specifies that at most one tuple may contain an instance of the associated class.
	**Completeness:** *total* when all the instances of the participating class must be in the relation, *partial* otherwise.
	**Direction:** establishes a direction for the relation.
	**Order:** tuples are lexicographically ordered by the value of its components.

Figure 4.15: Relation Mappings

Protocol	Messages
Simple-object	• initialize-instance • print-object • describe-object • class-of • class-name • <slot-name> • make-<name>
Composite-object	• components-of • comp-<component name> • comp-parent • comp-ancestors • compositep • component-of-p • descendant-p • add-component • delete-component • component-assoc • map-composite • make-<name>
Relation-object	• relate • unrelate • find-objects • map-relation • make-<name>

Figure 4.16: OREL Protocols

The data model of Figure 4.12 makes available OREL protocols listed in Figure 4.16 for programming routines that perform operations using the SARA objects. A brief example of how OREL protocols are used is presented next. Only a few of the messages listed in Figure 4.16 are used in this example. Words that are used in Figure 4.17 are

typeset in **boldface** in the text and in a plain typeface in Figure 4.1. OREL protocol message names are typeset underlined both in the text and in the Figure.

```
(defun traverse-interconnections (s i)
    (let ((next (find-objects i s)))
      (if next
          (let* ((s-parent (comp-parent s))
                 (next-parent (comp-parent next)))
                 (next-i    (if (descendantp s-parent next-parent)
                                 (first (comp-interconnections next-parent))
                                 (first (comp-interconnections
                                          (comp-parent next-parent))))))
             (traverse-interconnections next next-i))
          s)))
```

Figure 4.17: Example usage of OREL protocols

Figure 4.17 shows a function for traversing a path of interconnected sockets as far as possible. The function **traverse-interconnections** takes two arguments: a socket and an interconnection relation. The interconnection relation contains a tuple that represents the interconnection leading from the starting socket to another socket. The function **traverse-interconnections** "hops" from socket to socket, using interconnections, until a socket is reached that has no further interconnections.

Let us analyze the code of Figure 4.17 in greater detail. The first thing to do is to find the socket to which the starting socket s is

connected through the relation i. The OREL message **find-objects**, which belongs to the RELATION-OBJECT protocol has a specialized method for handling pairs. This method receives two arguments: a pair and an object. The method returns either a single object or a list of objects, depending on the cardinality constraint of the mapping by which the class of the retrieved object participates in the pair. If the cardinality constraint is **1** **find-objects** returns a single object, otherwise if the cardinality constraint is **N** it returns the list of all the objects associated to the argument object using the argument pair. The call to **find-objects** in line 2 returns the socket connected to **s** using the interconnection **i**.

The message **comp-parent** that belongs to the COMPOSITE-OBJECT protocol returns the composite of which its argument is a component. If the argument to **comp-parent** is not a component of any composite **nil** is returned. The call to **comp-parent** in line 4 returns the module that contains socket **s.**

The message **descendant-p** returns **t** either if its second argument is a component of its first argument or if parent composite of its second argument is a descendant of its first argument. It returns **nil** otherwise. The call to **descendant-p** in line 6 returns **t** if module **next-parent** is contained in module module **s-parent**.

For each composite, a set of methods are defined to access its components. The message **comp-interconnections** will return the list of the interconnections of its argument. We know that a

module has only one set of interconnections, therefore the list returned by **comp-interconnections** will always be of length one, hence we must use **first** to obtain the pair that is needed in lines 7 and 8.

4.3. Distributed operation of OREL objects

OREL objects are intended to be used in a distributed environment in which several users, working at different workstations and connected through a communication network, interactively share a set of objects. This section describes the semantic model used to conceptualize the distributed operation of OREL objects and the mechanisms that a tool designer can use to operate on shared objects.

4.3.1. The Object world model.

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.

Coordination among users is achieved by using transactions. 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 ended and committed. Of course, 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.

While executing an operation within a transaction, a user acquires locks on objects at any time and in any arbitrary sequence. All locks are released atomically when the transaction is committed to prevent cascading roll-backs. The locking scheme is a non-strict 2-phase locking protocol (Bernstein, Hadzilacos & Goodman [1987]), that may lead to deadlocks. We rely on the fact that these are long interactive transactions, controlled by human users, and permitting 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 could be carried out easily because designer 1 and designer 2 were working face-to-face and therefore it was easy to talk.

The following 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, 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.*

4.3.2. OREL Objects and Broadcast Methods

Broadcast methods are used to operate on OREL objects that are interactively shared by several users. This section describes the semantics of broadcast methods after clarifying the problem posed by shared object-oriented computation.

We explained in the section on object-oriented systems background, how an object-oriented computation proceeds by messages sent from one object to another. The response of an object to a message is encapsulated in a method. When the message is received, the corresponding method is invoked.

Let us consider now 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, following the rules for sharing as defined in the object world. The most straightforward way to achieve sharing of objects, according to the rules stated for the object world, 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 involve the object. The smallest possible change to the state of an object is to change the value stored in one of the object's slots. Therefore, this approach would consist of defining transactions to change the slot values of an object.

There are some problems with the approach just described:

- Transactions would require excessively large amounts of CPU cycles and network bandwidth. Assuming that these operations were implemented in a distributed system, 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).[3] Each access to an object's slot would carry all the transaction processing overhead.

[3] This cost is cuadratic when there are failures.

- Changes would be distracting when they propagate to each user's replica in fine-grained fragments,.

To alleviate these problems we have adopted a different technique in which we define a class of methods, called 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.

For example, consider the class of shared rectangles. One operation that can be done on rectangles is to reshape them, changing one or more of: height, width and position. The user is given two operations for reshaping a rectangle:

try-shape: takes four arguments, rectangle, height, width and position. Rectangle is temporarily displayed in a shape specified by the remaining three arguments. Height and width can be either an integer number or a null value. When a null value is received as argument, the rectangle's original values are used; otherwise the integer number is used for the corresponding dimension. The position argument can be either a pair of integers denoting the coordinates of the upper left corner of the rectangle or a null value. A non-null value changes the position of the rectangle. **try-shape** changes the shape of the rectangle only temporarily and displays the

rectangle in its temporary shape. When the next 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 the user who is reshaping the object tries would be seen by others. However, it is possible that the tool designer decides that collaborating users do not need to see the intermediate shapes tried by the user who is reshaping the rectangle. The tool designer would therefore 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.

We describe now the precise semantics of broadcast methods. A broadcast method takes an arbitrary number of arguments as input, which may be either OREL objects or any CommonLisp object, which is not an OREL object (for example the integer 5).. In the discussion that follows, let $\mathbf{O_1, ... O_n}$ be the OREL objects that are given to the broadcast method as argument and $\mathbf{W(O_i)}$ be the set of workstations that have replicas of the object $\mathbf{O_i}$. Execution of a broadcast method is done in three steps:

1. Copies of all objects $\mathbf{O_1,..., O_n}$ are dispatched to interested workstations. We define interested workstations to be the union of the sets $\mathbf{W(O_i)}$ **for** $\mathbf{i=1,...,n}$. This step is necessary to avoid the problem that arises when a broadcast method, executed in a remote workstation tries to use an object for which a replica does not exist in that remote workstation.

2. The method is executed locally in the workstation that invoked the broadcast method.

3. A request to execute the method in the interested workstations is broadcast.

4.3.3. Concurrency control

Sharing objects poses a problem of concurrency control. It is necessary that one user does not destroy other users' work. In the example given in the previous sections, 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. It

is convenient that a concurrency control mechanism be as flexible as possible in allowing the tool designer to make this sort of decision from a methodological point of view.

We share objects in this system to collaborate on the process of designing a computer system. Such a collaborative process requires that one user's actions be quickly propagated to other users. This is, group members should be able to observe work done by any other group member in real-time, as it is done, without the need to leave their workstations and look over the working user's shoulder.

There are several possible concurrency control methods for collaborative work.

Optimistic concurrency control: does not enforce any form of concurrency control in hopes that conflicts will not arise. These systems offer a good response because of the lack of concurrency control overhead, but often need to resort to personal interaction among members of a group of collaborators to resolve conflicts that do arise. Strong assumptions are necessary about shortness of the time frame in which one users actions are seen by the rest of the group and the ability to communicate personally, independently of the shared space in which the group works.

Locking: This is the simplest method of concurrency control. Objects are locked before their state is changed. Deadlock

avoidance can be handled by standard techniques, such as pre-locking all objects to be modified and locking objects in a predefined order (Bernstein, Goodman & Hadzilacos [1987]). Perhaps the biggest problem with locking is the grain size that locks should have, and how locks are acquired. A large grain size of locks will prevent parallelism in the group activities. A small grain size will allow greater parallelism in the group's work, but will also introduce a larger overhead. Locks can be acquired explicitly by the user when an object needs to be modified. If the grain size of lock is rather large this is not a problem. If the grain size is too small the user may be overwhelmed by the process of acquiring and releasing locks. Therefore for finer grain locks it becomes necessary to design mechanisms that will automatically acquire locks on behalf of the user. Let us consider the rectangle example that we described earlier. The operation **set-shape** could acquire a lock and release it after the shape of the rectangle has been redefined.

Transactions: It is not possible to consider transactions as valid mechanisms for concurrency control in a system in which objects are interactively shared according to the rules established for the object world. The problem with conventional transactions, used in database systems, is that transactions in a database system are designed to give the user the illusion that the database system is a single user

system, hiding the intermediate states of objects being modified by other transactions. This property may conflict with the rules of the object world that say that work done on one object should be visible to all users, in real-time. In the simplest case, the most elementary changes to the state of an object would be transactions. In our terms, changing the value of each slot would be a transaction. The cost incurred in managing such small transactions could be unaffordable. There are also methodological considerations regarding the grain size of transactions, analog to those discussed for locking in the previous paragraph.

Floor passing: Another simple system of concurrency control that some conferencing systems use is that of floor passing, in which one user at a time can act on the system. Floor passing from one user to another may be controlled by software or by social agreement. The great disadvantage of floor passing is that there is no parallelism possible in the group's actions. Only one group member can work at any time.

The concurrency control mechanisms that we have decided to use in OREL are transactions and hierarchical, implied locks on composite objects.

Let us first describe the transaction model that we use. OREL transactions are long, interactive transactions, controlled by the

user. A user may have several transactions in existence at any time but only one can be active. The intent is to *provide a way in which the user can organize a set of on-going tasks*. The operations that we can do on transactions are:

start: creates a new transaction for the user who issues the request and enters the transaction in a transaction directory pertaining to the object world. This is necessary to provide users with enough information for social discussion and negotiation of potential conflicts.

add a lock: obtains a new lock and adds it to the transaction list of locks. Locks are acquired in any arbitrary order using a non-strict 2-phase locking protocol. This protocol guarantees serializability, but does not prevent deadlock. We leave the resolution of deadlock conflicts to social interaction between users, based on the information provided by the system according to the rules of the object world. Mainly, the existence of all objects is known to all collaborating users, and any user may know who is using an object at any time.

commit: this operation ends the transaction. All the locks owned by the transaction on behalf of its user are released, the state of the objects that were written is committed in the database, the transaction is removed from the system's directories and the transaction is ended.

commit-restart: enables users to save work at any time without giving up the transaction. The only effect of this operation is to commit the state of written objects in the database. The locks are not released and the transaction is not ended.

abort: the transaction is ended and the state of modified objects is restored to the state they had when the transaction started.

The locks used in transactions are hierarchically implied locks on composite objects. 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)]. In this algorithm we introduce a *write-warning* type of lock. When an object is locked for writing all its ancestors, are given a write-warning lock. When an object has a write-warning lock, it means that one or more of its components have been locked for writing. We define lock compatibilities in such a way that write-warning locks are compatible with all types of locks but write locks. Therefore, we are prevented from acquiring a write lock on an object that already has a write-warning lock.

One requirement is that work done on any object must be visible to all users at any time, unless that user who is changing the state of the object chooses to make the changes invisible until they are done. This requirement leads us to define a type of lock that enables us to read an object while it is being written by some transaction. We call this

type of lock *dirty-read* locks. A dirty read lock enables one user to see changes to an object as they are made in real-time. It also warns the user that the state of the object, as it is seen, 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 of failures 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 may proceed to base work on such an unreliable state, destroying the strictness of the transaction model. Such user mistakes could lead to manual 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. A lock compatibility matrix is shown in Figure 4.17.

	write	read	dirty-read	write-warning
write	no	no	yes	no
read	no	yes	yes	no
dirty-read	yes	yes	yes	yes
write-warning	no	no	yes	yes

Table 4.1 Lock compatibility matrix

Following the general style set by broadcast methods, 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. The primitive operations for handling locks are:

lock-object: takes two arguments, the object to be locked and the type of the lock, which is one of write, read, dirty-read and write-warning. If a lock is requested, whose type produces a conflict according to the lock compatibility matrix of Table 4.1, a condition is raised that must be handled by a tool.

with-lock: denotes a body of text within which a set of locking conditions must be observed. The locking conditions that must be observed are specified as a list of pairs <**object**, **lock-type**>, where **object** must hold a lock of type **lock-type**. If some object does not satisfy the locking condition specified, a condition is raised that must be handled by a tool.

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.

4.4. A CLOS based implementation of OREL

This section describes the process to build and modify an OREL data model. Figure 4.18 displays a SARA dataflow model of that process (see SARA GMB primitives in Appendix I). The dataset (rectangle) labeled OREL Data Model, contains an object representing the data model. This data model representation is created and changed using the OREL Graphic Editor data processor, which is a graphic editor for the OREL graphic language. The OREL Data Model is input to the CLOS Translator and to the Schema Translator. The CLOS Translator produces CommonLisp/CLOS code, that contains definitions of classes and associated functionality to implement the stored OREL Data Model. The Schema Translator produces database

schema that allow the storage and retrieval of OREL objects to and from secondary storage devices using a data base management system. The data processor (hexagon) labeled OREL Graphic Editor is used to create or change the OREL Data Model.

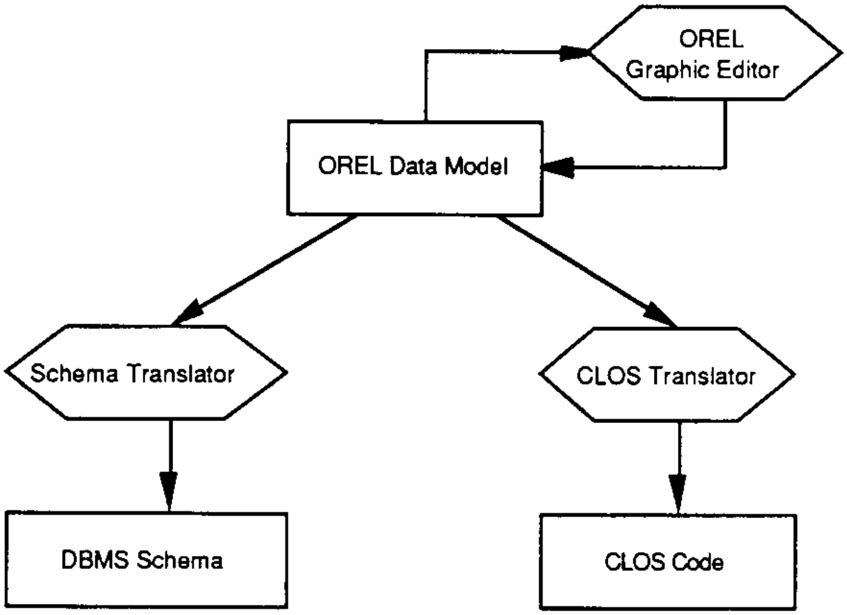


Figure 4.18: Dataflow model of the process to build or alter a data model

The code generated by the CLOS Translator and predefined functionality contained in a library (not shown in the figure), are an implementation of the OREL protocols discussed earlier. Such protocols constitute the programming interface used to manipulate data that conforms to an OREL data model. Programmers make use

of OREL protocols to implement tools that handle data objects according to the corresponding OREL data model.

Each time that an OREL data model is created from scratch the full process described in Figure 4.18 must be followed to obtain Common-Lisp code for manipulating OREL objects or to store OREL objects in a database. When the data model is changed, the process must also be completed to obtain up-to-date code that reflects the current state of the model.

Changes to the data model may impact existing, user-written code in varying degrees of severity. Some changes to the data model, in particular those that add extra features may have no impact at all. Other changes, such as removal of some classes from the data model may have a significant impact and require reworking of code to conform to the changes. Version control helps to manage such problems.

4.4.1. CLOS schema

In this section we describe how the different OREL primitives are implemented in CLOS. OREL simple classes, composite classes and relation classes are all represented using CLOS standard classes and use the inheritance network defined in Figure 4.1.

There is a problem due to the loose coupling of classes and methods in CLOS. In CLOS, classes and methods are independent entities

and there is no encapsulation of data in a set of classes and related methods. In CLOS all slots of every class are visible to all methods.

CLOS also lacks a strong notion of protocol in the sense used by OREL. There is no entity in CLOS that will group a set of methods and allow treating them collectively as protocol. The notion of protocol, is left to the mind of the CLOS programmer.

In our implementation we have chosen to follow the CLOS flavor for simplicity of prototyping. We have not implemented enforcement of OREL encapsulation rules, although they are available.

The following rules determine how CLOS schema are generated from a given OREL data model:

Simple classes: For each slot in the OREL class there is a corresponding slot in the CLOS class. Besides, when the OREL class is part of a composite, the CLOS class has a slot that points to the corresponding parent composite object. The list of superclasses is determined from the OREL inheritance network. In particular, the SIMPLE-OBJECT class is always a superclass of any simple class.

Composite classes: For each slot of the composite class, there is a corresponding slot in the CLOS class. Besides, when the OREL composite class is a part of a composite, the CLOS class has a slot that points to the parent composite object. For each component of the composite there is a slot in the CLOS

class that stores a list of components. The list of superclasses is determined from the OREL inheritance network . In particular, the COMPOSITE-OBJECT class is always a superclass of any composite class.

Relation classes: For each OREL relation class, two CLOS classes are created. One CLOS class represents the relation object and the other CLOS class represents tuples of the relation. The CLOS class that represents the relation object has one slot for each one of the slots defined for the OREL relation. It also has a slot pointing to its parent composite when it is part of a composite object. The CLOS representation of relation objects stores the relation's tuples as a list in a slot. The only superclass of a CLOS relation class is the class RELATION-OBJECT. The CLOS class that represents tuples has one slot for each class that participates in the relation. The superclasses of a CLOS tuple class are determined from the OREL inheritance network.

Pair classes: Pair classes include both directed and undirected pair objects. Each OREL pair class is represented by two CLOS classes. The pair object has one slot to point to its list of tuples and, when it is part of a composite, a slot to point to its parent composite object. The only superclass of the pair object is the class PAIR-OBJECT. The CLOS tuple class contains no slots, and serves as a place holder for the

superclasses defined for the OREL pair class. The actual pairs are stored as pointers in the related objects themselves rather than in a separate list of tuples as occurs with general relation objects.

Appendix II contains the code generated for the model of SARA objects shown in Figure 4.12.

4.4.2. CLOS implementation of distributed operation

There are several possible approaches to build an implementation of the object world, some of which have been used successfully in other systems. We will discuss in this section the following models: centralized storage, proxy objects (Decouchant [1989]), full replication of objects (Stefik et al. [1987]), and finally replication of objects on demand, which is the method that we have chosen for our implementation.

Centralized storage: In a centralized storage system, all objects reside in a single machine, and all operation on the objects is done by the machine that stores the objects. It is obvious that the single machine where all operation is done will be heavily taxed.

Proxy objects: In a proxy object system, each object is stored in one machine. Not all objects need to be stored in the same machine, one would expect that object storage is evenly distributed on a set of machines but this is not necessarily

true. When a user tries to operate on an object that is stored in a machine different from his/her own, a proxy object is used to represent the remote object in the machine that issued the operation request.

Fully Replicated data: In this model all the machines have an exact copy of all the objects in the system.

Replication on demand: In this model 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.

4.4.2.1. A CLOS-based implementation of replication on demand.

An active collaborative design session has a centralized server and zero or more client sessions. The server provides three services: locking of objects, management of persistent object storage and management of meeting configuration. A client consists of a main Lisp session plus several processes that implement 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 are forced to join sessions on client 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.

Our system is based upon the *storable* class of objects which is a superclass of the SIMPLE-OBJECT class. Storable objects[4]can be encoded into an ascii representation. 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 representation.

The object world is implemented using three levels of storage that allow management of storable objects:

[4] 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.

Local cache: is local to single workstations and contains a set of replicas of objects that are stored in the local workstation. This is implemented using a hash table that provides access to an object using either its 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. The global cache is implemented as a pair of hash tables that provide access to objects based either on its unique identifier or on its name and type. Each entry of the global directory has a list of the machines that store replicas of the object.

Central secondary storage: The central 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.

The following functionality exists for handling objects stored in the object world.

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.**

save-object: takes an object as argument. The object is saved to disk, sending an encoded representation to the central server over the network. If the object is a composite its components are saved recursively.

delete-object: takes an object as argument. The object is marked as deleted for later removal by a garbage collector process.

rename-object: takes an object and a string as arguments. The name of the object is made to be the string argument. All tables are updated accordingly.

copy-object: takes as argument an object and creates a new instance of the object, copying all its slot values. If the object is a composite the components are copied recursively.

encode-object: takes as argument an object and produces a string that contains the unique id of the object, its symbolic name, the name of its class and a list of slot name and value pairs. The slot names and values are those of the CLOS class that represents the OREL classes.

construct-object: takes as argument a string that contains the encoding of an OREL object and produces an instance of the object.

If the slots of an object contain data such as numbers, symbols, and strings, the object 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.

Site-specific data is necessary for dealing with objects in the Lisp/X Window System interface. Objects dealing with windows have pointers to several device dependent data structures. We have written specializations of encode-object and construct-object for such objects.

Pointers to storable objects are encoded as a call to read-object. An example will help make this clear[5] . Consider this class definition of a binary search tree:

```
(defclass search-tree (storable)
  ((key :accessor key :initform 0)
  (left :accessor left :initform nil)
  (right :accessor right :initform nil)))
```

An object of this class would have three slots (besides the slots of its superclasses), one for its left subtree, one for the right subtree, and a slot for the key of the node. Assume that we evaluate the following two forms:

[5] CommonLisp/CLOS syntax is used and the code is formatted in a typewriter font (Courier).

```
(setq root (make-instance 'search-tree))
(setf (key root) 15)
```

Make-instance creates an instance of this class and then we assign a value of 15 to the slot key. Evaluation of (encode-object root) produces the following representation:

```
"(SEARCH-TREE NIL
     "RA.asa-105558"
     (KEY . 15 )
     (LEFT . NIL)
     (RIGHT . NIL))"
```

We can create two more nodes in this tree evaluating the following forms:

```
(setq node-1 (make-instance `search-tree))
(setq node-2 (make-instance `search-tree))
(setf (key node-1) 10)
(setf (key node-2) 20)
(setf (left root) node-1)
(setf (right root) node-2)
```

Evaluation of (encode-object root) will yield now the following encoded representation of root.

```
"(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. So the representation of `root` shows that the `key` is (still) 15, but the `left` and `right` subtrees now have a value which is a function to read in the correct subtrees. The `decode-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 `node-1` and `node-2` 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. Defbroadcast generates three different methods, representing the following three parts:

1. Code that implements that method's functionality.

2. An :after method that broadcasts a request to invoke the method over the network.

3. A plain version of the method without an :after method. This is the version that other sites will run in support of the

original method. If they were to try 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[6]. `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/CLOS, as follows:

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

In this definition, the class `counter` is a subclass of the `storable` class. We have one slot named `value` which has an initial value of zero and whose value can be accessed by `value`. We create an instance of this class and do two `increment` operations by the command:

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

[6] This example was developed as an early test for broadcast methods by S. Berson.

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

```
(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. The macro-expanded code for the increment method is shown in Figure 4.19.

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 since other sites are independently applying that method. But the local site must be smart enough not to broadcast the second method or else all the remote sites will do the second operation twice. This problem is dealt with 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) means that the variable can be referenced anywhere as long as its binding is currently in effect.

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. The `send-args` function resolves that problem by making sure all objects mentioned in the method's argument list are stored at the sites where the broadcast method will be executed.

The macro generates the :`after` method, which handles the broadcasting. The broadcast-message function sends a message to a specific Unix socket using UDP datagrams, in this case, the update socket to which all participants listen for updates. The message contains a function call to `propagate-update`. The function `propagate-update` takes a list of sites and a form encoded as a string which is to be evaluated. Each site, on receiving through the update socket a `propagate-update` message, will check the site list. If it finds itself on the site list, it evaluates the form. In this case, the function call is to `nb-increment`.

```
(defmethod increment ((c counter))
  (if *sync*
      (progn
        (setf (value c) (+ 1 (value c)))
        c)
      (let ((*sync* t))
        (declare (special *sync*))
        (send-args (flatten-out (list (get-id c))))
        (setf (value c) (+ 1 (value c)))
        c)))


(defmethod increment :after ((c counter))
  (if (not *sync*)
      (broadcast-message
       (ow-update-socket object-world::*ow*)
       (concatenate 'string
                    "(propagate-update "
                    (package-name *package*)
                    " "
                    (get-dests (flatten-out (list (get-id c)))))
       (write-to-string (cons 'nb-increment (list (encode-all c))))
       ")")))

(defmethod nb-increment ((c counter))
  (if *sync*
      (progn
        (setf (value c) (+ 1 (value c)))
        c)
      (let ((*sync* t))
        (declare (special *sync*))
        (setf (value c) (+ 1 (value c)))
        c)))
```

Figure: 4.19 Expansion of defbroadcast

4.5. The power of OREL

Section 4.2 showed how OREL is used to model SARA primitives. In Chapter 5 we show how to use OREL to build a data model of a general tool object for coSARA (Figure 5.8) and how to build a a data model of one of the tools of coSARA.

In this section we present one more example of modeling data using OREL. GNU Emacs (Stallman [1981]) has a hypertext system (Conklin [1987] is a survey of hypertext systems) to store manuals of the GNU software called Info. This hypertext system stores a collection of named text nodes as a tree hierarchy. All brother nodes in the tree are linked together in such a way that it is possible to traverse them left-to-right and right-to-left. Arbitrary cross references between nodes also exists. Figure 4.20 shows an example of tree structure supported by GNU Emacs' Info.

The operation of GNU Emacs Info is as follows. The tree hierarchy is contained in the text of the node in the form of a menu. Cross references are also contained by the text of the node and occur at specific points in the text. The user of Info can navigate the hypertext in two ways:

- The user can traverse the tree hierarchy moving from a node to any of its children nodes chosen from the nodes' menu; the user can move from a node to either the node that precedes it in the menu or the node that follows it in the menu; the user

can move up from a node to the node's parent (except for the root node).

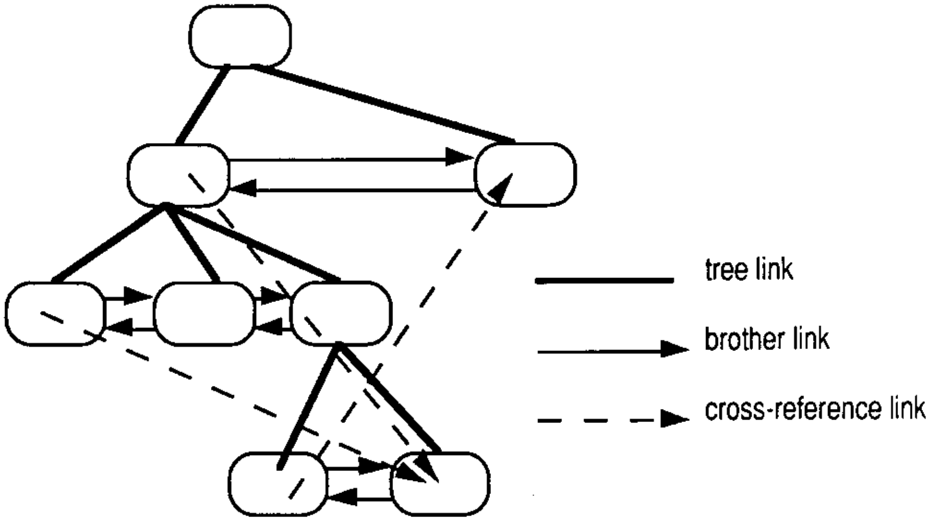


Figure 4.20: Example tree structure of GNU Emacs Info

- The user can move from one node to any other node that is linked using a cross reference link.

- The user can move to any arbitrary node by using the nodes name.

Figure 4.21 shows an OREL data model of GNU Emacs Info nodes. The class of nodes is represented by INFO-NODE. This node is related to itself using the directed pairs UP, PREV and NEXT to manage moving in the tree hierarchy up to the parent node and between brother nodes.

The class TEXT stands for the text of a node. It is related to INFO-NODE through the directed pairs: INFO-TEXT that associates a piece

The class TEXT stands for the text of a node. It is related to INFO-NODE through the directed pairs: INFO-TEXT that associates a piece of text with a node and MENU, that associates a piece of text with a list of nodes. XREF-POINT is a class of objects that represent cross-references that occur within a piece of text. INFO-NODE, TEXT and XREF-POINT are related by XREF which given a piece of text and a cross-reference will produce the node linked through the cross reference.

To support arbitrary access by name, node names are stored as objects of class NAME, which is related to INFO-NODE by the directed pair NODE-NAME.

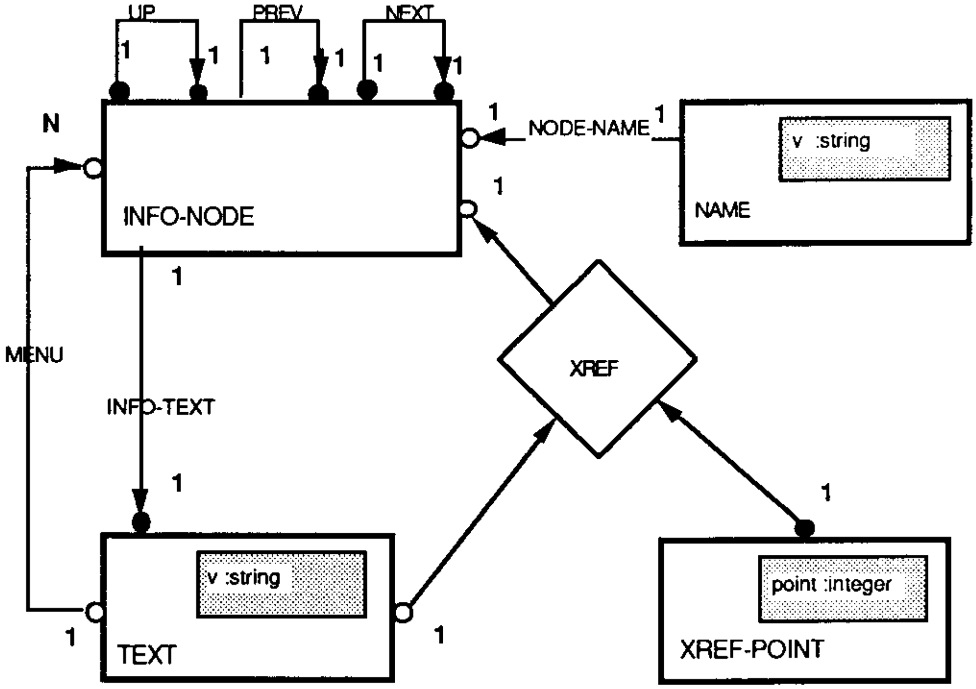


Figure 4.21 OREL model of GNU Emacs info nodes

4.6. Related work

In this section we discuss work related to OREL. We review some of the most significant efforts: Smalltalk (Goldberg [1984]), CLOS (Bobrow et al. [1988]), ORION (Kim et al [1987]) and Augmented entity-relationship models (AERM) (Landis [1988])

4.6.1. Smalltalk

In the early 1970s, at the Xerox Palo Alto Research Center, the Learning Research Group started the Smalltalk project. Smalltalk is currently a commercial product. The goals of the proposed project are described by Goldberg & Robson [1984, Preface] as:

> "...to create a powerful information system, one in which the user can store, access and manipulate information so that the system can grow as the user's ideas grow."

To realize the Smalltalk vision, research focused on two principal areas: a language description and a user interface which matches the human communication system to that of the computer.

The Smalltalk project exemplifies object oriented systems and provides a vocabulary for further discourse (Goldberg [1984]; Goldberg & Robson [1984]).

Smalltalk-80 is an integrated programming environment. In addition to a programming language and an interactive graphics

system it provides functions normally associated with an operating system. These functions include memory management, a file system, processor scheduling, display handling, and compilation.

The Smalltalk system itself is fully object-oriented. At the highest level the user's interaction with the system can be viewed as an interaction between two super-objects, the User and Smalltalk. All data in Smalltalk are objects. Each object type can be further classified into more specialized subtypes. The most important concepts in Smalltalk are classes, objects, methods, and messages.

In the Smalltalk programming language, the primary unit of data organization is the object. Everything in the system is represented and manipulated as an object. An object is a data structure consisting of local private memory and a set of operations, called methods, to manipulate information stored in the private memory or to perform actions based on that information. The only way to access the private memory of an object is through the methods defined for that object. An object is similar to the concept of an abstract data type in other programming languages. Other examples of objects include numbers, character strings, queues, rectangles, file directories, compilers, text editors, and programs.

A class is a description of a set of objects of the same type. The individual objects described by a class are its instances.

The methods of an object are its interface with the other objects in the system. The medium of communication between objects is the message. Whenever an object A (the sender) wants to get another object B (the receiver) to do something, A sends B a message which is interpreted by B's message interface. The message initiates execution of one of B's methods which calculates a response, and B then returns this response to the sender. This is the primary way of making things happen in Smalltalk.

Another important concept of object-oriented programming in Smalltalk is the subclass. A subclass is a class of objects possessing all the variables and methods of some other class, called its superclass, except for certain explicitly stated additions that extend or override the variables and methods of the superclass. A simple analogy to the subclass can be found in traditional dictionary definitions of English words. For example, a man can be defined as a male adult human. A father can be defined as a man who has one or more children. The class father is a subclass of the class man, and the class man is a superclass of the class father. Notice that father is defined in terms of man but is more specialized. In order for an object to be a father it must not only be a man but it must also have one or more children. All fathers are men but not all men are fathers. A subclass is thus a proper subset of its superclass.

Another idea important to subclasses is inheritance. A subclass is said to inherit all the attributes of its superclass. A father possesses

all the attributes of a man, with some additional attributes not required of the man class. A Smalltalk subclass inherits all the variables and methods of its superclass but it also adds new variables and methods in its implementation description, or it may override a method of its superclass with a new definition. Inheritance is transitive. If class B is a superclass of class C and class A is a superclass of B then by the definition of inheritance class C inherits the attributes of class A is well as B.

Smalltalk is an interactive programming environment with a graphical interface. Designing a Smalltalk program requires an implementation of each of the program's objects including a visualization of that object, which need not be graphical. To support the desired graphical interaction, Smalltalk expects a high-resolution graphical display screen and pointing device such as a mouse; besides the Smalltalk programming environment includes extensive graphics facilities in the form of library classes (for example, arc, rectangle, pen, etc.)

4.6.2. CommonLisp Object System

The CommonLisp Object System (CLOS) is an object system integrated in CommonLisp (Steele [1990]). It provides the usual features found in current object oriented programming systems: multiple inheritance, classes, metaclasses, instance and class variables, polymorphic functions. Less usual, is the method combination mechanism. We will discuss how CLOS integrates into

CommonLisp, the treatment of behavior encapsulation in the form of methods and the method combination mechanism. The other concepts have already been discussed.

In CommonLisp every Lisp object has a type and a number of operations are defined for handling types. CLOS introduces classes. Each class is a CommonLisp type, and the same operations that are available for handling basic CommonLisp types are also available for classes. Thus, CLOS augments the CommonLisp type system.

Methods are integrated in CommonLisp using generic functions as a base. To the caller generic functions appear exactly like any ordinary Lisp function. No syntactic difference is made. The implementation of a generic function is done by a set of methods. A particular method implementation is used when a generic function is invoked, depending on the classes of its arguments.

In other languages, like T (Slade [1987]), methods are dispatched according to the type of only one argument, providing a strong encapsulation of behavior in the corresponding class. In CLOS however, the selection of a method to be invoked is done based on the types (or classes) of all the arguments. This means that the encapsulation is weaker. For example, a method F is defined to have arguments of classes X and Y. How do we decide the encapsulation of F? Is it encapsulated by class X or Y?

Methods can be combined according to a given set of rules. A method can be defined to be invoked before, after or around a primary method associated with the same generic function. A primary method does not have any method combination specified for its execution, and usually does the principal part of the implementation of the generic function. A before method is executed before the primary method. An after method is executed after the primary method. An around method shadows the primary method and the decision of choosing when to call a primary method is left to the implementation of the around method.

Keene [1989] gives a thorough exposition of CLOS and illustrates how programming can be done effectively in the realm of CLOS.

4.6.3. ORION

Orion is a "prototype database system that adds persistence and sharability to object created and manipulated in object-oriented applications" (Banerjee et al. [1987]).

Orion supports all of the traditional object oriented features of programming languages, including multiple inheritance, with the added capability of letting the user define the permutation of the superclasses used for determining inherited properties. This is flexible, but may introduce undesired confusion in the programming.

Other features present in Orion that deserve mention are: composite objects, version control, locking types suited to support of long

transactions, and an evolutionary data schema that can change as defined by an established evolution taxonomy and rules.

4.6.4. The Augmented Entity-Relationship Model

The basic ERM (Chen [1976]) uses entities and relations as modeling primitives. Entities denote sets of data objects, similar to classes in object oriented programming systems. Relations do not have an explicit correspondence in object oriented programming systems and constitute an enrichment, expressing more information than can be done using common class definitions, such as those of CLOS. Relations can be considered to be special classes of objects that store an association between other objects. Roles seem to be a concept similar to subclasses.

The augmentations proposed by Landis [1988] are:

- Generalization: this augmentation is equivalent to simple inheritance, and therefore not enough to match the power of multiple inheritance found in current object oriented programming systems. Landis claims the reduction in redundancy of property specifications as a justification of the introduction of this feature in the ERM.

- Aggregation: This is a form of composition in which higher level objects called aggregates are defined to be composed of other objects and relations. They can be referred to as a whole and are in general treated as an entity. Landis

definitions: exclude recursive aggregation forcing the use of recursive relations, that in turn require the use of roles; and restrict the domain of objects that can take part in a relation tuple. Aggregation is not found in object oriented programming systems, and is desired as an enrichment of such systems. For example, a control-graph is an aggregation of control nodes, control arcs and the relations that associate control nodes and control arcs (an AERM diagram is shown in Figure 4.22) Other parts of the model can make reference to control-graph as it were an entity. In particular it may be a component of a GMB aggregate. It is claimed that this augmentation is essential for modeling entities like the SARA control graph, which is composed of other entities.

- The main purpose of roles in the original ERM was to provide a naming scheme to eliminate ambiguities in relations in which the same entity would participate more than once. For example, in Landis' modeling of the module-submodule relation, the entity module participates twice, one in the role of parent and another in the role of child. Landis proposes to attach properties to roles so as to associate properties with subsets of entities (those participating in the role involved.) This mechanism does not have a clear semantic definition however, and conflicts arise for example in recursive relations such as the one used to represent the module-

submodule relation. Landis does not present any case in which role properties are necessary or convenient for modeling SARA objects. The use of existential constraints in roles is shown to aid in the modeling the relation between sockets and interconnections.

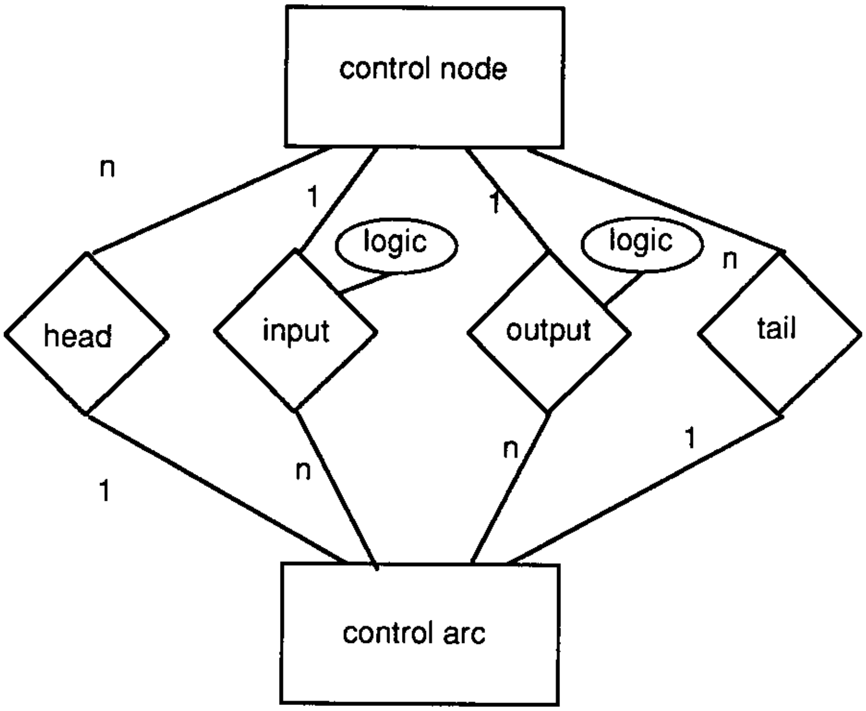


Figure 4.22:AERM diagram of SARA control graphs

- Property types: text (ASCII files), code (programs) and references (pointers to other objects), are defined as valid types of properties. In an object oriented programming system the slots of a class can have any type that is valid in the system. AERM properties are equivalent to CLOS slots.

This augmentation should be extended to cover any value that would be a valid CommonLisp object.

It is not demonstrated how this augmentation is essential or convenient for modeling SARA objects.

- Mapping class: a relation in the basic ERM can have a mapping class (1:1, 1:n, n:m), and a dependency or identity constraint. Additional mapping classes that lead to other integrity constraints constraints are introduced in the AERM: completeness, direction and order. All these are ways to capture additional semantics.

  Only total and ordered relations are justified as necessary. The first is used for modeling the control-data mapping in the GMB. Ordered relations are claimed to be needed to model input/output logic of control nodes in the GMB.

The AERM defines the following methods for operating on data that abides by the model's rules:

Find: selects and returns a unique set of identifiers for which some qualification clause is true. For example, in Figure 4.23 there is an AERM that has an entity called OPERA with properties NAME and TENOR, a find operation might be of the form:

```
find OPERA where NAME = "Otello"
```

Retrieval: is initiated after a unique identifier has been established using find.

Insertion/deletion: allow objects to be placed in the database or removed form the database. For example:

```
insert OPERA( NAME = "Otello", TENOR = "Domingo" )
```

Modification: allows changes in the value of properties attached to entities and relations. For example, if X is the unique identifier established by:

```
find OPERA where NAME = "Otello"
    and TENOR = "Domingo"
```

then the following would update the state of this instance of OPERA:

```
change TENOR = "Domingo" of X to "Vinay"
```

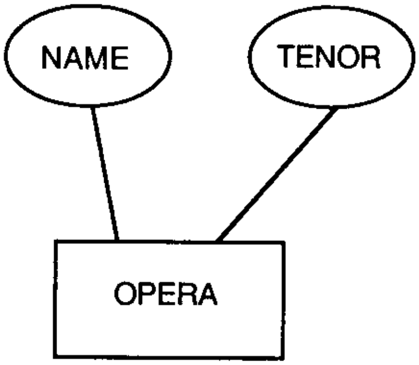


Figure 4.23: Example AERM

4.7. Review of contributions and directions for future work

4.7.1. A review of claimed contributions

The main contributions of our work on OREL are:

- A *distributed system* in support of *interactive sharing of data* objects with *persistent storage* (section 4.3 and section 4.4)).

- An object-oriented *graphic language for modeling data*, that includes: classes and multiple inheritance, recursive composition of objects, relations as first class objects and integrity constraints on relations (section 4.2).

- A *programming interface* that allows manipulation of distributed, interactively shared data objects (section 4.2 and Appendix II).

It was necessary to extend the classical notion of object to make possible the implementation of OREL. The extensions are:

- objects are accessed from many sites simultaneously,

- objects may contain site dependent data,

- objects have unique, site-independent identifiers,

- objects may have symbolic names, which are not necessarily unique,

- objects are used within long transactions and are subject to locking.

There are current research efforts aiming at efficient management of shared, persistent objects. Decouchant [1989], focuses on extending Smalltalk with these features, Moss [1989] is working to smoothly integrate database an programming language concepts. None of the current research, however, achieves interactive sharing of data objects. This is a unique characteristic of our work. Section 3 shows how we have designed an implemented OREL to support interactive sharing of data objects.

Other researchers have found that the combination of object-oriented techniques and relational database concepts constitutes a powerful data modeling methodology. Chen [1976] uses entities and relations as modeling primitives. Entities denote sets of data objects, similar to classes in object-oriented systems. Relations do not have an explicit correspondence and constitute an enrichment, expressing more information than can be done with class definitions. Landis [1988] proposes a number of augmentations to Chen's entity-relationship model for modeling complex design data. Landis' augmentations were targeted at the modeling and implementation of SARA/IDEAS (Landis [1988]; Worley [1986]) and are discused in detail in the section on related work.

Our data modeling language makes use of relations as a way to increase expressiveness and to enable management of constraints.

Unlike previous work, OREL relations are first class objects. Recursive composition also provides an effective way to manage complexity of design data.

The capability to interactively share data objects is the key notion that enables collaborative work by a group of designers. Distributed operation is essential if the system is to support of group of collaborating designers, that will each be operating the system from his/her own workstation. Persistent storage of data objects is necessary because the amount of data involved in a design task is such that it is not possible to build the data from scratch when starting a session and it is necessary to keep track of work progress across session boundaries.

We have used OREL to model SARA objects (Figure 4.12, page 16). The model that we produced is quite compact (fits in one page) and enables us to deal with the complexity of SARA objects. In particular we would like to highlight the modeling of modules as recursive composites and the use of directed pair relations in modeling GMB objects, where the hyperarc structure of the control graph is of great complexity. We have translated this model following the procedures described in this Chapter and produced code that is currently used in our implementation of coSARA.

We have built a prototype of the UCLA SARA Collaborative Design Environment, using the implementation of OREL that is discussed

here. The existence and behavior of that environment is a proof of concept for OREL.

4.7.2. Future work

Our research has produced a number of problems for future work. The main problems that are open for work are in the areas of supporting operation by geographically remote sites, concurrency control, object-oriented data modeling language design, abstractions for programming with objects and relations and implementation optimizations. We discuss these issues below.

4.7.3. Supporting operation by geographically remote sites

Currently, OREL operates correctly on a single local area network. This is because we take advantage of built-in facilities for broadcasting messages, that guarantee that every site will see the same sequence of messages. If this feature is removed, then sites on different local area networks might see different sequences of messages. We know that there is a need for computer supported cooperative work to span more than a single local area network. In particular, support for collaboration is even more important when two or more geographically separate design groups work on related parts of a design.

In anticipation of new high speed network technologies, we are developing new protocols to expand OREL to support such multiple geographically remote local area networks, and understand that

inherent artifacts will change the quality of collaboration as compared with collocation.

We believe that extensions to TCP/IP for multicasting and broadcasting would provide the necessary functionality to build support for geographically remote operation. Such extensions would be used as a foundation for the implementation of broadcast methods.

4.7.4. Concurrency control

Concurrency control needs of computer supported cooperative work are very different from conventional requirements. The traditional database model of transactions is inadequate. In conventional systems the intent is to give each individual user the illusion of being the only person operating the system. In collaborative systems the aim is the opposite. We intend to make each user of the system as aware as possible of other users actions.

4.7.5. Object-oriented data modeling language design

In the operation of coSARA there is a natural amount of parallelism due to the activities of the collaborating designers. An object model similar to actors (Hewit & Lieberman [1981]) could have an impact on the ability to model and program a system such as coSARA. Further research is necessary to evaluate this conjecture.

In section 4.2 we demonstrated the use of OREL for creation of a data model of design data used by SARA tools in support of the SARA

design methodology. In the process of creating this model and using it as the base for programming coSARA, several issues about the use of OREL emerged. The essential problem is how to set the programmer's mind to think of relations. It is necessary to research which are adequate abstractions for using a system such as OREL. For example, before we started to seriously focus on the modeling of SARA objects there were no pair objects in OREL, only general relations. During the process of modeling the SARA objects it became clear that many relations defined in the model were simply relating pairs of objects. This resulted in the inclusion of directed and undirected pairs as useful abstractions. We made use of this abstractions to introduce some optimization in their implementation, relative to the implementation of general relations.

4.7.6. Optimizing the implementation of OREL

OREL influences the performance of a system built with it in two ways: broadcast methods impose a significant overhead, and the generality of OREL protocols also carries an overhead.

There are not many ways to improve the overhead of broadcast methods other than careful programming of their implementation and judicious use of them.

The overhead of the OREL protocol is more amenable to improvement. Since the protocol is composed of messages, it is easy to redefine methods for specific OREL classes. For example, the

messages relate, unrelate, and find-objects could be redefined for certain pair classes to use hash tables to store the relation pairs.

CHAPTER 5

Tool Modeling and Integration Methodology

5.1. Introduction

This chapter reports research work on a methodology for modeling and integration of tools into integrated extensible environments. The modeling and integration methodology includes the case of partial integration of foreign, pre-existing tools.

For the purpose of this work 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. coSARA is such an integrated, extensible environment aimed at providing computer support for collaborative design. The collaborative nature of coSARA imposes two basic requirements on the environment:

- Availability of a multi-user interface, and
- ability to interactively share design objects.

We use OREL [Chapter 4] as a common data management system for all tools built according to our tool model. It has been shown previously how OREL satisfies the requirement of providing support for interactive sharing of design objects. OREL also provides a data modeling language and data model support that is used to model tools and to model data objects handled by tools.

The main contributions of this research in the area of tool modeling and integration methodology are:

- The coSARA tool model for environment extensibility and interactive tool sharing: allows partial and full integration of tools; enables tools to operate on interactively shared objects; and supports incremental extension of the environment.
- The coSARA user interface model and user interface development system: allows sharing of interaction mechanisms; provides support for expressing behavioral response to multi-user actions and enables early testing and analysis of user interfaces.

A great deal of research effort is being currently devoted to the problem of building integrated extensible systems (Henderson & Notkin [1987], editorial note), with particular attention to the management of pre-existing tools. Some of these efforts lead to systems that are said to be *open,* in the sense that they offer a model of data and control that newly written applications can adopt. There is no known solution however to the automatic inclusion of a pre-existing tool into an integrated environment. In this sense, integration and extension represent conflicting forces. The tighter the integration, the more similar application tools have to be. More general extensibility capabilities, that account for inclusion of foreign pre-existing tools, require that the coupling between application tools be as loose as possible.

Our research has produced a methodology for modeling and integration of tools that is not found in other systems. The approach

is to allow varying degrees of tightness in the integration of a foreign pre-existing tool in order to deal with conflicting forces that arise from the integration vs. extensibility issue. The methodology 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. 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.

User interface development systems have also been a productive field of research for the past decade (Myers [1989]). In particular, some researchers 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 (Greif & Sharin [1987]; Stefik et al. [1987]; Ellis et all [1990]; Gust [1988]; Lantz & Lawers [1990]).

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 research in user interface methodology has resulted in a unique approach: to formal modeling of user interfaces that are aware of multi-user actions, to allowing true semantic sharing of user interface objects and to allowing modeling of group processes into the user interface. 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, menus, etc. In this report we define precisely the user interface objects that are used in the coSARA environment.

This Chapter is organized in the following sections:

5.1. This introduction.

5.2. Description of requirements for tools and for tool modeling and integration methodology.

5.3. A user interface development system composed of a user interface model, a specification method and a procedure for construction of user interfaces. We show how this user interface development environment satisfies the requirement of providing a multi-user interface which is common to all tools that operate in the coSARA environment. The user interface of specific tools is specified by SARA models. These models provide a formalism that enables early testing and analysis of user interfaces and allow modeling of tool behavior in response to concurrent actions performed by multiple designers.

5.4. A data model of generic tool objects that encapsulates properties which are common to all tools. This section also shows how this general model is specialized for specific tools.

5.5. A procedure for tool integration that provides a method for building new integrated tools and for partial integration of pre-existing foreign tools.

5.6. A review of related work.

5.7. Review of contributions and directions for future work.

5.2. Tool Requirements

The following are requirements that tools must satisfy for the purposes of a computer-supported collaborative design environment such as coSARA:

- Tools should be adaptable to designer and group needs.
- Tools should affect the shared objects under the control of a designer or under the control of other tools.
- Tools should be objects defined in the coSARA data model.
- Tools that are integrated into coSARA, use the common data model and may require the modification of an existing data model to behave as intended.

The following are requirements for the tool modeling and integration methodology:

- The tool modeling and integration methodology should be able to deal with foreign tools.
- The tool modeling and integration methodology should allow incremental extension of the environment by integration of single tools without need to rebuild the environment. It is admissible, however, that the environment be rebuilt when a

tool requires drastic changes either in the common data model or in the common multi-user interface.

- The tool modeling and integration methodology should allow modeling of behavior in response to actions performed by multiple users

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 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.

5.3. User interface model and development system

The user interface of a system is the component that collects input data from users of the system, passes collected data on to the system for processing and presents output data to users of the system. For

example the user interface of a text editor collects text and editing commands from a user and displays the current state of the file text that is being edited.

The user interface code usually accounts for a large part of the code in the implementation of a system. Surveys of artificial intelligence applications report that about half of the code and runtime memory are devoted to the user interface (Myers [1989]; Bobrow, Mittal& Stefik [1986]).

An important kind of user interface is *direct-manipulation interfaces*. These let the user operate directly on objects that are visible on the screen (Schneiderman [1983]); Myers [1989]) says:

> "...The easy-to-use direct-manipulation interfaces popular on many modern systems are among the most difficult to implement...
>
> ...Direct-manipulation interfaces are difficult to create because they often provide elaborate graphics, many ways to give the same command, many asynchronous input devices, a mode-free interface and rapid semantic feedback."

One important aspect of user interfaces is how they communicate with the applications at run-time. Two main styles are used (Myers [1989]):

Internal control: The application program calls the interface procedures when input is desired and when output data must be presented.

External control: The user interface procedures call the application when the user gives a command. This allows the user interface to handle scheduling and sequencing of actions in response to user actions.

Communication through external control can be further classified by the form in which data and control are transferred between the user interface and the application programs. This may be done by callbacks, where the application passes to the user interface names of procedures to be invoked; or by shared memory, with user interface and application programs each polling the data to check for changes.

An important communication issue is the bandwidth between the application and the user interface. Early models advocated narrow band communication, tending to make the application independent of the user interface. However, to satisfy the rapid semantic feedback required by modern direct-manipulation interfaces, the user interface and applications must communicate in a tighter and more frequent form, for example, to determine legal positions when an object is dragged on the screen. Proposed user interface models address this issue by enabling the user interface and applications to share data. Sharing data and still providing good modularity is currently an open research issue (Myers [1989]; Szekely [1988]).

To alleviate the problem of building interfaces, methods and tools have been developed to aid in the process of user interface construction. One simple approach is the provision of a toolkit, that embodies a number of procedures for support of interaction between the user and the applications. A more thorough approach is found in User Interface Development Systems (or User Interface Management Systems, as they are sometimes called.)

A user interface development system provides functionality to support all aspects of the user interface development and use: specification of user interfaces, management of different input devices, validation of inputs and error handling, provision of feedback to show that input has been received, presentation of output data, help and facilitation of usage of the system and interaction with the systems's semantic processor.

To do this user interface development systems are usually composed of:

User interface toolkit: A user interface toolkit is a collection of data structures and associated procedures to support collection and presentation of data.

Dialog manager: The dialog manager is a module that accepts valid sequences of input from the user.

Programming interface: The programming interface is used to implement the interaction between the semantic processor and the user interface.

Layout editor: A layout editor allows the specification of the layout of different regions of the screen that will be used for interaction between the user and the system.

Analysis components: Analysis components are used to test and evaluate the user interface in early prototype stages as well as during production usage.

There are several ways in which a user interface designer can specify the interface. The specifications may be based on: menu networks, state transition diagrams, grammars, event languages, declarative languages, object oriented languages and knowledge based systems. There are interfaces that are "created by demonstration," like Peridot (Myers [1988.]). Some of these concepts are discussed in greater detail as we review significant development efforts and current research projects in the area of User Interfaces.

UIDS which are based on state transition diagrams use a finite state machine to recognize the dialog language. These are augmented with calls to routines that implement the semantics. This idea has been expanded to use more powerful formalisms, like ATNs (Worley [1986]) and pushdown automata (Olsen [1984]). Practical current systems use a formalism called Recursive Transition Networks

(RTN) in which subnetworks can be invoked as parameterless subroutines. It can be shown that RTNs have a recognition power equivalent to deterministic pushdown automata. In any case processing proceeds by simply walking the graph.

An example of this method is RAPID/USE. (Wasserman & Shewmake [1982]). RApid Prototypes of Interactive Dialogues (RAPID) is a tool to support the User Software Engineering (USE) method (Wasserman & Shewmake [1982]). This method is similar to other methods proposed recently (Jacob [1985]; Jacob [1986]). A description of RAPID/USE is found in subsection 6.1. On related work (page 22). Modern approaches tend to favor event based interfaces, use of the object-oriented paradigm and knowledge based systems

5.3.1. The user interface model

The multi-user interface model that we use in coSARA is shown as a SARA model in Figure 5.1. This model is a refinement of the High level model of Figure 3.2.

We assume that a small group of designers operate the coSARA system from the environment making gestures on displays with mice, keyboards and other input devices. A gesture may be to move the mouse, to press a button of the mouse, to release a button of the mouse, to click a button of the mouse, to type using the keys of the

keyboard. The system responds to gestures with graphic output on displays and possible other types of feedback such as audio and video.

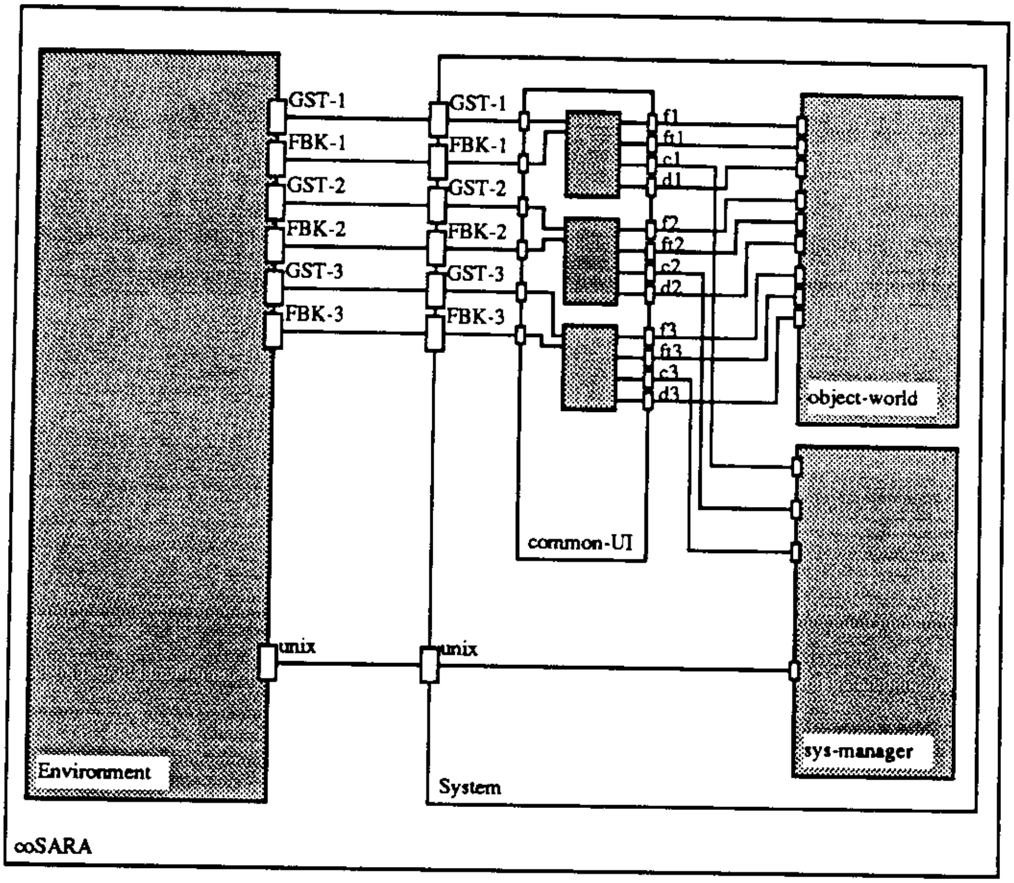


Figure 5.1: Model of coSARA

The Common-interface module is composed of one session$_i$ (i=1,...,3)[1] module. Each session$_i$ module corresponds to one designer in the environment. The individual sessions that compose the Common-interface module do not communicate directly with each other. The

[1] We use the number 3 here just as an example value.

different sessions communicate and coordinate by sharing objects that are stored in the object-world module.

The designer's gestures are received by the common-UI module through its GST-i, (i=1,...,3)[2] sockets. The common-UI module then delivers system feedback to designers through its FBK-i sockets.

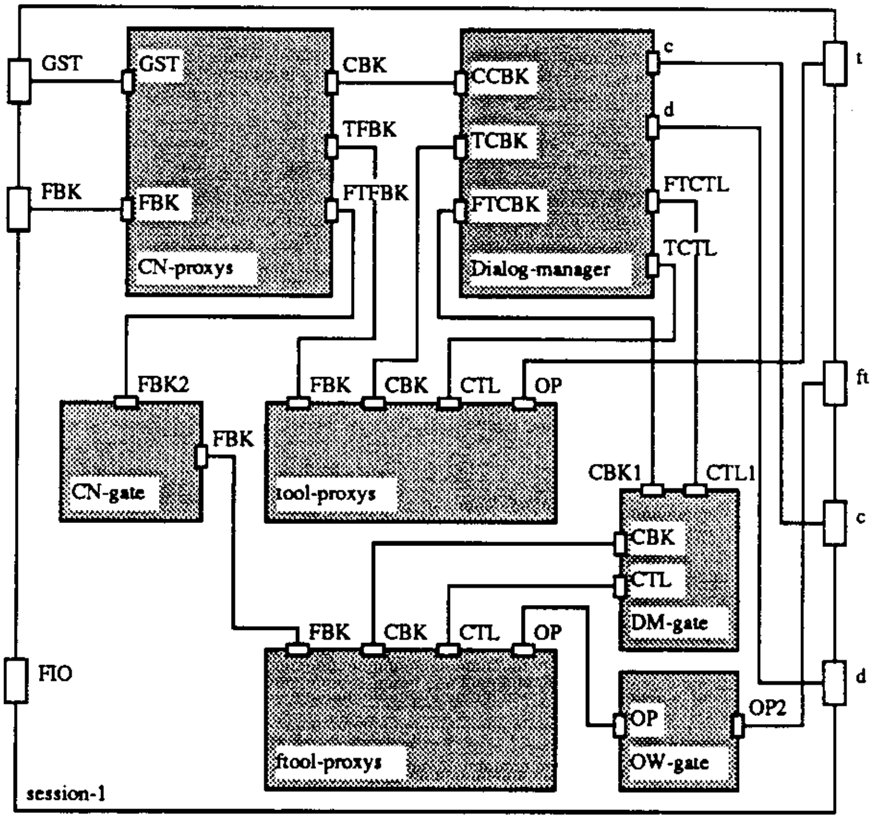


Figure 5.2: Model of session-1

Figure 5.2 shows a partition of a session module. It has the following component modules:

[2] In the following presentation we will refer to sockets such as GST-1, GST-2 and GST-3 collectively as GST.

tool-proxys represents the tool objects that are used by the designer who is operating the session at a given time. We call tools that are in use *active tools.* The real tool objects reside in the object-world. The properties of the object-world assure that tools can be shared by two or more designers.

CN-proxys represents the contact objects used by active tools. The real contact objects reside in the object world. The properties of the object-world assure that contacts can be shared by two or more designers.

ftool-proxys represents foreign tools. A foreign tool proxy acts similarly to a tool proxy except that interaction between a tool proxy and its environment is subject to translation procedures performed by CN-gate, DM-gate and OW-gate.

Dialog-manager is a module that controls the flow of control in the user interface activities. This module translates user actions into invocations of tool methods.

CN-gate receives feedback from the foreign tool through its proxy and translates it into a form that a contact can interpret.

DM-gate receives callbacks from ftool-proxys through the CBK socket. These callbacks are translated into coSARA callbacks and delivered to the Dialog-manager for further processing. DM-gate also receives method invocations from the Dialog

manager, which it translates in whatever form is necessary to invoke functionality of the foreign tool.

OW-gate transforms operations on data objects from the original form of the foreign tool into a form that is understandable by the object-world module.

Let us study now the behavior of the common-UI module. The common-UI module receives gestures from the designers that reside in the Environment module. These gestures are given to the session-i module that corresponds to the designer who made the gesture. Gestures are used by the common-ui to drive tools that will produce feedback for the designers as a result of their operation. As we said before, gestures are received through GST sockets and feedback is delivered through FBK sockets.

The first component of a session-i module that processes a gesture is the module CN-proxys. This module translates a gesture into a *contact callback* that is delivered through the socket CBK. This socket is connected to the socket CBK in the Dialog-manager module for the purpose of delivering the callback to the Dialog-manager. The CN-proxys receives feedback information from both the tool-proxys module and the ftool-proxys module, the latter through the CN-gate module. Feedback is processed by contacts, whose proxys reside in CN-proxys, which make whatever transformations are necessary for presentation to the designer. For example coSARA has contacts for drawing graphics that make coordinate transformations. Another

example is a contact that presents a request for input as a menu on the screen while a second contact presents the same request for input using audio.

The tool-proxys module receives method invocation from the Dialog-manager module throught the socket CTL. Invocation of a tool method results either in feedback sent to CN-proxys through the socket FBK or in callbacks being generated by the tool and sent to the Dialog-manager throught the socket CBK, or both. A method invocation will usually produce alteration of objects in the object world. Connection between the tool-proxys and the real tool objects is done through the socket OP.

The structure and behavior of ftool-proxys, the module that represents foreign tools, is similar to that of tool-proxys, except that all its communication with other modules is done through gateway modules (CN-gate, DM-gate and OW-gate). The behavior of CN-gate, DM-gate and OW-gate was already described.

The structure of Contact-proxys and Tool-proxys is determined by the activation of tool objects and their subsequent removal. The SARA modeling methods do not allow dynamic reconfiguration of system structure. For example, a session has a set of active tools represented in the tool-proxys and ftool-proxys modules. When a new tool is activated it becomes necessary to modify the internal structure of these modules. We have worked around this problem using an

external procedure, not present in our model of the coSARA system, that knows how to reconfigure a working system.

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.

This user interface model includes:

- *Interactive sharing of arbitrary contacts and tools at a semantic level:* Contacts and tools are stored in the Object-world, which provides that kind of object sharing (In Chapter 4 we showed how OREL allows interactive sharing of objects).. This is a novel feature, that cannot be found in other systems. Functionality to share user interface objects can be found only at the level of sharing windows through shared window servers (Gust [1988]), shared display areas (Stefik et al. [1987]).

- *Incremental extension of the user interface:* Each tool provides its own piece of code that is invoked by the user interface dialog manager. In this way the system does not need to be rebuilt when a new tool is brought into the system.

- *Satisfaction of the common user interface requirement:* All tools interact with the designers through a common set of contacts and are driven by a single dialog manager.

- *Provides support for foreign tools:* The contact and callback gateways allow interaction between foreign tools and the common user interface.

5.3.2. The user interface creation procedure

A procedure for building and extending the multi-user interface of an integrated system such as coSARA is required to produce interfaces according to the model previously described and to enable incremental introduction of new tools into the environment. Tool implementors should be able to formally specify user interfaces of new tools. Such a process for building and extending the user interface should make use of an extensible library of contacts and an extensible library of tools that perform various user interface tasks. The process to build and extend the user interface should translate the formal specification of a tool's user interface into working code. Figure 5.3 displays a dataflow model of the process to build and extend tools that is implemented in coSARA.

Contacts are specified using SARA models. A Contact Specification 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 Dialog Control Module (DCM) Specification can be defined similarly.

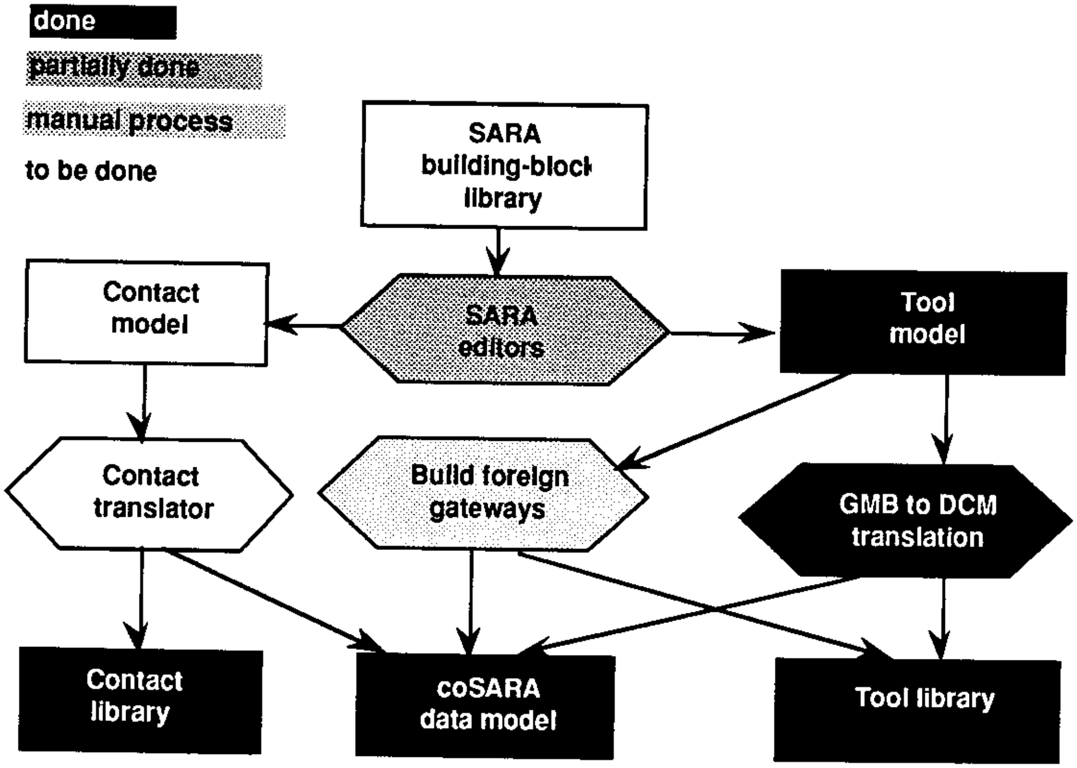


Figure 5.3: Dataflow model of the process to build and extend tools[3]

The Contact Translator processor converts a Contact Specification 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 SARA Model to DCM Translator converts a Dialog Control Module (DCM) Specification 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.

[3] The shading of the elements of this figure indicates the status of the implementation at the time this dissertation was filed.

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 are:

Zoom: provides functionality for management of graphic space in canvases. It presents a view of all graphic objects in a canvas and provides functionality to define a subset of objects to be displayed in one or more canvases. The subset of objects is selected by drawing a box around them.

Selection: allows designers to select an arbitrary set of selected objects that are displayed on a canvas for subsequent operation on them.

Graphic editor: allows one to create and edit graphic objects on canvases.

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, control flow analysis and performance analysis tools) provide a set of analysis tools for our user interface development system.

5.3.3. Specification of user interfaces using SARA

We specify a user interface using SARA models to describe the behavior of a tool in response to user actions. In this section we explain the approach to modeling tool behavior using SARA by means of an example drawn from the coSARA system. We use SARA to model the coSARA zoom tool.

Let us first describe the operation of the zoom tool.

An operational view of the zoom tool is illustrated by Figure 5.4. The zoom tool is part of the library of basic user interface tools.described in the previous section. The purpose of this tool is to manage screen space for graphic operations, providing functionality to zoom and pan. It uses two canvases, which are contacts with graphic capabilities, on which graphics on a world coordinate system (Foley & van Dam [1990]) can be seen using an arbitrary viewport. The SM Editor canvas is used to do graphic editing, This canvas is of type Editor-canvas, meaning that it provides the callbacks and the functionality necessary to support the operation of the selection tool and the graphic editor tool that were described in the previous section; the SM Zoom canvas, of type Zoom-canvas is used to control the viewport for SM Editor. A Zoom-canvas is a type of contact that provides the callbacks and the functionality necessary to support the operation of a zoom tool.

The SM Editor canvas has a button labeled zoom, for popping up and hiding the SM Zoom canvas. The SM Zoom canvas has four buttons, labeled edit, zoom, pop and reset. It also highlights a rectangular region of the screen that defines a viewport in the world coordinate system. The designer can manipulate this rectangle using the mouse. Pressing a button of the mouse sets the upper left corner of the rectangle and dragging the mouse defines the bottom right corner of the rectangle. When the button of the mouse is released, the new rectangular region is highlighted. Figure 5.5 shows a view of the SM Zoom canvas after setting a new rectangular region by dragging the mouse. The edit button is used to set the viewport of the SM Editor canvas to the current rectangle displayed in the SM Zoom canvas (Figure 5.6). The zoom button sets the viewport of the SM Zoom canvas, stacking the previous viewport. The pop button restores the most recently stacked viewport in the SM Zoom canvas. The reset button sets the viewport of the SM Zoom canvas to be the complete world coordinate system and clears the the stack where the zoom viewports are stacked.

The operation of the Zoom tool corresponds to a dialog, that can be modeled using SARA as shown in Figure 5.7. This figure has a number of simplifications intended to improve clarity in the limited space allowed by a single page. Simplifications include not displaying in full the models of Designers, Editor-canvas, Zoom-canvas, View and Buttons. All the contact models are stored in the building block database described before. These models are retrieved

from that database to be used for composing the model of a new system.

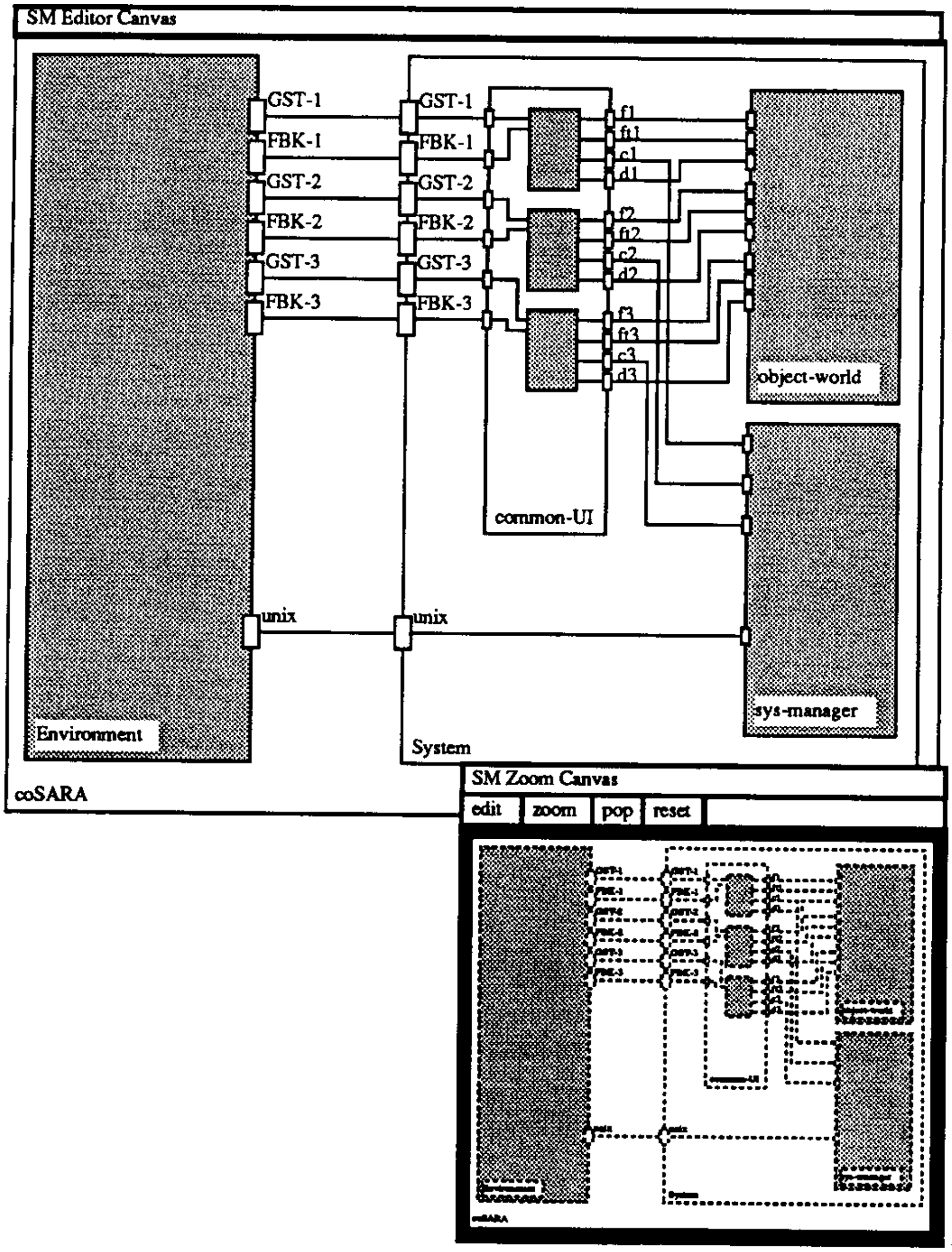


Figure 5.4: Operational view of the zoom

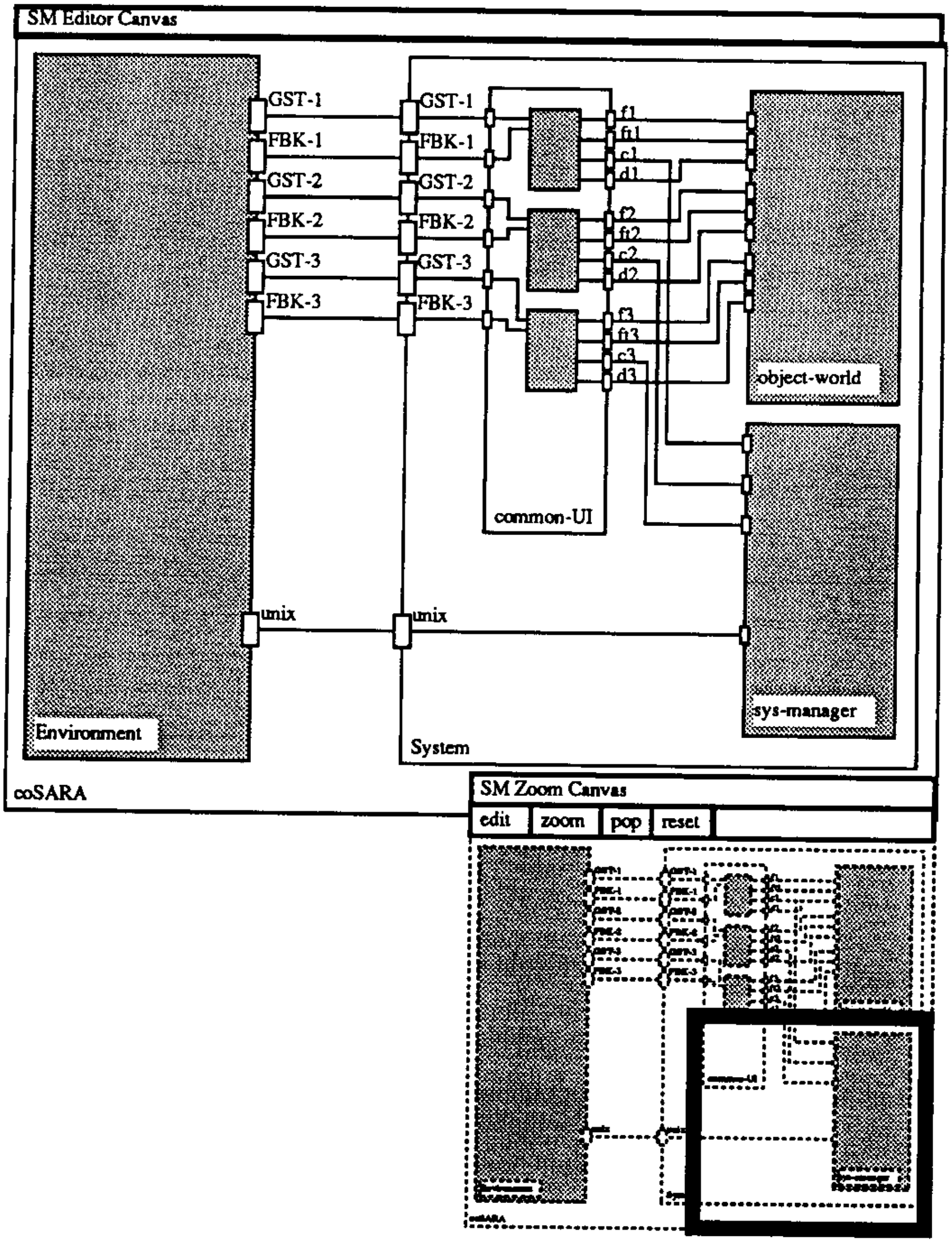


Figure 5.5: Changing the rectangle in SM Zoom Canvas

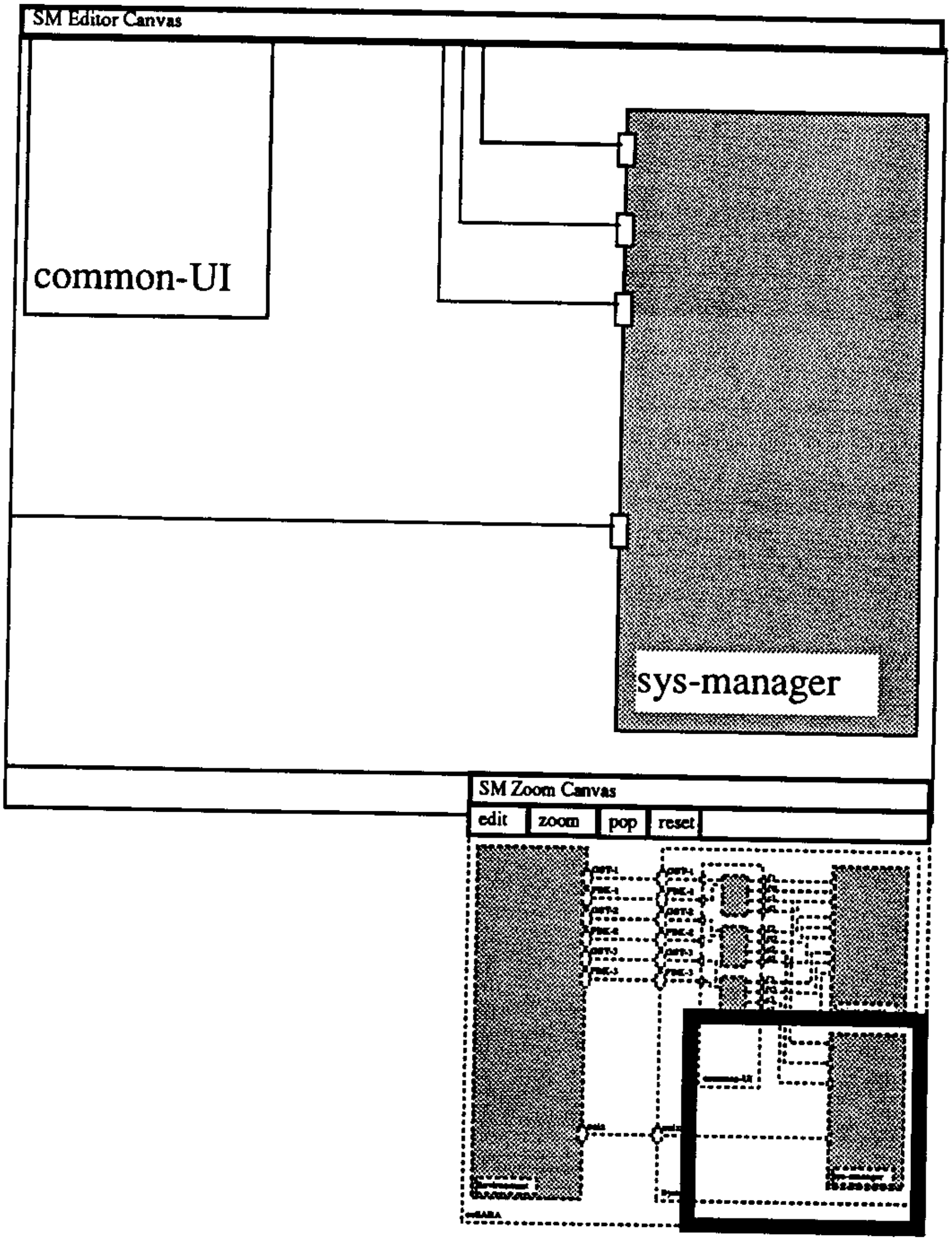


Figure 5.6 Changing the viewport of SM Editor Canvas

Let us understand now the overall structure of the model of the zoom tool. Module Environment, includes a module Designers to represent designers who operate the zoom tool[4]. Designer's actions are accepted by the Editor-canvas and Zoom-canvas contacts. These contacts respond to the user actions applying callbacks. The name of the callbacks that are provided by the contacts are attached to the control arcs of the corresponding modules. In the case of Editor-canvas we only show the behavior to activate and suspend the Zoom-tool module. The Position data arc in Zoom-canvas is used to provide the position of the mouse inside of the canvas, relative to the upper left corner of the canvas.

Module Zoom-tool defines the behavior of the zoom tool in response to the callbacks applied by the contacts in Environment. We have partitioned the Zoom-tool module into two submodules: View-manager and Zoom-manager. The module View-manager provides functionality to change the viewport displayed by the zoom in the zooms' canvas as a highlighted rectangular region. The module Zoom-manager provides functionality to set the viewport of the Editor-canvas or the viewport of the Zoom canvas, actions that are performed in response to clicks on the zoom buttons.

[4] In this case we have chosen to include two designers to illustrate how modeling tools with SARA allows us to model group processes. In this case we have chosen to use a simple baton-passing protocol between two designers.

Let us first focus attention on the processing of press-move-release sequences. Initially there is a token in control arc a1, module View-manager (Figure 5.7-1). When the user presses a button of the mouse, a token is put in the control arc Press of the module View of Zoom-canvas. Because of the structural interconnections this token will be available as input to the node n1. The initial conditions shown in Figure 5.7-1 specify that a token exists initially in the control arc a1. The input logic of node n1 is *, therefore having tokens in both input arcs the node becomes active and the interpretation associated to data processor Start-focus is executed. The procedure followed by StartFocus is to read the dataset Position, which is written from the environment with the current position of the mouse, and set the new upper left corner of the rectangular region, writing this information to the dataset Temporary-focus.

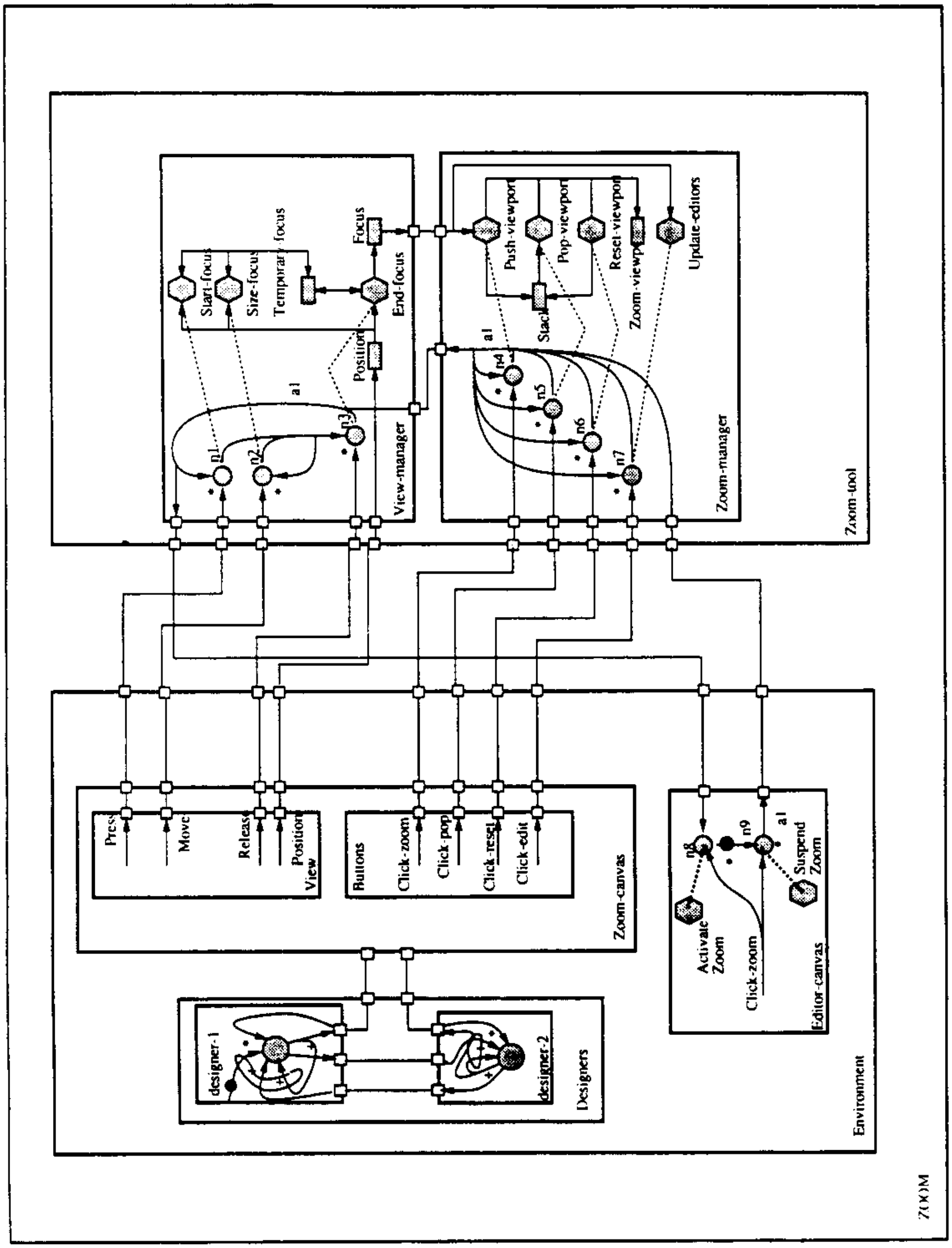


Figure 5.7: SARA model of the Zoom

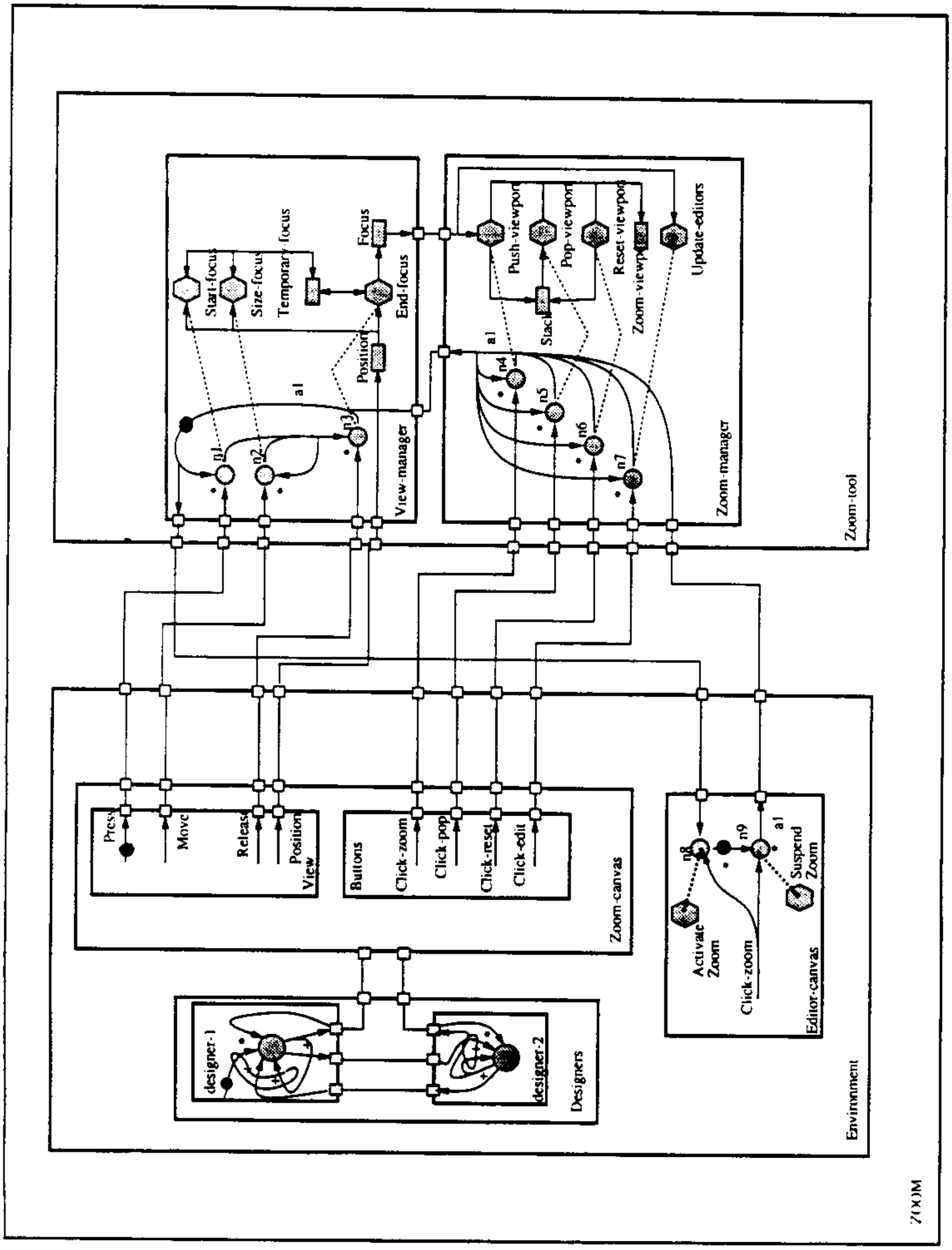


Figure 5.7-1: SARA model of the Zoom (Execution trace)

Upon completion, the control node n1 outputs a token to arc a2. The new state of the control graph is shown in Figure 5.7-2. Arc a2 enables activation of either node n2 or node n3. Moving the mouse after pressing a button will result in a token being put on arc Move (Figure 5.7-4). Existence of tokens on arcs a2 and Move triggers activation of node n2, that in turns trigger execution of the interpretation procedure of dataprocessor Size-focus. Size-focus updates the rectangular area of the zoom and writes this data to the dataset Temporary-focus. Upon completion the node n2 outputs a token to arc a2, enabling either n2 or n3 for activation (Figure 5.7-5). A sequence of tokens in Move, consequence of moving the mouse, will produce a sequence of updates of the rectangular region stored as numerical data in dataset Temporary-focus. When the button of the mouse is finally released, a token will be put in arc Release (Figure 5.7-6), triggering activation of node n3, mapped to dataprocessor End-focus. The interpretation of End-focus defines the new rectangular region of the zoom, storing it in the dataset Focus. Upon deactivation, node n3 will output a token to arc a1(Figure 5.7-7), enabling in this way further activations of node n1 or nodes n4 through n7 in module Zoom-manager.

Now, let us turn to the behavior of the zoom tool in response to the set of buttons of the Zoom-canvas. We will analyze the response to each of: Click-zoom, Click-pop, Click-reset and Click-edit in turn.

When a token is put in the arc Click-zoom (Figure 5.7-8), node n4 is activated in the module Zoom-manager. The interpretation of the dataprocessor associated to n4 saves the current viewport of the Zoom-canvas pushing it into a stack, reads the new rectangular region from the dataset Focus in module View-manager and sets the viewport of the Zoom-canvas to be the area defined by the new rectangular region. Upon completion, node n4 outputs a token to arc a1 (Figure 5.7-9). A token in arc a1 enables either sequences of press-move-release actions or clicking any of the Zoom-canvas buttons.

In the following paragraphs we explain the processing of clicks on the rest of the Zoom-canvas buttons. Since they are all similar to the zoom button, we will not use a graphic representation to follow the process.

A token in arc Click-pop trigger activation of node n5. The interpretation of dataprocessor Pop-viewport is executed, popping the Zoom-canvas viewport stored at the top of the stack (dataset stack), discards the current Zoom-canvas viewport and sets the Zoom-canvas viewport to the value that was retrieved from stack. When node n5 is deactivated it outputs a token to arc a1. A token in arc Click-reset will activate node n6, which is mapped to dataprocessor Reset-viewport. The interpretation of Reset-viewport is to set the stack empty, restore the initial viewport of the Zoom-canvas and display an initial default rectangular region. Upon completion node n6 outputs a token to arc a1, enabling sequences of mouse press-move-release or clicking any button.

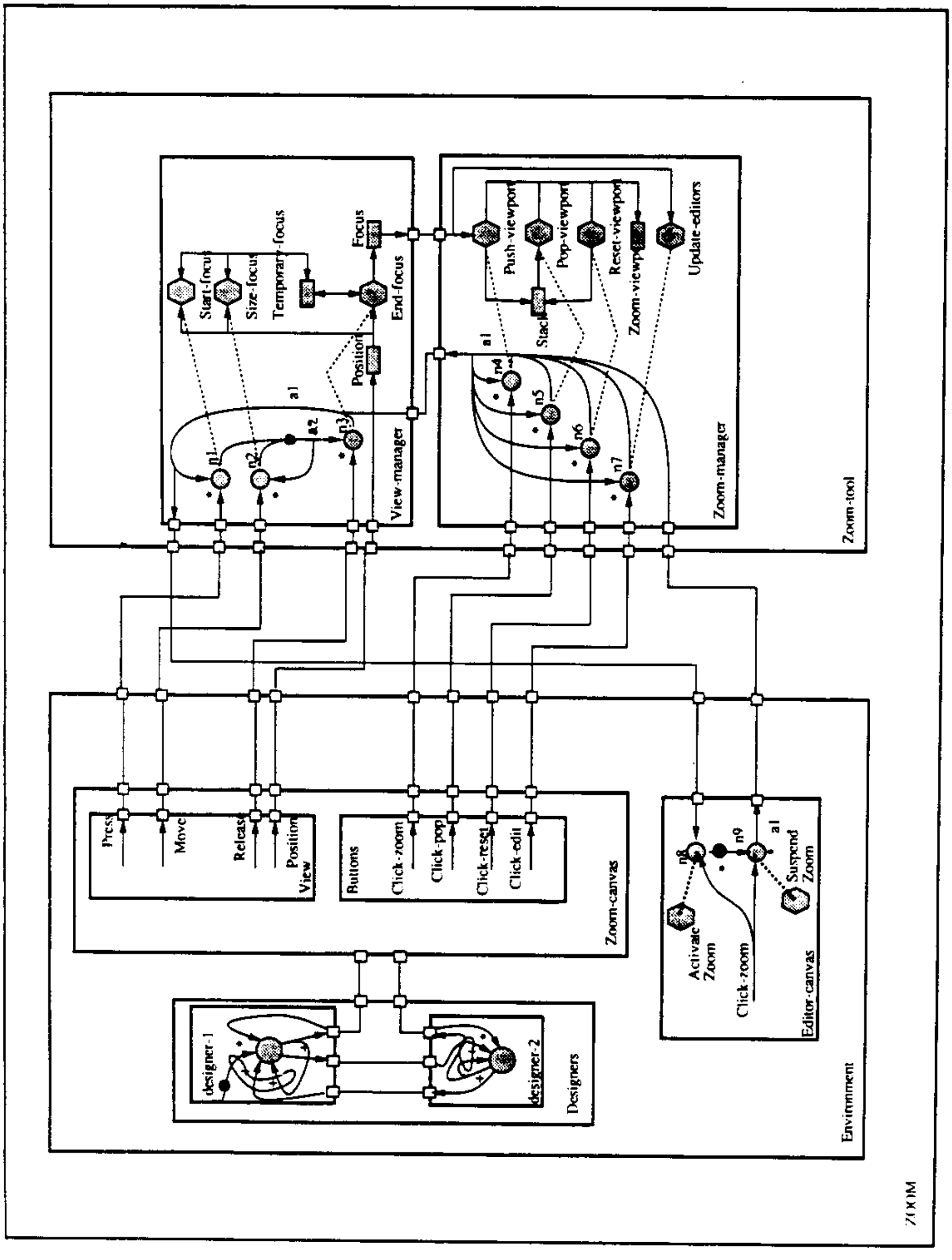


Figure 5.7-2: SARA model of the Zoom (Execution trace)

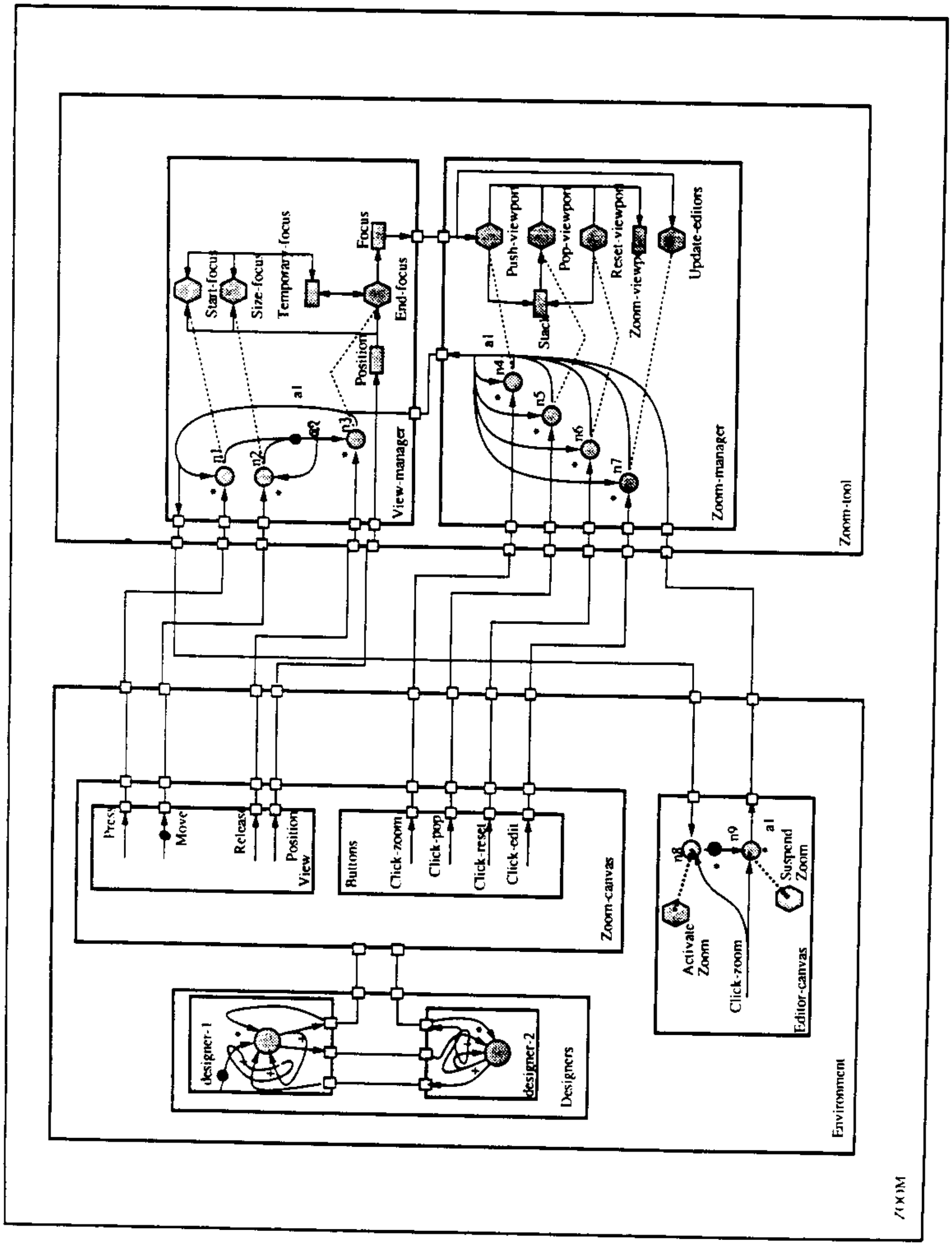


Figure 5.7-3: SARA model of the Zoom (Execution trace)

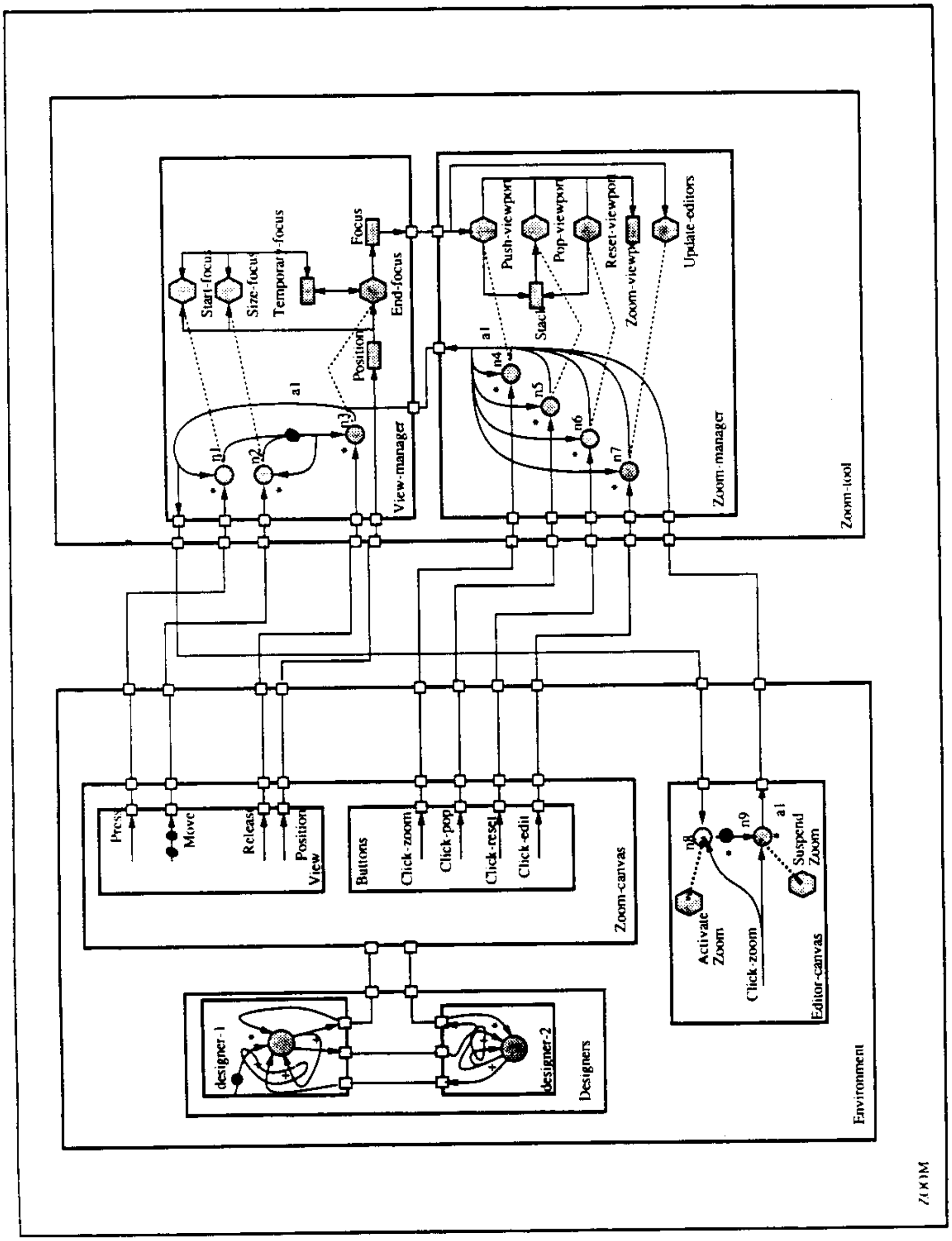


Figure 5.7-4: SARA model of the Zoom (Execution trace)

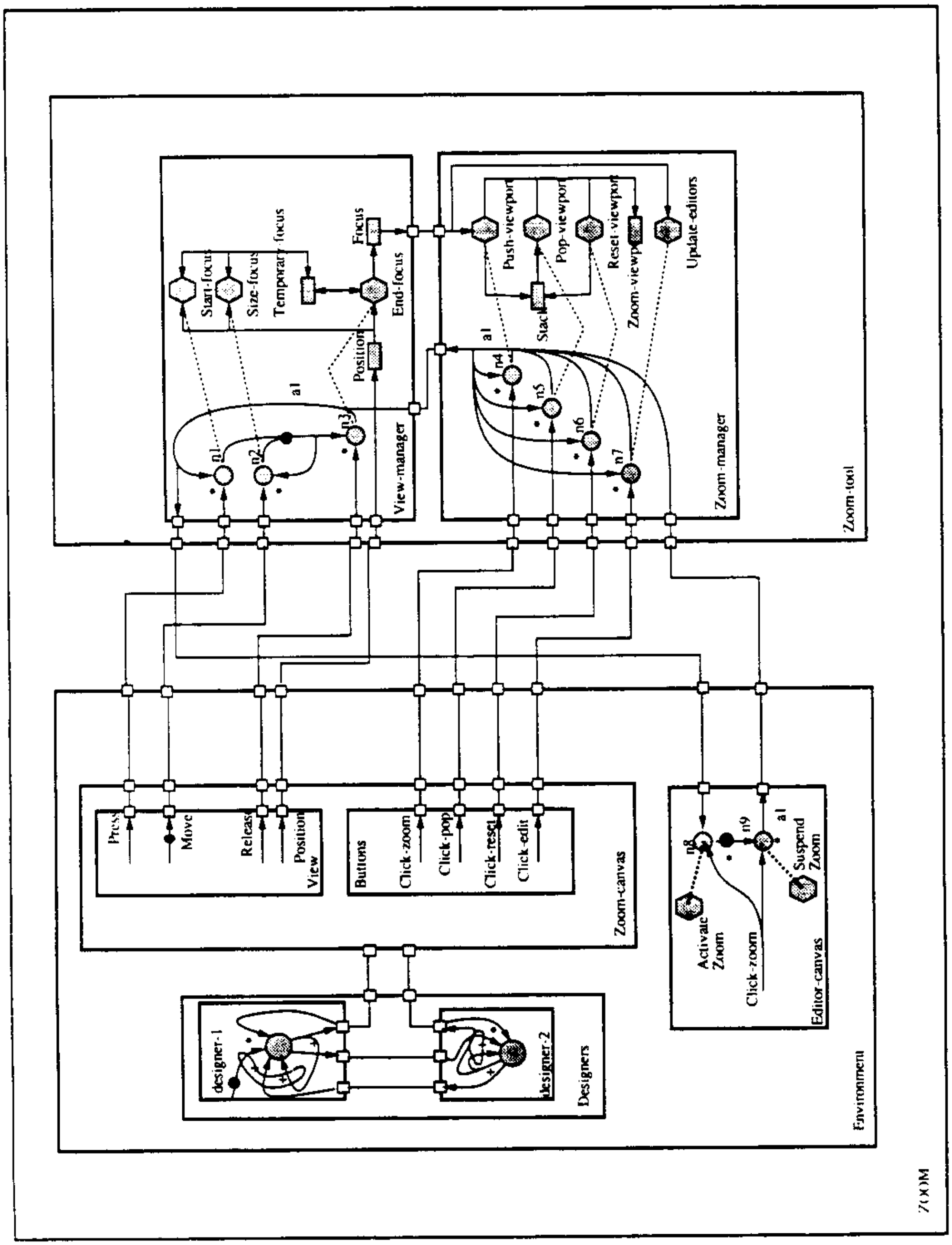


Figure 5.7-5: SARA model of the Zoom (Execution trace)

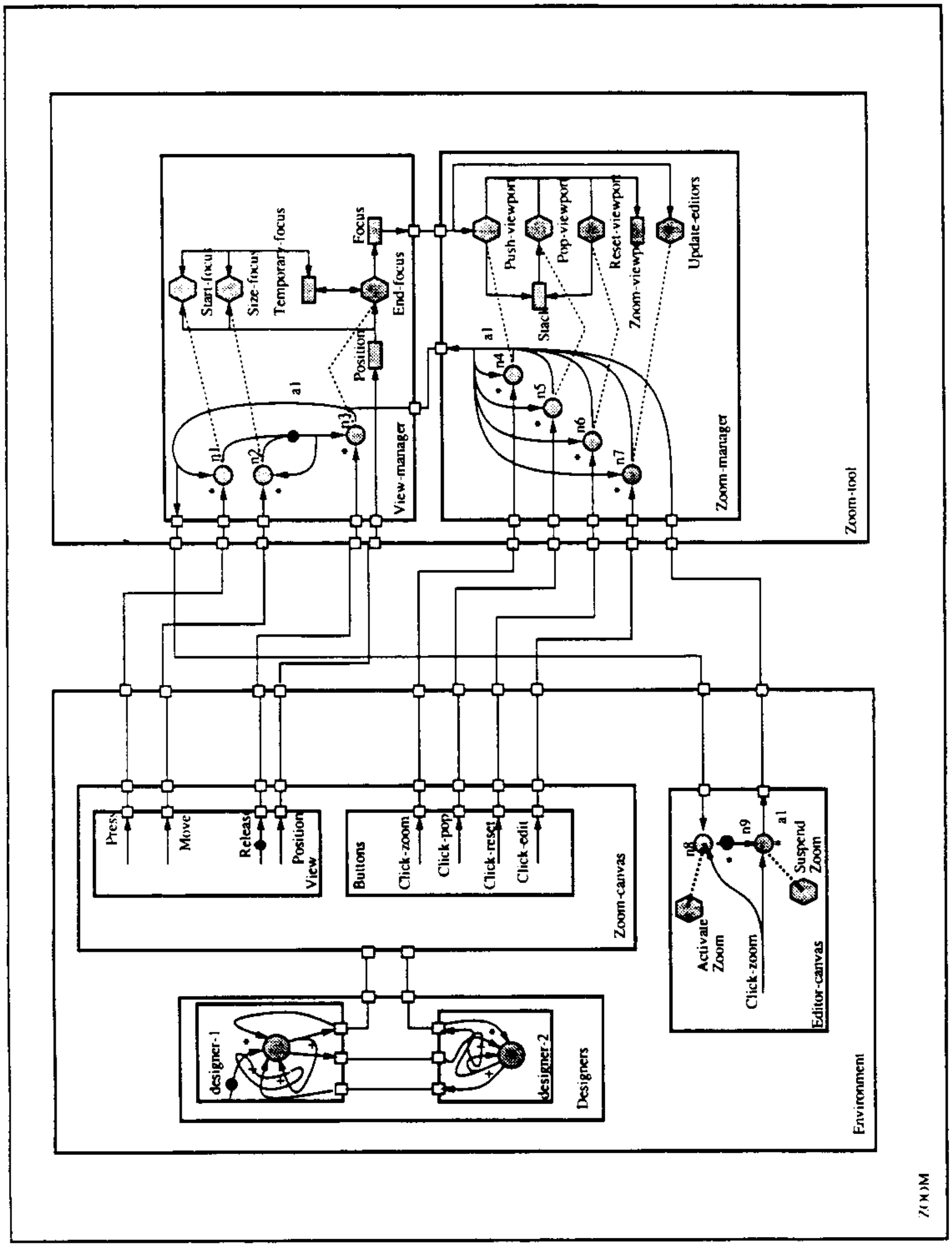


Figure 5.7-6: SARA model of the Zoom (Execution trace)

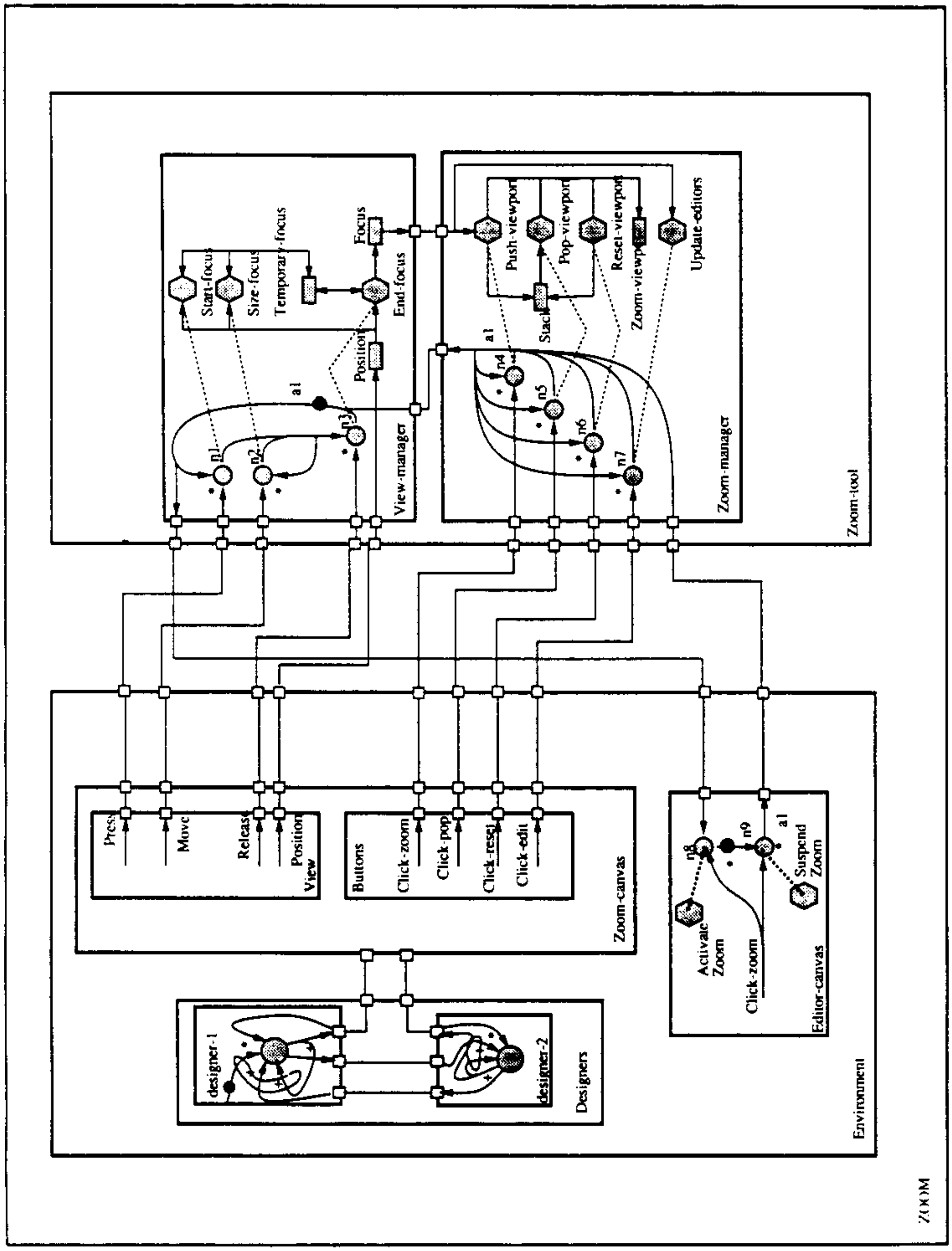


Figure 5.7-7: SARA model of the Zoom (Execution trace)

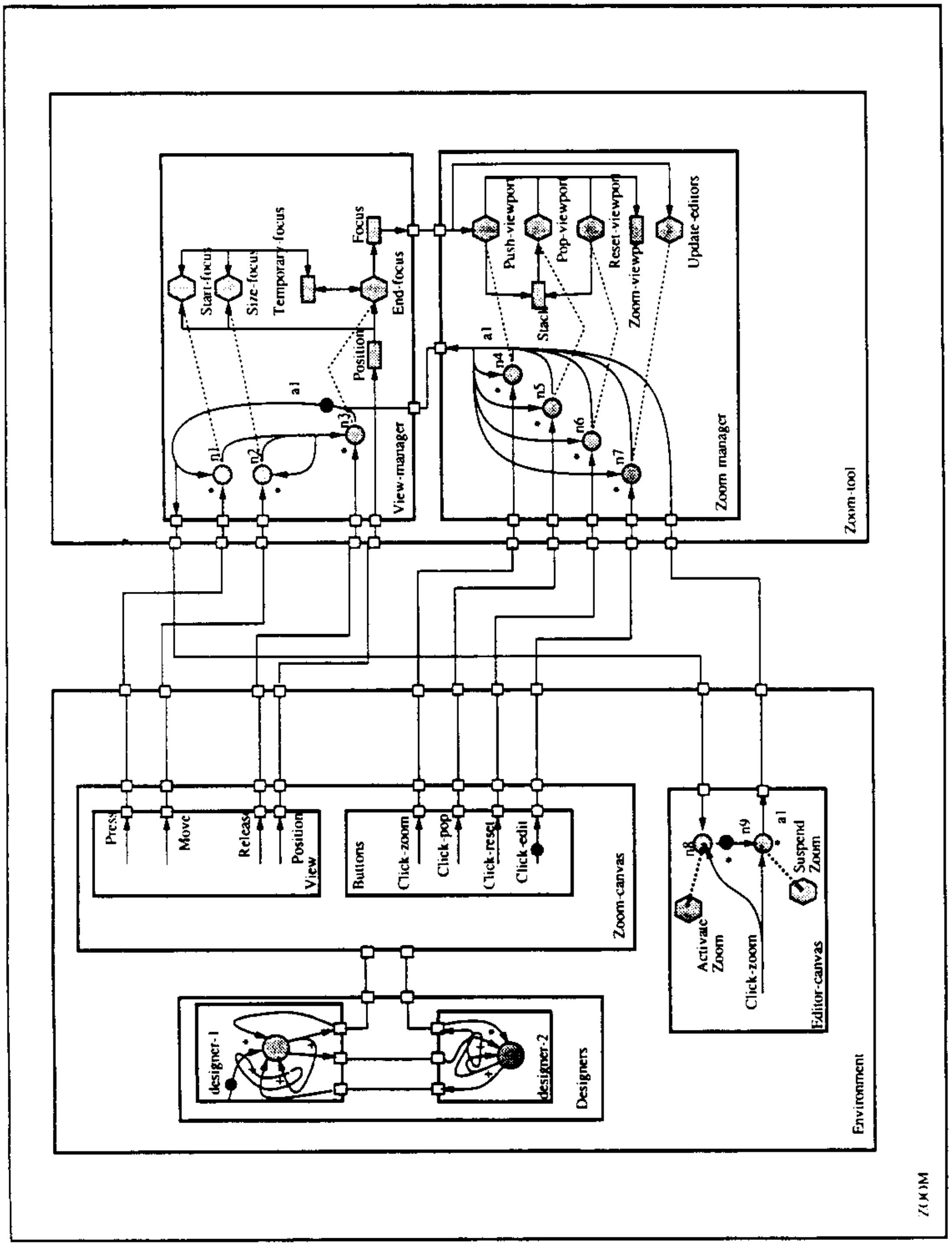


Figure 5.7-8: SARA model of the Zoom (Execution trace)

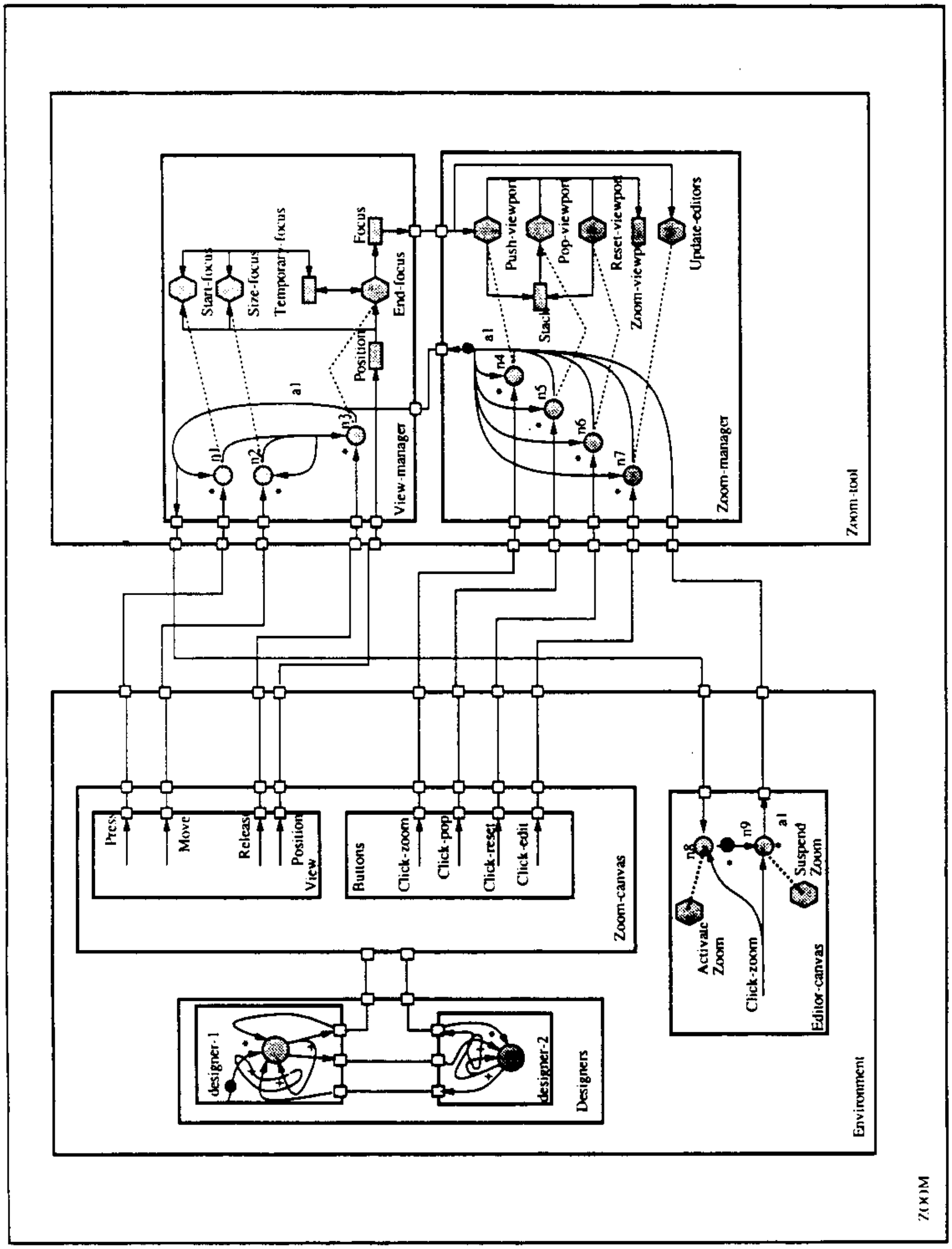


Figure 5.7-9: SARA model of the Zoom (Execution trace)

Finally, a token in arc Click-edit activates node n7 mapped to dataprocessor Update-editors. The interpretation of this dataprocessor is to set the viewport of the editor canvases controlled by the zoom tool to the current rectangular region stored in dataset Focus. As well as n4, n5 and n6, node n7 will output a token to arc a1 upon completion, enabling further action.

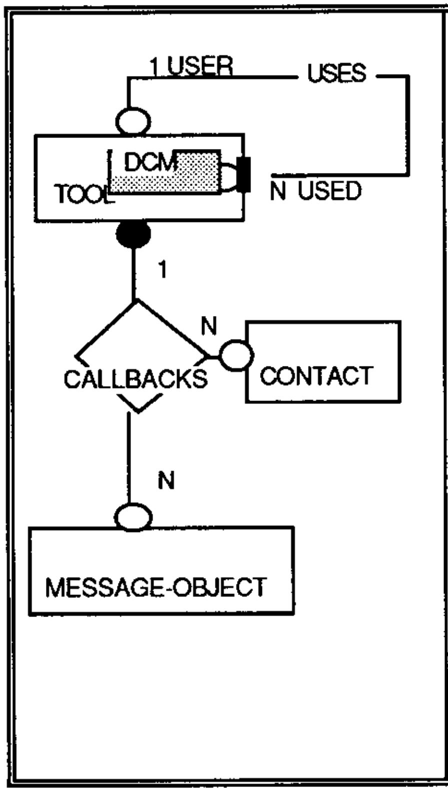
Module designer represents the model of a group process in which to designers share the zoom according to a baton-passing protocol. In the model, designer-1 has control initially being enabled to operate on the zoom. At any time designer-1 may choose to deliver control to designer-2. The behavior of designer-2 becomes then identical to the initial behavior of designer-1.

This example shows how the specification of tool behavior can be done using SARA models. The formal methods and analysis tools allow *early simulation and analysis of such models*, one of the goals of our work. The example also shows how we can use SARA models to incorporate in the tools a *model of group processes* in which the protocols followed by designers to interact are expressed explicitly.

5.4. Data Model

Figure 5.8 shows a data model of tools. TOOL is the class that encapsulates the properties common to all classes of tools. We will refer to the class TOOL as the generic tool class. Specific tools are defined as subclasses of this generic tool and therefore inherit all its

properties. A particular tool class may have its own data slots and participate in other relations as we will show in an example later in this section.



- TOOL objects have a DCM which is an encoding of the behavior of the tool in response to user actions.
- TOOL objects are related to other TOOL objects by the pair USES. There is one USER tool and many USED tools. The mapping labeled used is ordered. The used tools must be installed in the order specified by this mapping.
- MESSAGE-OBJECT is the class that represents the messages used by the TOOL object
- CONTACT is the class of user interface contacts
- TOOL objects, CONTACT objects and MESSAGE objects are related by the a relation object CALLBACKS. The meaning of a CALLBACK relationship is defined in the section on the user interface model.

Figure 5.8: Generic TOOL data model

To model a particular tool we specialize the class TOOL, defining subclasses of TOOL. For example the zoom tool discussed previously is an object whose class is defined as subclass of TOOL. Subclasses of TOOL may then participate in relation 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 mentioned in Figure 5.3.

As an example of such specialization, Figure 5.9 shows the OREL definition of the class ZOOM. The slots of the ZOOM class are derived from its behavioral model by a procedure (GMB to DCM translation) that we will explain in the following section.

5.5. Tool integration procedure

The following is the integration procedure for fully integrated 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.

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.

3. Process the tool model specifications, both OREL and SARA, to produce working code to be included in an operating environment.

The following is the integration procedure for partially integrated tools:

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. Specify the Dialog-gateway using SARA models. Process this gateway to obtain code for dialog management.

3. Build the required Contact-gateway, Dialog-gateway and Data-gateway.

The condition that any foreign tool has to meet in order to be partially integrable into the coSARA environment is that the above mentioned gateways 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. We have also shown the existence of a *procedure to integrate a foreign tool into the coSARA environment*. We will show an example of how we have achieved integration of text editors into the coSARA environment.

5.5.1. GMB to DCM translation

This section describes the rules for deriving a DCM from SARA model. Appendix I includes the code generated for the zoom tool whose specification was discussed in previous sections (Figure 5.7).

From the SARA model used to specify the Zoom tool the following information can be extracted:

- The Zoom-tool uses an Editor-canvas and a Zoom-canvas.
- The Zoom-tool does not use any other tool.
- The Zoom-tool uses the zoom callback provided by the Editor-canvas, the Click-edit, Click-zoom, Click-pop and Click-reset provided by the buttons of the Zoom-canvas and the Press, Move and Release callbacks provided by the Zoom-canvas View contact.
- The Zoom-tool obtains the value of the dataset Position from the Editor-canvas.
- The flow of control is simple, there is no concurrency and each callback results in a single control node activation.
- The Zoom tool class contains a rectangle, a stack and a list of Editor-canvases.

Using this information a procedure to initialize a Zoom tool is generated by the SARA Model to DCM Translator. An OREL data

model of Zoom-tool objects is also generated as shown in Figure 5.9. In this figure the class Zoom-tool is defined to be a subclass of Tool. The data slots of the class Zoom-tool correspond to the datasets in the SARA model of the Zoom-tool.

The GMB to DCM translation procedure is the basis that sustains the feasibility of incremental user interface extension.

5.5.2. A fully integrated tool

In this section we show how our tool integration procedure is used to integrate the zoom tool into the coSARA system.

The first step is to modify the coSARA datamodel to include an OREL model of the zoom tool (Figure 5.9). This modification of the coSARA datamodel is purely additive and no further consideration of its effect on other tools is necessary. In the following steps we store the zoom model (Figure 5.7) in a building block library. We process the zoom model using the GMB to DCM translator to obtain code that we place in the tool library.

The Zoom-tool object is a subclass of the Tool class shown in Figure 5.8 and it inherits all its properties. The Zoom-tool class provides specialization in support of the tool's behavior whose model is shown in Figure 5.7. The slots of the Zoom-tool class correspond to the datasets defined in the SARA model of the zoom defined in Figure 5.7. Their exact meaning is defined by the behavioral model of the Zoom-tool shown in Figure 5.7.

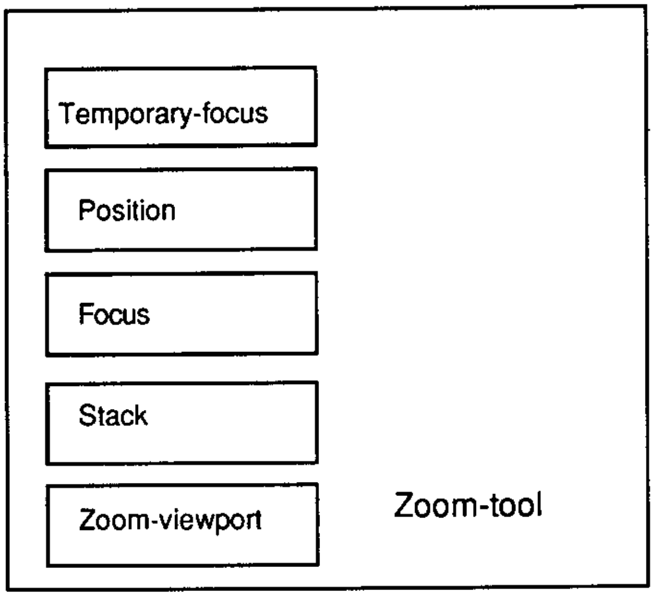


Figure 5.9: Data model of the Zoom-tool class.

The slot Temporary-focus contains the position of the mouse and the size of the rectangle while it is being dragged to define a new rectangular region for the zoom. The slot Position contains the screen coordinates of the mouse. The slot Focus contains the upper left corner position, width and height of the current rectangular region defined for the zoom. This is the region defined by the last complete sequence of press-move-release actions done with the mouse on the Zoom-canvas contact. The slot Stack contains a list rectangular areas that are stored when the viewport of the Zoom-canvas is changed. The Zoom-viewport slot contains the current viewport of the Zoom-canvas.

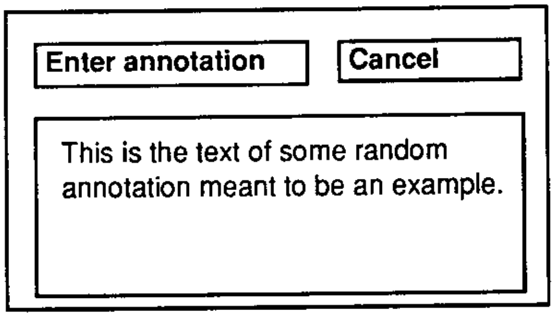
Having the OREL data model of the zoom tool built into the coSARA data model and the code derived from its SARA model in the corresponding libraries completes the integration procedure.

When the sys-manager (Figure 5.1) receives a request to install the zoom tool, it searches the libraries and installs the code generated by the GMB-to-DCM translator in the Dialog manager and the Dialog manager directs the tool-proxys module to create a proxy for the zoom tool object

5.5.3. A partially integrated tool

One of our contact classes, dialog-box, requires collection and editing of text. Whereas the contact provides some basic editing capabilities, such as erasing characters, it is often necessary to resort to more powerful editing functionality. For this purpose we have partially integrated GNU Emacs (Stallman [1981]) in coSARA.

Figure 5.10 shows a dialog box. When a control character is input in the text area, an Emacs window pops up with the contents of the dialog box displayed in an Emacs buffer. The designer can edit this text using Emacs and upon exit the text is displayed again in the dialog box. Not all the functionality of OREL objects is available while the user operates in Emacs. Updates to the text of the dialog box can not be seen from other workstations until editing is complete.



This is the text of some random annotation meant to be an example.

Figure 5.10: A dialog box

To partially integrate Emacs we built the gateway modules shown in Figure 5.2: CN-gate, OW-gate and DM-gate. These gateways are implemented as follows:

CN-gate: this gateway is null because Emacs does not interact with any coSARA contact for its own input/output.

OW-gate: this gateway has two parts one that extracts the text of the dialog box as a string and writes the string to a file for editing; the second part restores the value, reading it from the file written by the first part after edition is complete. These parts are implemented as simple ComonLisp functions.

DM-gate: this gateway is implemented as a lisp function that responds to a callback from the dialog manager by sending a message to the UNIX system that runs Emacs waits for its completion. When Emacs completes it sends a message to the Emacs proxy to update the dialog box object with the new text collected from Emacs.

Dialog specification is trivial since only one gesture of the user (typing a control character) must be specified. No modifications to the data model are required by this tool.

5.6. Related work

In this section we review Important related work in the area of integrated environments and user interface development systems.

5.6.1. Software through Pictures (StP)

StP (Wasserman & Pircher [1987]) is an environment that embodies a multitude of tools for software engineering based on structured design (Myers & Stevens [1974]). The system utilizes a powerful graphical interface that allows graphics editing of models of the system being analyzed or designed in data and control domains. The system is aimed at generation of substantial amounts of code from the models of the system built using the tools.

StP operates using the TROLL relational database system as a basis. It supports the rapid prototyping of user interfaces using RAPID/USE that is described in the section on user interfaces and provides facilities for integration of new tools.

The dialog specification language of RAPID is based on an RTN, which is represented as a diagram that consists of two major parts:

1. Nodes: the specification of nodes consists of a list in which the output associated with each node is specified. This output is strictly textual.

2. Arcs: The specification of arcs is done by listing all the nodes and for each node listing all the transitions that go out of that node. A transition specification is composed of a condition and an action.

Conditions used in transitions may be one of:

Empty: is true when a single carriage return has been input.

Skip: indicates that no input is to be read.

Else: else is true when no other condition is true.

Alarm: this condition becomes true when no user input has been received in the indicated amount of time.

On: if the parameter is a string, it is true when the user inputs the parameter string. If the parameter is a variable, the next input is read into that variable.

Key: takes a single character as parameter. Is true if the when the user hits the key that corresponds to that character in the keyboard.

Actions specified in transitions are of two classes: a transition to a new state or to a subdiagram, and invocation of application routines.

An application routine may be specified by name with both value and reference parameters. A multi-way branch may be done based on the value returned by the application routine, which must be an integer.

The system is implemented using a Transition Diagram Interpreter (TDI) that walks the graph of the state transition diagram, and an Action Linker which links the TDI to action routines written in some programming language (like C, Fortran, etc.) RAPID also provides access to a database system directly from the state transition diagram or form the application routines.

RAPID is a complete UIDS; it is sufficiently powerful to fully support the class of interfaces to which it is targeted. It has been used to implement several ``real'' interfaces of commercial product quality for the StP software.

Recently Wasserman & Pircher [1989] presented an object oriented, structured design method for code generation, whose integration in the StP environment is claimed to be in progress.

The method leans on a graphical language to specify classes and objects, in which operations are denoted explicitly. The notion of information cluster is introduced, matching the classical concept of object and class, as a module that encapsulates data and behavior. Use of a cluster by some other module is then represented by a connection between the user module and the representation of the

operation that is used in the cluster. Only those operations that are actually used are represented graphically.

The model includes primitives to support features such as separate compilation of lexical units, clearly targeted in Ada.

Inheritance is treated from the point of view of data generalization, resembling generic packages in Ada. The design method also accounts for the asynchronous activation of a module, supporting fork/join synchronization and message passing, as well as Ada's rendezvous.

5.6.2. Arcadia

The goals of the Arcadia project are: the discovery and development of environment architecture principles and creation of novel software development tools (Taylor & Osterweil [1986]).

The term architecture is used to denote the set of rules and support infrastructure which characterize, bind together and enable utilization of the software development support tools existing within an environment. Object managers, user interface tools and tool activation managers may all be elements of an environment architecture.

Two principal qualities sought in the development of Arcadia are: extensibility, which refers to the ease of adding new capabilities to the environment; and integration, which refers to consistent user

interfaces, easy context switching, efficient communication between tools.

The tools and objects manipulated in Arcadia are classified in three broad categories:

Basic components:These include the internal representation for programs, suitable for compilation, interpretation, analysis and program transformation.

Tool-building tools: These include tools such as lexer and parser generators.

Analysis tools: These include testing and debugging tools, design analysis tools and other tools applicable in pre-implementation stages of software development.

The process of developing software in Arcadia, depends on the process of creating, organizing, augmenting and exploiting a collection of persistent objects and aggregates of information. Arcadia users are encouraged to think of their work in these terms. Taylor & Osterweil [1986] give the following example':

> "...rather than requesting the execution of a program, the user will request the display of the output of a program as applied to a specified set of data."

Objects in Arcadia are typed and are organized in a structure known as the Object Derivation Graph (ODG), which defines how objects are

derived from other objects. This organization is compared to that of RCS (Tichy [1982]), a system for version control. Hierarchy is also used to organize the object store. Users may define arrays or structures of objects that may be in turn organized as arrays and structures.

Arcadia is designed to support cooperative activities of teams of software developers and maintainers, working in separate workstations connected in a network. Each worker has a separate store of persistent software objects in his/her own workstation and also has access to objects stored in different workstations. Sharing of objects is done using the principles of Software Federation (Mcleod [1985]).

In Arcadia a tool is a collection of tool fragments, temporarily allied, under the control of the environment, to complete some activity. For example, the tool fragments may be connected in a Unix pipeline. This notion contrasts with the more conventional concept in which a tool is more or less equivalent to a single program. Arcadia tools may be either passive or active. Active tools are executed without direct invocation by users. They perform according to predefined plans, and are invoked by Arcadia using devices such as timers and daemons that watch for relevant changes in the object store.

Creating larger tool capabilities out of smaller more general tool fragments is beneficial because, if the tool fragments are well chosen they will be usable for composing a variety of larger tools, at a lower

cost. For example a pretty-printer can be composed using pieces such as a parser, a lexer and a formatter.

In many cases it is possible to determine which tool fragments will have to be invoked and in which order to accomplish a given task. However, there are cases when this is not possible. For example, consider a two-pass dataflow analyzer, that determines the order of analysis of the second pass during the first pass. The design of Arcadia includes a planning tool fragment whose job is to dynamically create tool fragment invocation sequences.

Arcadia tools are objects too, and there are tools to manipulate them. Using this approach a method for incorporating new tools is defined. It is required that at least one tool to create instances of new tools be written initially.

Inter-tool communication is done by remote procedure calls. When tight integration is desired, tools must share a set of common data structures, which are implemented as monitors.

5.6.3. SARA/IDEAS

The SARA/Ideas User Interface Management System (UIMS) was designed to be part a part of the system to support integrated environments for computer aided design of computer systems (Worley [1986]; Landis [1988]). The SARA/Ideas UIMS shows a strong correlation with the method proposed by Worley to create and integrate new tools in the design environment. The UIMS that

supports Worley's interface proposal has provisions for integral help, device independence and bidirectional execution.

The process of creating a new tool is divided into four phases:

Conceptual phase: the first step in which the tool designer identifies the objects upon which the tool will operate, their relationships and the operations that the user of the tool will apply to these objects.

Syntactic phase: in which the tool developer is concerned with the tool's user interface. Factors such as: the syntax of users' inputs, semantic response to inputs, object representation, screen layout, etc.

Logical device phase: in which the tool designer focuses attention on the devices that will be used to interact with the user[5]. An example of a logical device is a ``pick-device'' used to "pick" objects that are represented on the screen.

Physical device phase: in which access to the physical devices for interaction that are available is provided together with a binding of logical to physical devices.

[5] Worley's concept of logical device seems to be similar (if not equivalent) to the now common notion of interaction technique.

This phase is implemented by a method based on the use of Augmented Transition Networks (ATN). The dialog syntax is specifies using an LL(1) grammar, which is translated into an equivalent ATN. The user interface is then driven by an ATN interpreter. The processor that performs this translation is called the Grammar Compiler. Worley, however, envisioned that a more direct representation than that offered by grammars was convenient:

> "It is conceivable and perhaps desirable that instead of writing a linear representation of the syntax of a tool, a tool with graphical interaction could be offered to the syntax definer This tool could allow the direct expression of ATNs in a graphical form and perform that LL(1) analysis upon user request..."

The logical and physical device phases also make use of special purpose languages for description of devices available to the tool designer, that allow the construction of interaction tasks (Foley & Van Dam [1990]) and providing various libraries to implement the mapping of device dependent information to device independent procedures used at the higher levels of the interface.

In general, Worley's proposal was aimed at introducing an indirect style of interaction, based on the metaphor of a dialog or conversation---contrasting with the concept of direct-manipulation interface (Schneiderman [1983]).

5.6.4. Sassafras

The major contributions made by (Hill [1987]) in the development of SASSAFRAS are: support for multiple concurrent dialogs (including synchronization and communication), and a technique to specify dialog syntax that deals with localized concurrency.

System responses to user inputs are specified as Event-Response Systems (ERS). Communication among and synchronization of the modules that make up the interface are achieved using the Local Event Broadcast Method (LEBM). Both notions are explained below:

ERS are language recognition automata that have a power equivalent to that of finite automata (i.e. they recognize regular languages.) An operation denominated concurrent-composition is defined for the class of regular languages, and is proven to be closed on the regular languages (Hill [1987], Appendix A). Thus it is possible to write independent ERS specifications of concurrent dialogs and then merge them using the concurrent-composition operation. The advantage over finite state automata claimed by Hill is that ERS do not radically increase in size when dealing with concurrent dialogs.

A practical dialog specification can not be directly specified in ERS. A specification language, called Event-Response Language (ERL), has been developed around ERS, adding output and a few other features. The main elements of ERL are incoming events, outgoing events, and flags. An event is a signal that something has happened

and may carry data relevant to the event. Events are the only form of I/O available. Flags are local variables used to encode the state of the system.

An ERL specification consists of a list of rules. Each rule specifies either the response to some external event or an action to be taken when some state is entered.

The LEBM is a run-time structure that supports communication and synchronization among the modules that constitute an interactive system and schedules their execution. LEBM works on modules and clusters. A module is a component of the user interface that exchanges information and synchronizes with other modules via the LEBM event passing mechanism. Each module does only one of: input/output, dialog control, application routines. Clusters are groups of modules linked by a single instance of the LEBM structure. Although, it may be expected that one cluster per interface exists, it is reasonable to believe that a large and complex interface would be partitioned into several clusters. LEBM allows modules within a cluster to communicate sending events. This event mechanism is similar to asynchronous, non-blocking message passing.

Besides implementations of ERS, ERL and LEBM, SASSAFRAS contains an interaction module, icon libraries, and an interface assembler.

5.6.5. UIDS

The User-Interface Development Environment (UIDE) is a knowledge based system to user interface design and implementation (Foley et al. [1989]). It utilizes a knowledge-based representation of the interface's conceptual design based on an object-oriented data model and on an operation-oriented control model.

The UIDE knowledge representation consists of:

- The class hierarchy. [6]
- Properties of objects.
- Actions that can be performed on the objects.
- Units of information required by the actions.
- Preconditions and postconditions for the actions.

The knowledge base has several uses: Producing a description of the design in IDL (Foley et al. [1989]), checking the conceptual interface design for consistency and completeness, transforming the knowledge base into a functionally equivalent interface via a set of transformation algorithms, providing input to the interface, evaluating the interface speed of use using a keystroke analysis model, and automatically generating run-time help.

[6] Only simple inheritance is supported} of the objects in the system.

The design specification of an interface is done in three major definition stages:

1. Building the design knowledge base.

2. Checking it for completeness.

3. Analyzing it for consistency.

Completeness checks verify that all frames in the knowledge base contain enough information for the transformation system to operate and for the Simple User Interface Management System (will be described in following paragraphs) to implement the interface. Consistency checks examine the overall design knowledge base, for the existence of potential inconsistencies.

It is typical that in the design phase several conceptual models of an interface are evaluated, Using transformations, UIDE can automatically generate alternative designs that are slight variations of one another. Several generic design paradigms have been implemented, including: factoring, establishing a selected set as a generalization of the selected object concept, establishing initial defaults, specializing and generalizing commands based on object and command hierarchies, modifying the scope of some types of commands. Foley et al. [1989] present an example in which an interface that operates on a single selected object is transformed into another that operates on a selection set.

Using transformation algorithms, many functionally equivalent designs for an application can be produced. Dialog syntax, presentation style and interaction techniques are ignored so far. The Simple User Interface Management System (SUIMS) is used to define this information. A set of definitions done through the SUIM is used across sets of functionally equivalent interfaces obtained by transformations. SUIMS follows the following procedure:

1. Establish and update the screen layout.
2. Check all preconditons and recognize enabled actions.
3. Accept the action the user has selected.
4. Process each parameter according to its kind.
5. Access parameter values.
6. Confirm or cancel actions.
7. Execute actions.
8. Evaluate postconditions.

UIDE uses the conceptual design and SUIMS runtime knowledge base to generate context sensitive help messages about command semantics and explaining why a specific command cannot be used at some specific time.

5.6.6. Summary

None of the work we have reviewed achieves interactive sharing of data objects and in particular of user interface objects. None of the user interface methodologies reviewed deals with the formal specification of behavioral response to concurrent actions by multiple users.

5.7. Review of contributions and directions for future work

5.7.1. Review of contributions

The main contributions of this research in the area of tool modeling and integration methodology are:

- The coSARA tool model for environment extensibility and interactive tool sharing: allows partial and full integration of tools; enable tools to operate on interactively shared objects; and supports incremental extension of the environment (section 5.5).

- The coSARA user interface model and user interface development system: allows sharing of interaction mechanisms; provides support for expressing behavioral response to multi-user actions and enables early testing and analysis of user interfaces (section 5.3).

Our research has produced a methodology for modeling and integration of tools that is not found in other systems. The approach

is to allow varying degrees of tightness in the integration of a foreign pre-existing tool in order to deal with conflicting forces that arise from the integration vs. extensibility issue. Such varying degree of tightness is enabled by the existence of the gateway modules shown in Figure 5.2. The methodology 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. 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.

Our research in user interface methodology has resulted in a unique approach: to formal modeling of user interfaces that are aware of multi-user actions, to allowing true semantic sharing of user interface objects and to allowing modeling of group processes into the user interface (section 5.3). 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 (section 5.3). 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 (Chapter 4 and section 5.3.2).

5.7.2. Directions for future work

Our research work has unveiled a number of problems for future research. Below we describe these problems and suggest directions for approaching them.

There is a mismatch between SARA modeling power and the need to model dynamically changing structures. The case when the structure of the system changes dynamically arises frequently in user interfaces. For example take the case of linking a new canvas to an existing zoom tool. Since the environment of the zoom must contain a model of all the contacts and other tools that are used by the zoom, it becomes necessary to alter the structure of the environment module of the zoom.

We have adopted a pragmatic solution that is not quite satisfactory. There is a privileged process that knows how to alter the structure of a running system. This privileged process is foreign to the SARA modeling methodology. This is explained in section 5.3.1.

It is necessary to search for alternative modeling techniques that allow modeling of dynamically changing structures. We believe that a reasonable approach is to define mechanisms that will alter structures dynamically by application of proper refinement.

Although we have a procedure for partially integrating foreign tools, The method is manually intensive because it is necessary to program the data gateways that enable the foreign tools to share OREL with

other tools and it is also necessary to build the callback gateways that enable a foreign tool to share the common user interface. We still seek ways for automatic generation of gateways to support integration of foreign tools.

There are many similarities between tools and contacts at certain levels of abstraction. It is still an open problem to determine good abstractions to design and build the kind of user interface that we have developed. We believe now that contacts should be assimilated tools. This would provide a smoother interface to the user interface and tool designer.

SARA modeling methods and tools provide powerful formalisms for specifying tool behavior and tool user interface. However, there is an intensive programming phase in creating the layout of individual contacts and of the user interface. A graphic tool to aid the user interface builder to specify these layouts would be a great asset.

CHAPTER 6

Conclusions

This Chapter reviews the contributions of this research, discusses future work and ends with a summary statement.

6.1. Review of contributions

The central contribution of our research is the creation of coSARA, a computer-based environment for collaborative design. The creation of coSARA is a proof of concept for collaborative design environments.

coSARA is an integrated software environment that supports interactive sharing of design objects, and can be incrementally extended. The foundation that makes the existence of coSARA possible is constituted by two elements: OREL, an object oriented system for distributed, interactive sharing of data objects which is discussed in Chapter 4; and a tool modeling and integration methodology which is discussed in Chapter 5. We summarize the contributions of research done in these two topics below.

The main contributions of our work on OREL are:

- A distributed system in support of interactive sharing of data objects with persistent storage (section 4.3 and section 4.4).
- An object-oriented graphic language for modeling data, that includes: classes and multiple inheritance, recursive composition of objects, relations as first class objects and integrity constraints on relations (section 4.2).

- A programming interface that allows manipulation of distributed, interactively shared data objects (section 4.2 and Appendix II).

Our data modeling language makes use of relations as a way to increase expressiveness and to enable management of constraints. Unlike previous work, OREL relations are first class objects. Recursive composition also provides an effective way to manage complexity of design data.

The capability to interactively share data objects is the key notion that enables collaborative work by a group of designers. Distributed operation is essential if the system is to support a group of collaborating designers, that will each be operating the system from his/her own workstation. Persistent storage of data objects is necessary because the amount of data involved in a design task is such that it is not possible to build the data from scratch when starting a session and it is necessary to keep track of work progress across session boundaries.

We have used OREL to model SARA objects (Figure 4.12). The model that we produced is quite compact (fits in one page) and enables us to deal with the complexity of SARA objects. In particular we would like to highlight the modeling of modules as recursive composites and the use of directed pair relations in modeling GMB objects, where the hyperarc structure of the control graph is of great complexity. We have translated this model following the procedures

described in Chapter 4 and produced code that is currently used in our implementation of coSARA (the code is shown in Appendix III).

The main contributions of this research in the area of tool modeling and integration methodology are:

- The coSARA tool model for environment extensibility and interactive tool sharing: allows partial and full integration of tools; enables tools to operate on interactively shared objects; and supports incremental extension of the environment (section 5.5).

- The coSARA user interface model and user interface development system: allows sharing of interaction mechanisms; provides support for expressing behavioral response to multi-user actions and enables early testing and analysis of user interfaces (section 5.3).

Our research has produced a methodology for modeling and integration of tools that is not found in other systems. The approach is to allow varying degrees of tightness in the integration of a foreign pre-existing tool in order to deal with conflicting forces that arise from the integration vs. extensibility issue. Such varying degree of tightness is enabled by the existence of the gateway modules shown in Figure 5.2. The methodology 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. 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.

Our research in user interface methodology has resulted in a unique approach: to formal modeling of user interfaces that are aware of multi-user actions, to allowing true semantic sharing of user interface objects and to allowing modeling of group processes into the user interface (section 5.3). 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 (section 5.3.3). 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 (Chapter 4 and Section 5.3.2).

6.2. Future work

This section reviews problems that our research has identified as worth to solve. We outline directions for attacking these problems.

6.2.1. Problems related to object-oriented systems

Currently, OREL operates correctly on a single local area network. This is because we take advantage of built-in facilities for

broadcasting messages, that guarantee that every site will see the same sequence of messages. If this feature is removed, then sites on different local area networks might see different sequences of messages. We know that there is a need for computer supported cooperative work to span more than a single local area network. In particular, support for collaboration is even more important when two or more geographically separate design groups work on related parts of a design. In anticipation of new high speed network technologies, we are developing new protocols to expand OREL to support such multiple geographically remote local area networks, and understand what inherent artifacts will change the quality of collaboration as compared with collocation. We believe that extensions to TCP/IP for multicasting and broadcasting would provide the necessary functionality to build support for geographically remote operation. Such extensions would be used as a foundation for the implementation of broadcast methods.

Concurrency control needs of computer supported cooperative work are very different from conventional requirements. The traditional database model of transactions is inadequate. In conventional systems the intent is to give each individual user the illusion of being the only person operating the system. In collaborative systems the aim is the opposite. We intend to make each user of the system as aware as possible of other users actions.

In the operation of coSARA there is a natural amount of parallelism due to the activities of the collaborating designers. An object model similar to actors (Hewit & Lieberman [1981]) could have an impact on the ability to model and program a system such as coSARA. Further research is necessary to evaluate this conjecture.

In section 4.2 we demonstrated the use of OREL for creation of a data model of design data used by SARA tools in support of the SARA design methodology. In the process of creating this model and using it as the base for programming coSARA, several issues about the use of OREL emerged. The essential problem is how to set the programmer's mind to think of relations. It is necessary to research which are adequate abstractions for using a system such as OREL. For example, before we started to seriously focus on the modeling of SARA objects there were no pair objects in OREL, only general relations. During the process of modeling the SARA objects it became clear that many relations defined in the model were simply relating pairs of objects. This resulted in the inclusion of directed and undirected pairs as useful abstractions. We made use of this abstractions to introduce some optimization in their implementation, relative to the implementation of general relations.

OREL influences the performance of a system built with it in two ways: broadcast methods impose a significant overhead, and the generality of OREL protocols also carries an overhead. There are not many ways to improve the overhead of broadcast methods other than

careful programming of their implementation and judicious use of them. The overhead of the OREL protocol is more amenable to improvement. Since the protocol is composed of messages, it is easy to redefine methods for specific OREL classes. For example, the messages relate, unrelate, and find-objects could be redefined for certain pair classes to use hash tables to store the relation pairs.

6.2.2. Problems related to tool integration and user interfaces

There is a mismatch between SARA modeling power and the need to model dynamically changing structures. The case when the structure of the system changes dynamically arises frequently in user interfaces. For example take the case of linking a new canvas to an existing zoom tool. Since the environment of the zoom must contain a model of all the contacts and other tools that are used by the zoom, it becomes necessary to alter the structure of the structure of the environment module of the zoom.

We have adopted a pragmatic solution that is not quite satisfactory. There is a privileged process that knows how to alter the structure of a running system. This privileged process is foreign to the SARA modeling methodology. This is explained in section 5.3.1. It is necessary to search for alternative modeling techniques that allow modeling of dynamically changing structures. We believe that a reasonable approach is to define mechanisms that will alter structures dynamically by application of proper refinement.

Although we have a procedure for partially integrating foreign tools, The method is manually intensive because it is necessary to program the data gateways that enable the foreign tools to share OREL with other tools and it is also necessary to build the callback gateways that enable a foreign tool to share the common user interface. We still seek ways for automatic generation of "gateways" to support integration of foreign tools.

There are many similarities between tools and contacts at certain levels of abstraction. It is still an open problem to determine good abstractions to design and build the kind of user interface that we have developed. We believe now that contacts should be assimilated tools. This would provide a smoother interface to the user interface and tool designer.

SARA modeling methods and tools provide powerful formalisms for specifying tool behavior and tool user interface. However, there is an intensive programming phase in creating the layout of individual contacts and of the user interface. A tool to aid the user interface to specify these layouts would be a great asset.

6.2.3. Modeling of multi-user design protocols

A problem in collaborative design is how to model the process by which a group of collaborators interact with each other during a design procedure. Winograd and Flores [1986] have done pioneering work in this area, proposing a state transition model of

conversations. They assume that a conversation represents interaction between persons with enough accuracy. A particular type of conversation that they have modeled represents conversations for actions and a commercial software system, The Coordinator, has been built based on this concept.

We feel a lack of a data model in the state transitions diagrams of Winograd and Flores. In section 5.3.3 we have shown how we can use SARA to model the behavior of tools in response to actions performed by multiple users simultaneously. A simple extrapolation of this example let us foresee how we could use SARA to model the behavior of a group of designers as multi-user design protocols. A collection of such models could then be stored in libraries and used for structuring tools to provide assistance in achieving progress in the group process of collaborative design. For example a tool could indicate which are the recommended steps to follow at a given point during the design of a system.

Knowledge extracted from the models of collaborative design protocols could also be used in more ways than providing assistance towards progress in the design process. Such knowledge could be used for structuring and tracing design history.

6.2.4. Management of design history

We consider management of design history to be a problem of great relevance in design methodology. We can point two main

subproblems of management of design history: how to record history using mechanisms that are not intrusive and how to trace history information.

Mechanisms for recording history in a non-intrusive way must be conceived if we are to be successful in collecting history information. The overhead of manual history collection can be so big that it may discourage users from collecting significant history information. This is a fact that our experience in software development using version control systems such as RCS (Tichy [1982]) reveals painfully.

Much can be automated by incorporating the role of history collection in the models of collaboration protocols. In such models we can introduce a *historian* process which would record design operations and request specific input from the user at prescribed points during the design process, in which it is known that design decisions will be made. For example, when partitioning a module into submodules it seems appropriate to record this operation and to request the team of designers to document the reasons that lead to a specific partition.

It is obvious that a great amount of information can be collected[1]. The problem then is how to use it to effectively assist a team of designers in achieving progress successfully. It is necessary to

[1] Conklin and ... [1990] report having manually collected 8,000 items of information in a single design project.

conceive ways to structure the information in useful ways, so that it can be quickly perused at different levels of abstraction. A straightforward representation for design history is a hypertext system, in which the nodes contain history items.

The problem of browsing the design history is solved by finding ways in which to structure the information. Furthermore we foresee automatization of this process based on information extracted from the different components of a design, including requirements, models of the system being designed, evaluations of the design, and models of design protocols used to structure the interaction among designers. One possible way to automatize this process is to take an approach similar that used by Waters in the Programmer's Apprentice project (Rich & Waters [1988]), which is described in Section 2.3.4. We could use cliches of designers actions and decisions to derive structural relations between history items automatically. For example, partitioning a module and making a design decision are design history cliches.

6.3. Summary

Our research has produced a proof of concept for feasibility of computer-based environments in support of collaborative design. The main focus of this work has been on the creation of enabling technologies that will allow to build such system, while having a concern for the group processes that exist in the domain of design. A central design goal that we have achieved is to build a system

general enough to allow modeling and implementation of different group processes for design, thus providing a platform for experimentation.

Contributions have been made to that area of object-oriented systems, by the creation of OREL, a system for interactive sharing of data, that includes objects and relations as modeling primitives. OREL provides a graphic language for modeling data, a programming interface to data, interactive sharing of data in a distributed set of workstations and persistent storage. The features of the graphical language for modeling data allow management of the high complexity found in design data. Interactive sharing of data is used as the basis for supporting communication and coordination in a collaborative environment.

Contributions have been made to the area of integrated systems and user interfaces by the creation of a methodology for modeling and integration of tools that includes a tool modeling method, a procedure for tool integration and a multi-user interface development system. The tool modeling method and the user interface systems are based on OREL and SARA for modeling of data and behavior respectively. The use of SARA for modeling tool behavior allows the specification of concurrent actions in the tools and response to concurrent actions by multiple users, enabling thus the modeling group processes for design of computer systems.

Appendix I: SARA primitives

Structural Model

The primitives to create hierarchical structural models in SARA are: *modules*, *sockets* and *interconnections*. These primitives are summarized in Figure I.1. Named parent modules contain fully nested children modules. Sockets are associated with a module and are known both inside and outside the module. A sockets is a named place for delivery of either a service provided by the module that contains the socket or a service required from some other module. Interconnections provide binding between sockets. Interconnections are not directed, representing only a communication line between two sockets. Nothing is said about neither the direction in which communication flows through the interconnection (i.e., which module provides a service and which module uses the provided service) nor the type of information that is carried by the interconnection.

Every SARA structural model has a top level module called UNIVERSE which has no sockets. This module is always partitioned in two submodules: ENVIRONMENT and SYSTEM. Module ENVIRONMENT is a model of the environment in which SYSTEM is supposed to exhibit its intended behavior. Moduel SYSTEM is a model of the system under design.

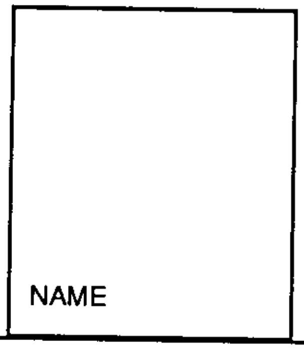
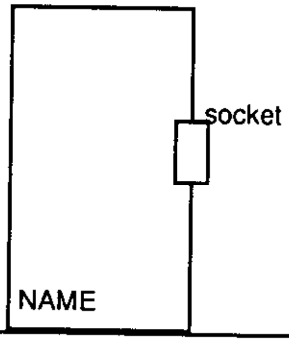
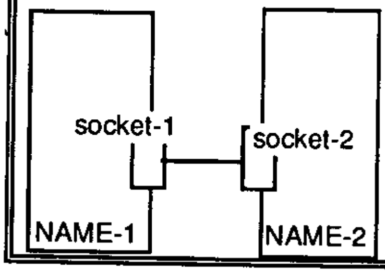
	**MODULE:** [rectangle] It may contain a set of submodules or it may contain a GMB. A module is also composed of sockets and interconnections.
	**SOCKET:** [small rectangle on the boundary of a module]: Represents a named place for delivery of services by modules.
	**INTERCONNECTION:** [line] connects pairs of sockets

Figure I.1: Summary of structure modeling primitives

Sockets provide an information hiding mechanism allowing control of the access to services provided by the module. It has been shown that for software design it is necessary to augment the concept of sockets with a Module Interface Description (MID) model (Berry, Estrin & Penedo [1981]; Penedo [1981]) which establishes the accessibility of resource names (procedures, data types, etc) and helps bind resource names to providers and users of those resources. Berry [1984] showed how Ada satisfies the requirements for the MID model tool.

Behavioral Model

The behavioral model is composed of models in three related domains: control, data and interpretation. It is therefore possible to approach these three aspects of a system independently, building related models for each domain. Figure I.2 and I.3 summarize the primitives used for modeling the behavior of a system.

The control domain modeling primitives are control nodes, which represent processing activity, and control arcs, which define sequencing of node initiations. Control arcs are directed arcs that connect many nodes to many nodes. All nodes which are the source of a directed control arc comprise the control arc's tail set and the control arc is said to be an output arc of the control node; all nodes which are destinations for a directed control arc comprise the head set of the control arc and the control arc is said to be an input arc of the control

node. A set of tokens on named arcs of the control graph represents the state of the graph.

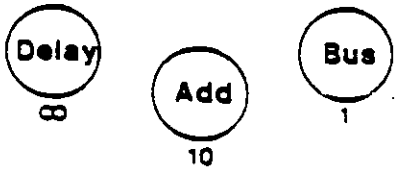
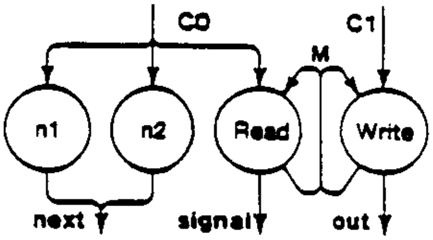
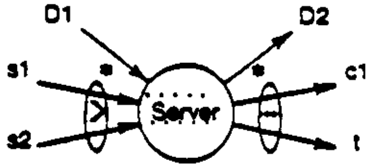
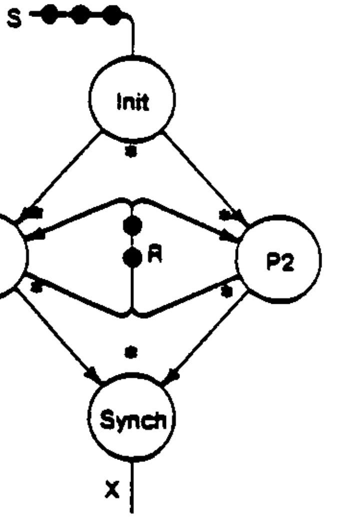
$N = (\{a\}, ServerType)$ $ServerType: \{a\} \rightarrow \{1,2,3,\ldots\} \cup \{\infty\}$ Each *control node* represents an event in the computation being modeled. The event is characterised by finite or infinite capacity to respond to simultaneous new inputs.	
$A_c = (\{a\}, SourceSet, DestinationSet, QueueDiscipline)$ $SourceSet, DestinationSet: \{a\} \rightarrow 2^{(N \cup M)}$, $QueueDiscipline: \{a\} \rightarrow \{FCFS, LCFS\}$, *Control arcs* represent static precedence relations between sets of events. A control arc carries control from an event at its source to a nondeterministically selected event at its destination. The QueueDiscipline function specifies the order in which control inputs are processed by finite-capacity destination events.	 C0 C1 M n1 n2 Read Write next signal out
Event Pre- and Post-Conditions $L = (L-, L+)$ $L-: a \in N \rightarrow >\vert 2^{(m-)} - \phi \vert m-: \{a \vert a \in DS(a)\} \rightarrow \{0,1,2,\ldots\} \cup \{*\} \vert$, $L+: (L- \times a \in N) \rightarrow \vert 2^{(m+)} - \phi \vert m+: \{a \vert a \in SS(a)\} \rightarrow \{0,1,2,\ldots\} \vert)$ The *input logic* for an event $\vert L-(a) \vert$, specifies a partially ordered set of pre-conditions, or input-arc markings, which can initiate the event. The *output logic* for an initiating condition, $\vert L+(L-(a),a) \vert$, specifies alternative output arc markings which will occur when the event terminates. ">" is a partial ordering on the input markings which defines static and dynamic priorities among possible event initiation conditions.	 $m_1 = \{s1(1), D1(1)\}$ $m_2 = \{s2(1), D1(1)\}$ $L-(Server) = \{>: m_1, m_2\}$ $L+(m_1) = \{c1(1), D2(1)\}$ $L+(m_2) = \{t(1), D2(1)\}$
Initial Control State $S_0: A_c \rightarrow \{0,1,2,\ldots\}$ The *initial control state vector* species the distribution of *tokens* on control arcs at the start of the period over which system behavior will be studied.	

Figure I.2: Behavioral model primitives

(adapted from Mary K. Vernon's Ph.D. dissertation [1984])

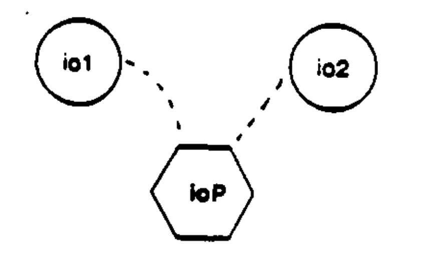
<table>
<tr>
<td>

Processors, Mapping of Processors to Control Nodes

$P = \{p\}$
$M = (M-:P \rightarrow 2^N-\{\phi\}, M+:N \rightarrow P \bigcup \{\phi\})$

A *data processor* represents a data transformer which is mapped to one or more events in the control domain. The processor is activated whenever any one of its mapped control nodes is initiated.

</td>
<td>



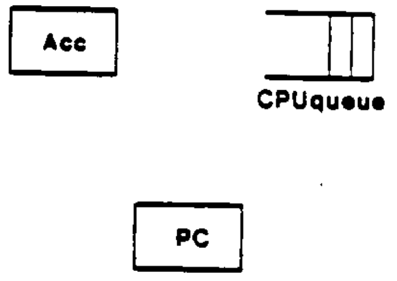
</td>
</tr>
<tr>
<td>

Datasets

Q = ({q},QueueType)
QueueType: {q}→{simple, FCFS, LCFS}

A *dataset* retains data, of a type specified in the interpretation domain, which is communicated among processors. Datasets may be simple, which hold one data value of the specified type at a time, or queues which hold multiple data values.

</td>
<td>



</td>
</tr>
<tr>
<td>

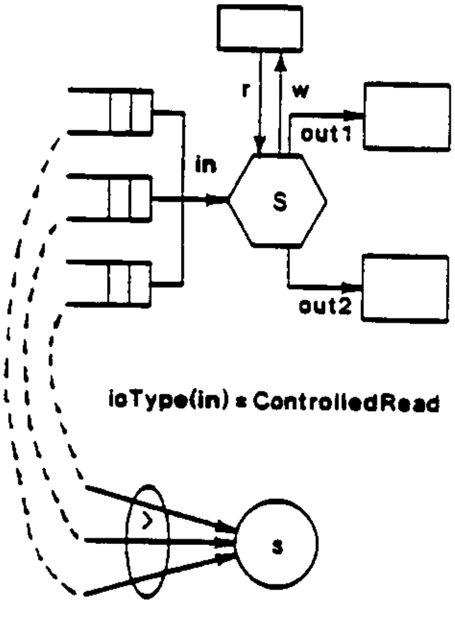
Data arcs

$A_d = (\{d\},ioType,ProcessorSet,DataSet,ioControl)$

ioType: {d}→{Read, ControlledRead, PriorityRead, FCFSread, LCFSread} ∪ {Write, ControlledWrite, WDataAndControl, WControlIfValueChanges}
ProcessorSet: $\{d\} \rightarrow 2^{P \bigcup N}-\{\phi\}$
DataSet: $\{d\} \rightarrow >>[2^{Q \bigcup N}-\{\phi\}]$
ioControl: $(DataSet \times \{d\}) \rightarrow >>[2^{A_c}]$

Data arcs specify data access paths for processors. Processors can have read or write access to datasets, and data access can be related to control flow in the control graph.

</td>
<td>



</td>
</tr>
</table>

Figure I.3: Behavioral model primitives

(adapted from Mary K. Vernon's Ph.D. dissertation [1984])

Control nodes have an input logic expression and an output logic expression. The terms of these logic expressions are the input control arcs and the output control arcs respectively. When a control arc has a token, it represents a value true in the logic expression and it represents a value false if it has no tokens. The input control logic of a node defines the conditions under which the control node is initiated and defines how tokens are absorbed from the input control arcs. The output control logic of a node defines how control is distributed upon termination of a control node activation, subject to branch selection in the interpretation domain.

Tokens are placed on the control arcs of a model to provide the initialized control state of the model. Any control node is initiated when there are sufficient tokens on its input arcs to satisfy its input logic. The dynamic behavior of the graph is determined by the the flow of tokens. The control flow domain incorporates non-determinism, permits mutual exclusion constructs and allows management of finite resources.

The data domain modeling primitives are: data processors, data arcs, datasets and dataset queues. Data processors are mapped one-to-many to control nodes in an associated control graph. Datasets represent data values and data arcs represent read/write access paths from data processors. Data processors may embody control flow decisions,

processing delays to model timing and data transformations as specified in the interpretation domain.

The interpretation domain corresponds to a programming language used to specify data transformations, with suitable extensions to specify access to datasets, branching control flow decisions, and processing delays.

The Buffer Example

We have chosen a simple system to illustrate how SARA is used to model a system. The example is an bounded input/output buffer with a capacity to hold up to **N** messages which are all of the same size. The buffer is initialized to be empty. When the system is in operation the buffer can be read and written concurrently. When the buffer is full and a write operation is attempted, the writer will be suspended until there is space in the buffer to hold the message. When the buffer is empty and a read operation is attempted the readre will be suspended until there is a message in the buffer. All messages are delivered by the buffer in the same order in which they are received by the buffer. No message is destroyed and no message is read twice.

The environment contains two processes: sender and receiver. The sender process behaves as follows:

1. It writes messages into the buffer.

2. It sends one message at a time.

3. It writes a finite number number of messages to the buffer.

4. It terminates after writing the last message.

The receiver process behaves as follows:

1. It reads messages from the buffer.

2. It reads one message at a time.

3. After the last message is read it terminates

Figure I.4 shows a structural model of the BUFFER design. We partition the BUFFER design into two submodules: ENVIRONMENT and SYSTEM. These submodules communicate through the interconnections READ and WRITE. Both modules ENVIRONMENT and SYSTEM could be partitioned further if needed. However, they are simple enough to synthesize behavioral models for them and no further partitions are needed.

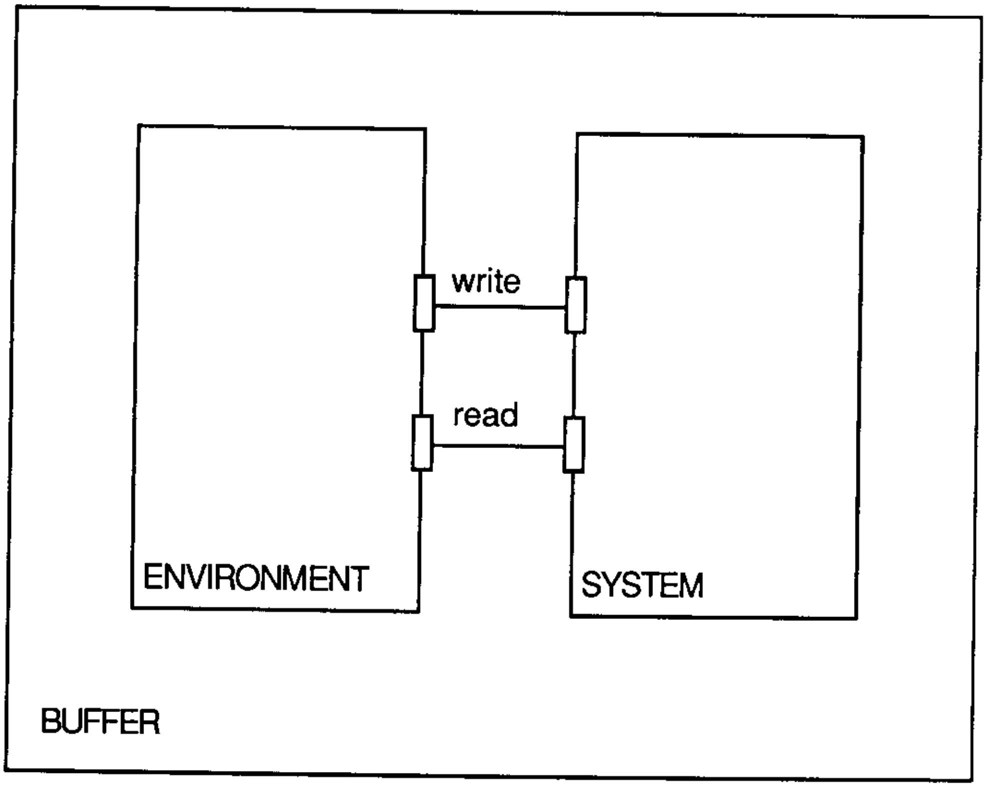


Figure I.4: Structural model of BUFFER

Figure I.5 contains the control domain part of a behavioral model of the BUFFER design. The functions of the various components in the control graph are:

INIT initiation process, initiates sender and receiver.

SEND sender process, sends messages to the BUFFER and waits for acknowledgment from the BUFFER.

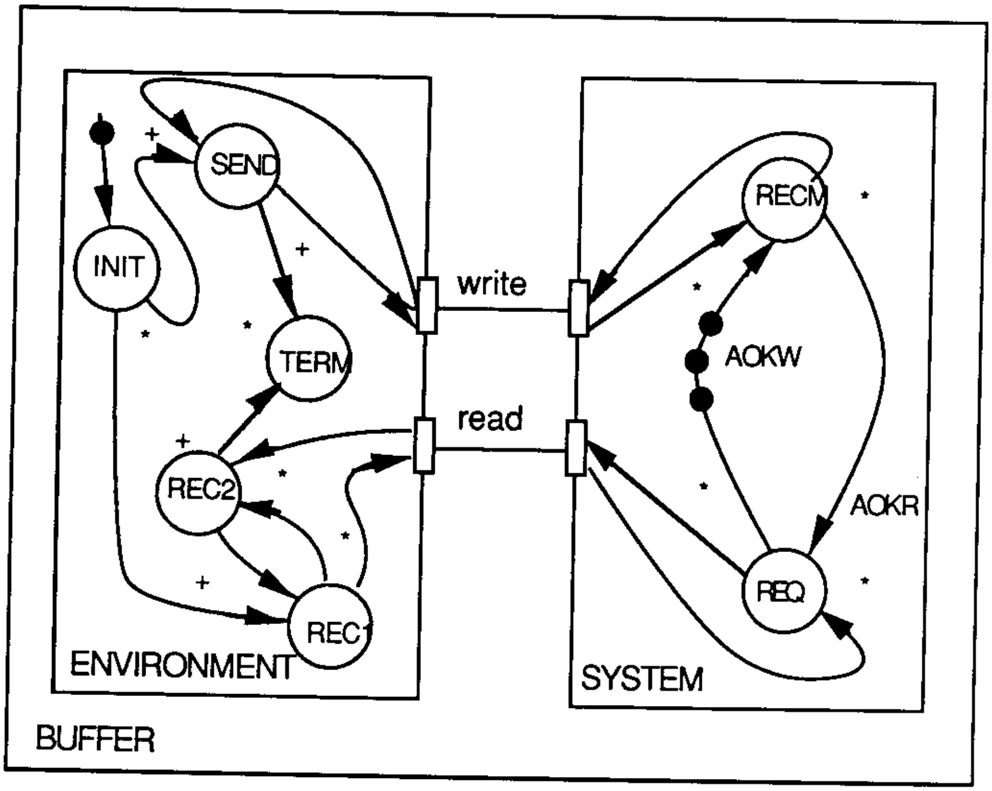


Figure I.5: Control flow model of BUFFER[3]

REC1 part of the receiver process that issues a read request.

REC2 part of the receiver process that sccepts a message from the BUFFER and enables REC1 to issue more read requests.

TERM termination process.

[3] In this Figure we have assumed that the BUFFER can hold up to 3 messages, what is modeled by initializing the control arc AOKW with 3 tokens.

RECM receives a message from ENVIRONMENT, acknowledges and stores the message.

REQ receives a read request, retrieves a stored message and sends it to ENVIRONMENT.

AOKW the number of tokens in this control arc represents the number of empty messages slots in the buffer.

AOKR the number of tokens in this control arc represents the number of messages stored in the buffer.

Figure I.6 contains the data domain part of the behavioral model of the BUFFER design. The functions of the various components of the data graph are:

CHK mapped to control node TERM; it checks the message received against the initial messages to determine that they are the same and in the same order before termination.

RDI mapped to control node SEND; it reads messages from INPUT and writes them into MESIN.

RDO mapped to control node REC2; it reads messages from MESOUT and writes them into OUTPUT.

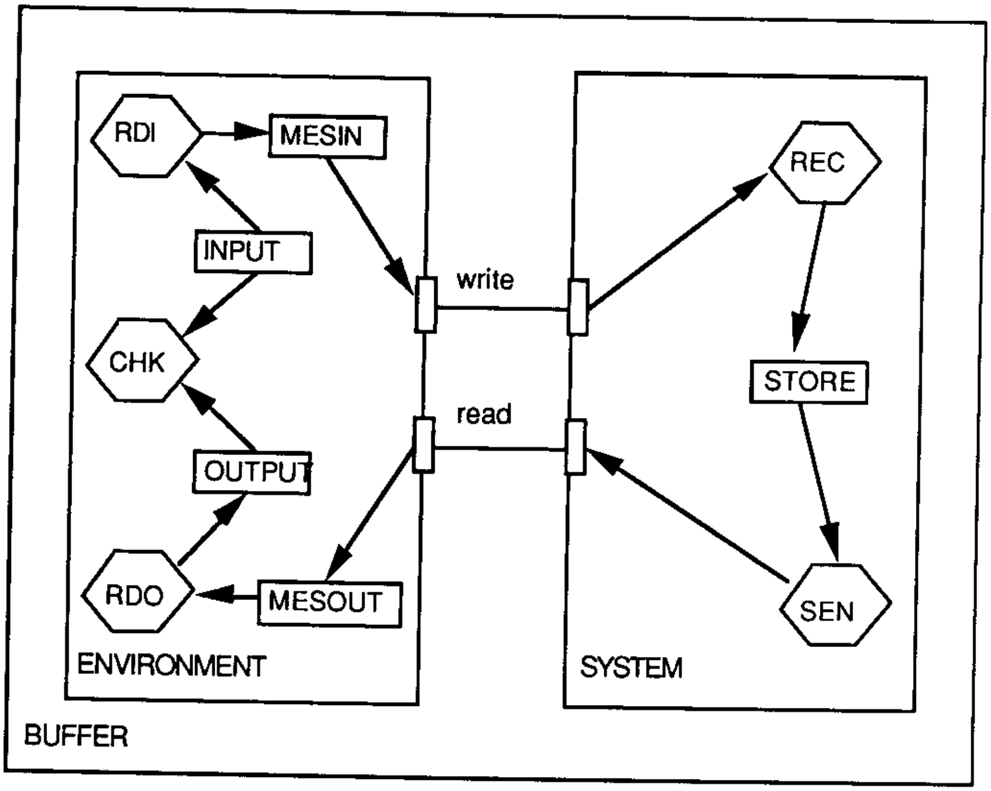


Figure I.6: Data flow model of BUFFER

REC mapped to control node RECM; it receives messages from ENVIRONMENT and stores them in STORE.

SEN mapped to control node REQ; it reads messages from STORE and and passes them to ENVIRONMENT.

INPUT initial sequence of messages.

OUTPUT messages read from BUFFER, to be checked against INPUT upon termination.

MESIN and **MESOUT**

data interface with BUFFER.

STORE storage for messages.

Appendix II: OREL object protocols

A set of basic protocols is supported by all OREL objects including functionality to make and initialize objects, find the class of an object, navigate composite objects, relate objects and query relations.

We have designed protocols for simple classes, composite classes and relation classes. These protocols include functionality for class objects as well as for their instances.

In this section we describe each one of the OREL protocols. To specify a message we use the following format:

message name

$\mathbf{ARGUMENT_1}$ [$\mathbf{type_1}$]

...

$\mathbf{ARGUMENT_n}$ [$\mathbf{type_n}$]

Description of the functionality associated with the message. References to the i-th argument is $\mathrm{ARGUMENT_i}$.

II.1. SIMPLE-OBJECT Protocol

initialize-instance

OBJECT simple-object

&rest INITARGS

Initializes OBJECT according to initialization arguments INITARGS. INITARGS is a list of pairs of the form keyword value, where keyword is the name of some slot defined either for the class of OBJECT or one of its superclasses. The programmer is not supposed to write a primary initialize instance method, but to write combined methods to be called either before, after or around the primary method of this message.

print-object

OBJECT simple-object

STREAM stream

Writes a printed representation of OBJECT to the stream STREAM (streams are defined in (Steele [1990]).

class-of

OBJECT simple-object

Returns the class of OBJECT.

class-name

CLASS

Takes a class object CLASS and returns a symbol that names the class CLASS.

make-<name>

&key STORABLE-ID STORABLE-NAME
&rest INITARGS

This function creates and returns a new instance of a simple class named NAME. If STORABLE-NAME is specified, the new object is given that name otherwise it is initialized as an unnamed object. If STORABLE-ID is specified, the object is assigned that id, otherwise a new unique system-wide id is generated and assigned to the newly created object. The object is otherwise initialized according to INITARGS. INITARGS has the same meaning that it has for the initialize-instance message.

II.2. RELATION-OBJECT protocol

relate

RELATION relation-object

LIST-OF-OBJECTS

Adds a tuple composed by the objects in the list LIST-OF-OBJECTS to the relation RELATION. Each of the objects is qualified either by its class or by its role. For example, suppose that relation RELATION relates classes A and B. Then we would call (relate R :A obj :B obj).

unrelate

RELATION relation-object

FUNCTION

Removes all the tuples of the relation RELATION that satisfy the conditon: (apply FUNCTION tuple) returns non-nil.

unrelate

RELATION pair-object

OBJECTS

Removes the tuple that contains the objects OBJECTS. The argument OBJECTS must be a list of length 2.

find-objects

RELATION relation-object

FUNCTION

&rest ARGS

Returns a set of tuples **T** ∈ **R** such that for each **T** in the list, the evaluation of:, (apply FUNCTION T args) is not **nil.**

find-objects

PAIR pair-object

OBJECT

Returns the objects or objects related to the argument OBJECT through

map-relation

RELATION relation-object

FUNCTION

&rest ARGS

For each tuple T stored in the relation RELATION, the form (apply FUNCTION T ARGS) is evaluated. The result of these evaluations are stored and returned as a list. This behavior is similar to the function mapcar of CommonLisp.

II.3. COMPOSITE-OBJECT protocol

components-of

OBJECT composite-object

Returns a list of all components of object OBJECT

comp-<CLASS>

OBJECT composite-object

Returns the components of class CLASS of the composite object OBJECT. If there is no class named CLASS defined as a component of OBJECT this method signals an error.

comp-parent-of

OBJECT simple-object

Returns the parent composite of OBJECT.

comp-ancestors-of

OBJECT simple-object

Returns the path from the root of the composite, that contains OBJECT as one of its parts, to the object OBJECT. The path is returned as a list of objects whose first element is the root composite object.

compositep

OBJECT simple-object

Returns true if OBJECT is a composite.

component-of-p

COMPOSITE composite-object

OBJECT object

Returns true if OBJECTis a component of COMPOSITE.

descendant-p

COMPOSITE composite-object

OBJECT object

Returns true if OBJECT is either a direct or indirect component of COMPOSITE.

add-component

COMPOSITE composite-object

OBJECT simple-object

Add OBJECT to COMPOSITE..

del-component

COMPOSITE composite-object

OBJECT simple-object

Delete OBJECT from COMPOSITE.

comp-assoc

COMPOSITE composite-object

FUNCTION

&rest ARGS

Returns a list of components of COMPOSITE such that evaluation of the form: (apply FUNCTION component args) returns a not nil value.

Appendix III: Code generated for SARA objects by OREL translator

```
;;; -*- Mode: Lisp; Package: SARA -*-


(in-package 'OREL :use '(pcl object-world lisp))
```

The SARA-OBJECT class encapsulates properties that are common to all SARA such as annotations, graphic representation etc. The slot parent-composite of an instance of this class points to the composite that contains the instance.

```
(pcl:defclass SARA-OBJECT (SIMPLE-OBJECT STORABLE)
  ((annotation :initform nil :accessor SARA-OBJECT-annotation)
   (open       :initform t   :accessor SARA-OBJECT-open)
   (graphic :initform nil :accessor SARA-OBJECT-graphic)
   (name :initarg :init-name :initform nil :accessor SARA-OBJECT-
name)
   (parent-composite :initform nil :accessor comp-parent)))
```

make-SARA-OBJECT creates an instance of the class SARA-OBJECT

```
(defun make-SARA-OBJECT (&optional (name "") init-name )
  (make-instance 'SARA-OBJECT :storable-name name :storable-id
(object-world::unique-sym) :init-name init-name))


(pcl:defclass MODULE (COMPOSITE-OBJECT SARA-OBJECT)
```

```
   ((type :initform nil :accessor MODULE-type)
    (GMB :initform nil :accessor comp-GMB)
    (INTER :initform nil :accessor comp-INTER)
    (SOCKET :initform nil :accessor comp-SOCKET)
    (MODULE :initform nil :accessor comp-MODULE)
    (parent-composite :initform nil :accessor comp-parent)))

(defun make-MODULE (&optional (name "") )
  (make-instance 'MODULE :storable-name name :storable-id
                 (object-world::unique-sym)))

(pcl:defclass DESIGN (MODULE)
  ((parent-composite :initform nil :accessor comp-parent)))

(defun make-DESIGN (&optional (name "") &key )
  (make-instance 'DESIGN :storable-name name :storable-id (object-
world::unique-sym)))

(pcl:defclass INTER (UNDIRECTED-PAIR-OBJECT SARA-OBJECT)
  ((orel::pair-list :initform nil :accessor pair-list)))

(defun make-INTER (&optional (name "") &key )
  (make-instance 'INTER :storable-name name :storable-id (object-
world::unique-sym)))
```

```
(pcl:defclass SOCKET (SARA-OBJECT)
  ((INTER :initform nil :accessor SOCKET-INTER)
   (INTER :initform nil :accessor SOCKET-INTER)
   (parent-composite :initform nil :accessor comp-parent)))

(defun make-SOCKET (&optional (name "") )
  (make-instance 'SOCKET :storable-name name :storable-id (object-
world::unique-sym)))

(pcl:defclass OUTPUT-ARCS (DIRECTED-PAIR-OBJECT SARA-OBJECT)
  ((pair-list :initform nil :accessor pair-list)))

(defun make-OUTPUT-ARCS (&optional (name "") &key )
  (make-instance 'OUTPUT-ARCS :storable-name name :storable-id
(object-world::unique-sym)))

(pcl:defclass INPUT-ARCS (DIRECTED-PAIR-OBJECT SARA-OBJECT)
  ((pair-list :initform nil :accessor pair-list)))

(defun make-INPUT-ARCS (&optional (name "") &key )
  (make-instance 'INPUT-ARCS :storable-name name :storable-id
(object-world::unique-sym)))

(pcl:defclass TAIL-SET (DIRECTED-PAIR-OBJECT SARA-OBJECT)
  ((pair-list :initform nil :accessor pair-list)))
```

```
(defun make-TAIL-SET (&optional (name "") &key )
  (make-instance 'TAIL-SET :storable-name name :storable-id
(object-world::unique-sym)))

(pcl:defclass HEAD-SET (DIRECTED-PAIR-OBJECT SARA-OBJECT)
  ((pair-list :initform nil :accessor pair-list)))

(defun make-HEAD-SET (&optional (name "") &key )
  (make-instance 'HEAD-SET :storable-name name :storable-id
(object-world::unique-sym)))

(pcl:defclass GMB-MAPPING (DIRECTED-PAIR-OBJECT SARA-OBJECT)
  ((pair-list :initform nil :accessor pair-list)))

(defun make-GMB-MAPPING (&optional (name "") &key )
  (make-instance 'GMB-MAPPING :storable-name name :storable-id
(object-world::unique-sym)))

(pcl:defclass DATA-ARC (SARA-OBJECT)
  ((dyn-dataset :initform nil :accessor DATA-ARC-dyn-dataset)
   (type :initform nil :accessor DATA-ARC-type)
   (parent-composite :initform nil :accessor comp-parent)))

(defun make-DATA-ARC (&optional (name "") )
```

```
  (make-instance 'DATA-ARC :storable-name name :storable-id
(object-world::unique-sym)))


(pcl:defclass GENERIC-DATASET (SARA-OBJECT)
  ((parent-composite :initform nil :accessor comp-parent)))


(defun make-GENERIC-DATASET (&optional (name "") &key )
  (make-instance 'GENERIC-DATASET :storable-name name :storable-id
(object-world::unique-sym)))


(pcl:defclass DATA-SET-QUEUE (GENERIC-DATASET)
  ((value :initform nil :accessor DATA-SET-QUEUE-value)
   (q-type :initform nil :accessor DATA-SET-QUEUE-q-type)
   (init-value :initform nil :accessor DATA-SET-QUEUE-init-value)
   (parent-composite :initform nil :accessor comp-parent)))


(defun make-DATA-SET-QUEUE (&optional (name "") )
  (make-instance 'DATA-SET-QUEUE :storable-name name :storable-id
(object-world::unique-sym)))


(pcl:defclass DATA-SET (GENERIC-DATASET)
  ((init-value :initform nil :accessor DATA-SET-init-value)
   (parent-composite :initform nil :accessor comp-parent)))


(defun make-DATA-SET (&optional (name "") )
```

```
   (make-instance 'DATA-SET :storable-name name :storable-id
(object-world::unique-sym)))


(pcl:defclass DATA-PROCESSOR (SARA-OBJECT)
   ((process :initform nil :accessor DATA-PROCESSOR-process)
    (pid :initform nil :accessor DATA-PROCESSOR-pid)
    (parent-composite :initform nil :accessor comp-parent)))


(defun make-DATA-PROCESSOR (&optional (name "") )
   (make-instance 'DATA-PROCESSOR :storable-name name :storable-id
(object-world::unique-sym)))


(pcl:defclass CONTROL-ARC (SARA-OBJECT)
   ((queue :initform nil :accessor CONTROL-ARC-queue)
    (break-points :initform nil :accessor CONTROL-ARC-break-points)
    (local-tokens :initform nil :accessor CONTROL-ARC-local-tokens)
    (tokens :initform nil :accessor CONTROL-ARC-tokens)
    (aliases :initform nil :accessor CONTROL-ARC-aliases)
    (init-tokens :initform nil :accessor CONTROL-ARC-init-tokens)
    (TAIL-SET :initform nil :accessor CONTROL-ARC-TAIL-SET)
    (HEAD-SET :initform nil :accessor CONTROL-ARC-HEAD-SET)
    (parent-composite :initform nil :accessor comp-parent)))


(defun make-CONTROL-ARC (&optional (name "") )
   (make-instance 'CONTROL-ARC :storable-name name :storable-id
```

```
(object-world::unique-sym)))


(pcl:defclass CONTROL-NODE (SARA-OBJECT)
  ((break-points :initform nil :accessor CONTROL-NODE-break-points)
   (queue :initform nil :accessor CONTROL-NODE-queue)
   (actp :initform nil :accessor CONTROL-NODE-actp)
   (visits :initform nil :accessor CONTROL-NODE-visits)
   (customers :initform nil :accessor CONTROL-NODE-customers)
   (capacity :initform nil :accessor CONTROL-NODE-capacity)
   (out-logic :initform nil :accessor CONTROL-NODE-out-logic)
   (in-logic :initform nil :accessor CONTROL-NODE-in-logic)
   (OUTPUT-ARCS :initform nil :accessor CONTROL-NODE-OUTPUT-ARCS)
   (INPUT-ARCS :initform nil :accessor CONTROL-NODE-INPUT-ARCS)
   (GMB-MAPPING :initform nil :accessor CONTROL-NODE-GMB-MAPPING)
   (parent-composite :initform nil :accessor comp-parent)))


(defun make-CONTROL-NODE (&optional (name "") )
  (make-instance 'CONTROL-NODE :storable-name name :storable-id
(object-world::unique-sym)))
```

DATA-REL is a general relation class. the slot tuple-list points to a list of all the tuples that are stored in the relation object. The other slots store properties of the relation.

```
(pcl:defclass DATA-REL (RELATION-OBJECT)
  ((DATA-ARC-many-p :reader DATA-ARC-many-p :initform t)
```

```
    (DATA-ARC-ordered-p :reader DATA-ARC-ordered-p :initform t)
    (DATA-ARC-directed-p :reader DATA-ARC-directed-p :initform t)
    (DATA-ARC-map-type :reader DATA-ARC-map-type :initform t)
    (DATA-PROCESSOR-many-p :reader DATA-PROCESSOR-many-p :initform
t)
    (DATA-PROCESSOR-ordered-p :reader DATA-PROCESSOR-ordered-p
:initform t)
    (DATA-PROCESSOR-directed-p :reader DATA-PROCESSOR-directed-p
:initform t)
    (DATA-PROCESSOR-map-type :reader DATA-PROCESSOR-map-type
:initform t)
    (GENERIC-DATASET-many-p :reader GENERIC-DATASET-many-p :initform
t)
    (GENERIC-DATASET-ordered-p :reader GENERIC-DATASET-ordered-p
:initform t)
    (GENERIC-DATASET-directed-p :reader GENERIC-DATASET-directed-p
:initform t)
    (GENERIC-DATASET-map-type :reader GENERIC-DATASET-map-type
:initform t)
    (parent-composite :initform nil :accessor comp-parent)
    (orel::tuple-list :initform nil :accessor orel::tuple-list)))

(defun make-DATA-REL (&optional (name "") &key )
  (make-instance 'DATA-REL :storable-name name :storable-id
(object-world::unique-sym)))
```

The class DATA-REL-tuple represents one tuple of the relation DATA-REL Each instance of the tuple class has one slot for each one of the classes that participates in the relation.

```
(pcl:defclass DATA-REL-tuple (storable)
  (
   (DATA-ARC :accessor DATA-ARC :initarg :DATA-ARC)
   (DATA-PROCESSOR :accessor DATA-PROCESSOR :initarg :DATA-
PROCESSOR)
   (GENERIC-DATASET :accessor GENERIC-DATASET :initarg :GENERIC-
DATASET)))


(pcl:defclass GMB-SOCKET (RELATION-OBJECT)
  ((SOCKET-many-p :reader SOCKET-many-p :initform t)
   (SOCKET-ordered-p :reader SOCKET-ordered-p :initform t)
   (SOCKET-directed-p :reader SOCKET-directed-p :initform t)
   (SOCKET-map-type :reader SOCKET-map-type :initform t)
   (CONTROL-ARC-many-p :reader CONTROL-ARC-many-p :initform t)
   (CONTROL-ARC-ordered-p :reader CONTROL-ARC-ordered-p :initform
t)
   (CONTROL-ARC-directed-p :reader CONTROL-ARC-directed-p :initform
t)
   (CONTROL-ARC-map-type :reader CONTROL-ARC-map-type :initform t)
   (DATA-ARC-many-p :reader DATA-ARC-many-p :initform t)
   (DATA-ARC-ordered-p :reader DATA-ARC-ordered-p :initform t)
```

```
    (DATA-ARC-directed-p :reader DATA-ARC-directed-p :initform t)
    (DATA-ARC-map-type :reader DATA-ARC-map-type :initform t)
    (parent-composite :initform nil :accessor comp-parent)
    (orel::tuple-list :initform nil :accessor orel::tuple-list)))


(defun make-GMB-SOCKET (&optional (name "") &key )
  (make-instance 'GMB-SOCKET :storable-name name :storable-id
(object-world::unique-sym)))

(pcl:defclass GMB-SOCKET-tuple (storable)
  (
   (SOCKET :accessor SOCKET :initarg :SOCKET)
   (CONTROL-ARC :accessor CONTROL-ARC :initarg :CONTROL-ARC)
   (DATA-ARC :accessor DATA-ARC :initarg :DATA-ARC)))

(pcl:defclass DATA-GRAPH (COMPOSITE-OBJECT SARA-OBJECT)
  ((DATA-REL :initform nil :accessor comp-DATA-REL)
   (DATA-ARC :initform nil :accessor comp-DATA-ARC)
   (DATA-SET-QUEUE :initform nil :accessor comp-DATA-SET-QUEUE)
   (DATA-SET :initform nil :accessor comp-DATA-SET)
   (DATA-PROCESSOR :initform nil :accessor comp-DATA-PROCESSOR)
   (parent-composite :initform nil :accessor comp-parent)))

(defun make-DATA-GRAPH (&optional (name "") &key )
```

```
    (make-instance 'DATA-GRAPH :storable-name name :storable-id
  (object-world::unique-sym)))

  (pcl:defclass CONTROL-GRAPH (COMPOSITE-OBJECT SARA-OBJECT)
    ((OUTPUT-ARCS :initform nil :accessor comp-OUTPUT-ARCS)
     (INPUT-ARCS :initform nil :accessor comp-INPUT-ARCS)
     (TAIL-SET :initform nil :accessor comp-TAIL-SET)
     (HEAD-SET :initform nil :accessor comp-HEAD-SET)
     (CONTROL-ARC :initform nil :accessor comp-CONTROL-ARC)
     (CONTROL-NODE :initform nil :accessor comp-CONTROL-NODE)
     (parent-composite :initform nil :accessor comp-parent)))

(defun make-CONTROL-GRAPH (&optional (name "") &key )
   (make-instance 'CONTROL-GRAPH :storable-name name :storable-id
(object-world::unique-sym)))

(pcl:defclass GMB (COMPOSITE-OBJECT SARA-OBJECT)
   ((GMB-SOCKET :initform nil :accessor comp-GMB-SOCKET)
    (GMB-MAPPING :initform nil :accessor comp-GMB-MAPPING)
    (DATA-GRAPH :initform nil :accessor comp-DATA-GRAPH)
    (CONTROL-GRAPH :initform nil :accessor comp-CONTROL-GRAPH)
    (parent-composite :initform nil :accessor comp-parent)))

(defun make-GMB (&optional (name "") &key )
  (make-instance 'GMB :storable-name name :storable-id (object-
```

```
world::unique-sym)))
```

The methods of the form add-<class>, such as add-GMB add a component of the class <class> to the copmposite given as argument

```
(defmethod add-GMB ((c MODULE) (o GMB))
  (setf (comp-GMB c) (cons o (comp-GMB c)))
  (setf (comp-parent o) c))


(defmethod del-GMB ((c MODULE) (o GMB))
  (setf (comp-GMB c) (orel::class-remove-list (comp-GMB c) o)))
(defmethod del-module ((c MODULE) (o module))
  (setf (comp-module c)  (delete o (comp-module c))))


(defmethod add-INTER ((c MODULE) (o INTER))
  (setf (comp-INTER c) (cons o (comp-INTER c)))
  (setf (comp-parent o) c))
```

del-<class> methods delete a component of the argument composite.

```
(defmethod del-INTER ((c MODULE) (o INTER))
  (setf (comp-INTER c) (orel::class-remove-list (comp-INTER c) o)))


(defmethod add-SOCKET ((c MODULE) (o SOCKET))
  (setf (comp-SOCKET c) (cons o (comp-SOCKET c)))
  (setf (comp-parent o) c))
```

```
(defmethod del-SOCKET ((c MODULE) (o SOCKET))
  (setf (comp-SOCKET c) (delete o (comp-SOCKET c))))

(defmethod orel:components-of ((o MODULE))
  (append  (comp-module o)
           (comp-GMB o)
           (comp-INTER o)
           (comp-SOCKET o)))

(defmethod add-MODULE ((c MODULE) (o MODULE))
  (setf (comp-MODULE c) (cons o (comp-MODULE c)))
  (setf (comp-parent o) c))

(defmethod add-DATA-REL ((c DATA-GRAPH) (o DATA-REL))
  (setf (comp-DATA-REL c) (cons o (comp-DATA-REL c)))
  (setf (comp-parent o) c))

(defmethod del-DATA-REL ((c DATA-GRAPH) (o DATA-REL))
  (setf (comp-DATA-REL c) (orel::class-remove-list (comp-DATA-REL
c) o)))

(defmethod add-DATA-ARC ((c DATA-GRAPH) (o DATA-ARC))
  (setf (comp-DATA-ARC c) (cons o (comp-DATA-ARC c)))
  (setf (comp-parent o) c))
```

```
(defmethod del-DATA-ARC ((c DATA-GRAPH) (o DATA-ARC))
  (setf (comp-DATA-ARC c) (orel::class-remove-list (comp-DATA-ARC
c) o)))

(defmethod add-DATA-SET-QUEUE ((c DATA-GRAPH) (o DATA-SET-QUEUE))
  (setf (comp-DATA-SET-QUEUE c) (cons o (comp-DATA-SET-QUEUE c)))
  (setf (comp-parent o) c))

(defmethod del-DATA-SET-QUEUE ((c DATA-GRAPH) (o DATA-SET-QUEUE))
  (setf (comp-DATA-SET-QUEUE c) (orel::class-remove-list (comp-
DATA-SET-QUEUE c) o)))

(defmethod add-DATA-SET ((c DATA-GRAPH) (o DATA-SET))
  (setf (comp-DATA-SET c) (cons o (comp-DATA-SET c)))
  (setf (comp-parent o) c))

(defmethod del-DATA-SET ((c DATA-GRAPH) (o DATA-SET))
  (setf (comp-DATA-SET c) (orel::class-remove-list (comp-DATA-SET
c) o)))

(defmethod add-DATA-PROCESSOR ((c DATA-GRAPH) (o DATA-PROCESSOR))
  (setf (comp-DATA-PROCESSOR c) (cons o (comp-DATA-PROCESSOR c)))
  (setf (comp-parent o) c))

(defmethod del-DATA-PROCESSOR ((c DATA-GRAPH) (o DATA-PROCESSOR))
```

```
  (setf (comp-DATA-PROCESSOR c) (orel::class-remove-list (comp-
DATA-PROCESSOR c) o)))

(defmethod orel:components-of ((o DATA-GRAPH))
  (append
          (comp-DATA-REL o)
          (comp-DATA-ARC o)
          (comp-DATA-SET-QUEUE o)
          (comp-DATA-SET o)
          (comp-DATA-PROCESSOR o)))

(defmethod add-OUTPUT-ARCS ((c CONTROL-GRAPH) (o OUTPUT-ARCS))
  (setf (comp-OUTPUT-ARCS c) (cons o (comp-OUTPUT-ARCS c)))
  (setf (comp-parent o) c))

(defmethod del-OUTPUT-ARCS ((c CONTROL-GRAPH) (o OUTPUT-ARCS))
  (setf (comp-OUTPUT-ARCS c) (orel::class-remove-list (comp-OUTPUT-
ARCS c) o)))

(defmethod add-INPUT-ARCS ((c CONTROL-GRAPH) (o INPUT-ARCS))
  (setf (comp-INPUT-ARCS c) (cons o (comp-INPUT-ARCS c)))
  (setf (comp-parent o) c))

(defmethod del-INPUT-ARCS ((c CONTROL-GRAPH) (o INPUT-ARCS))
  (setf (comp-INPUT-ARCS c) (orel::class-remove-list (comp-INPUT-
```

```
ARCS c) o)))


(defmethod add-TAIL-SET ((c CONTROL-GRAPH) (o TAIL-SET))
  (setf (comp-TAIL-SET c) (cons o (comp-TAIL-SET c)))
  (setf (comp-parent o) c))


(defmethod del-TAIL-SET ((c CONTROL-GRAPH) (o TAIL-SET))
  (setf (comp-TAIL-SET c) (orel::class-remove-list (comp-TAIL-SET
c) o)))


(defmethod add-HEAD-SET ((c CONTROL-GRAPH) (o HEAD-SET))
  (setf (comp-HEAD-SET c) (cons o (comp-HEAD-SET c)))
  (setf (comp-parent o) c))


(defmethod del-HEAD-SET ((c CONTROL-GRAPH) (o HEAD-SET))
  (setf (comp-HEAD-SET c) (orel::class-remove-list (comp-HEAD-SET
c) o)))


(defmethod add-CONTROL-ARC ((c CONTROL-GRAPH) (o CONTROL-ARC))
  (setf (comp-CONTROL-ARC c) (cons o (comp-CONTROL-ARC c)))
  (setf (comp-parent o) c))


(defmethod del-CONTROL-ARC ((c CONTROL-GRAPH) (o CONTROL-ARC))
  (setf (comp-CONTROL-ARC c) (orel::class-remove-list (comp-
CONTROL-ARC c) o)))
```

```
 (defmethod add-CONTROL-NODE ((c CONTROL-GRAPH) (o CONTROL-NODE))
   (setf (comp-CONTROL-NODE c) (cons o (comp-CONTROL-NODE c)))
   (setf (comp-parent o) c))


 (defmethod del-CONTROL-NODE ((c CONTROL-GRAPH) (o CONTROL-NODE))
   (setf (comp-CONTROL-NODE c) (orel::class-remove-list (comp-
CONTROL-NODE c) o)))
```

components-of returns a list with all the components of the composite given as argument

```
(defmethod orel:components-of ((o CONTROL-GRAPH))
  (append
            (comp-OUTPUT-ARCS o)
            (comp-INPUT-ARCS o)
            (comp-TAIL-SET o)
            (comp-HEAD-SET o)
            (comp-CONTROL-ARC o)
            (comp-CONTROL-NODE o)))


(defmethod add-GMB-SOCKET ((c GMB) (o GMB-SOCKET))
  (setf (comp-GMB-SOCKET c) (cons o (comp-GMB-SOCKET c)))
  (setf (comp-parent o) c))


(defmethod del-GMB-SOCKET ((c GMB) (o GMB-SOCKET))
```

```
  (setf (comp-GMB-SOCKET c) (orel::class-remove-list (comp-GMB-
SOCKET c) o)))

(defmethod add-GMB-MAPPING ((c GMB) (o GMB-MAPPING))
  (setf (comp-GMB-MAPPING c) (cons o (comp-GMB-MAPPING c)))
  (setf (comp-parent o) c))

(defmethod del-GMB-MAPPING ((c GMB) (o GMB-MAPPING))
  (setf (comp-GMB-MAPPING c) (orel::class-remove-list (comp-GMB-
MAPPING c) o)))

(defmethod add-DATA-GRAPH ((c GMB) (o DATA-GRAPH))
  (setf (comp-DATA-GRAPH c) (cons o (comp-DATA-GRAPH c)))
  (setf (comp-parent o) c))

(defmethod del-DATA-GRAPH ((c GMB) (o DATA-GRAPH))
  (setf (comp-DATA-GRAPH c) (orel::class-remove-list (comp-DATA-
GRAPH c) o)))

(defmethod add-CONTROL-GRAPH ((c GMB) (o CONTROL-GRAPH))
  (setf (comp-CONTROL-GRAPH c) (cons o (comp-CONTROL-GRAPH c)))
  (setf (comp-parent o) c))

(defmethod del-CONTROL-GRAPH ((c GMB) (o CONTROL-GRAPH))
  (setf (comp-CONTROL-GRAPH c) (orel::class-remove-list (comp-
```

```
CONTROL-GRAPH c) o)))


(defmethod orel:components-of ((o GMB))
  (append
           (comp-GMB-SOCKET o)
           (comp-GMB-MAPPING o)
           (comp-DATA-GRAPH o)
           (comp-CONTROL-GRAPH o)))
```

References

Robert Scheifler et al. [1988], "CLX," *On-Line Documentation, X Window System Version 11 Release 3*.

Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou & Hyoung-Joo Kim [January 1987], "Data Model Issues for Object-Oriented Applications," *ACM Transactions on Office Information Systems* 5, 3–26.

Philip A. Bernstein, Vassos Hadzilacos & Nathan Goodman [1987], *Concurrency Control and Recovery in Database Systems*, Addisson-Wesley Publishing Company, Reading Massachusetts.

Daniel M. Berry [January 1984], "On the Use of ADA as a Module Interface Description," *Proceedings, Hawaii International Conference on Systems Science*.

Daniel M. Berry, Gerald Estrin & Maria H. Penedo [March, 1981], "An Algorithm to select Code Skeleton Generation for Concurrent Systems," IEEE, Proceedings of the IEEE 5th International Software Engineering Conference , San Diego, California.

D.G. Bobrow, S. Mittal & M.J. Stefik [September 1986], "Expert Systems: Perils and Promise," *Communications of the ACM*.

Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales & David A. Moon [February 1988], "Common Lisp Object System Specification," *Limited Distribution Draft*.

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

O. P. Buneman & E. K. Clemons [October 1979], "Efficiently Monitoring Relational Databases," *ACM Trans. Database Systems* 3, 353–387.

Peter Pin-Shan Chen [March 1976], "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactions on Database Systems* 1, 9–36.

J. Conklin [September 1987], "Hypertext: An Introduction and Survey," *Computer* 20, 17–41.

O.J. Dahl & K. Nygaard [1966], "SIMULA— an Algol Based Simulation Language," *Comm. ACM* 9, 671–678.

D. Decouchant [1989], "A Distributed Object Manager for the Smalltalk-80 System," in *Object-Oriented Concepts, Databases, and Applications*, Won Kim & Frederick H. Lochovsky, eds., ACM Press, 487–520.

U.S. Department of Defense [February 1983], *Ada Programming Language Reference Manual (ANSI/MIL-STD-1815A)*, U.S. Government Printing Office.

N. Derret, W. Kent & P. Lyngbaek [1985], "Some Aspects of Operation in an Object-Oriented Database," *Database Eng.* 8, 66–74.

D. Engelbart [February 28 - March 1, 1984], "Authorship Provisions in AUGMENT," *IEEE COMPCON Digest*, San Francisco, California.

D. Engelbart & W. English [1968], "A Research Center for Augmenting Human Intellect," *AFIPS Conference Proceedings* 33 Part 1, 395–410.

G. Estrin [1978], "A Methodology for Design of Digital Systems - Supported by SARA at the Age of One," *AFIPS Conference Proceedings* 47, 313–324.

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.

D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G.Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan & M.C. Shan [January 1987], "Iris: An Object-Oriented Database Management System," *ACM Trans. on Office Information Systems* 5, 48–69.

J. D. Foley & A. vanDam [1982], *Fundamentals of Interactive Computer Graphics*, Addisson-Wesley, Reading, Massachusetts.

James Foley, Won Chul Kim, Srdyan Kovacevic & Kevin Murray [January, 1989], "Defining Interfaces at a HIgh Level of Abstraction," *Computer* 6, 25–36.

Gregg Foster [December 1986], "Collaborative Systems and Multi-user Interfaces: Computer-Based Tools for Cooperative work," Computer Science Division, University of California at Berkeley, Doctoral Dissertation.

Adele Goldberg [1984], *Smalltalk-80: The Interactive Programming Environment*, Addisson-Wesley, Reading, Massachusetts.

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

I. Greif & S. Sarin [April 1987], "Data Sharing in Group Work," *ACM Transactions on Office Information Systems* 5, 187–211.

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

D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman & A. Shtul-Trauring [April 1988], "STATEMATE: A Working Environment for the Development of Complex Reactive Systems.," *Proceedings of the 10th International Conference on Software Engineering*, Singapore.

Dennis Heimbigner & Dennis McLeod [July, 1985], "A Federated Architecture for Information Management," *ACM Transactions on Office Information Systems* 3, 253–278.

P.B. Henderson & D. Notkin [November 1987], "Integrated Design and Programming Environments," *Computer*.

C.E. Hewitt [1977], "Viewing Control Structures as Patterns of Passing Messages," *Jopurnal of Artificial Intelligence* 8, 323–364.

Ralph D. Hill [May 1987], "Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction," Computer Systems Research Institute, University of Toronto, CSRI-197, Toronto, Canada.

Anatol W. Holt [April 1988], "Diplans: A New Language for the Study and Implementation of Coordination," *ACM Transactions on Office Information Systems* 6, 109–125.

Scott E. Hudson & Roger King [April 1986], "A Generator for Direct Manipulation Office Systems," *ACM Transactions on Office Information Systems* 4, 132–163.

IDE, "Troll User's Manual," in *In StP User's Manual*, Interactive Development Environments, San Francisco, California.

IFIP 8.4 Group [August 24 and 25, 1989], "Groupware Technology Workshop," Palo Alto, California.

Lucid Inc, [August, 1988], *Lucid CommonLisp Documentation*, Lucid, Inc..

R. J. K. Jacob [August, 1985], "A State-Transition Diagram Language for Visual Programming," *Computer* 2, 51–59.

R. J. K. Jacob [October, 1986], "A Specification Language for Direct-Manipulation Interfaces," *ACM Transactions on Graphics.*

KCL [Not datedd], "KCL (Kyoto CommonLisp)," in *Public Domain Implementation of CommonLisp.*

Sonya Keene [1989], *Object-Oriented Programming in Common Lisp*, Addisson-Wesley, Reading, Ma..

Kerry Kimbrough & LaMott Oren [1988], "Common Lisp User Interface Environment," *Manual.*

Henry F. Korth, Won Kim & Francois Bancilhon, "On Long Duration CAD Transactions," Private Communication, 1987.

D. Landis [February 1988], *CADIS: A Kernel Approach Toward the Development of Intelligent Data Management Support of Computer Aided Design Systems*, University of California, Los Angeles.

Poman R. Leung [March 1989], "Database Kernel of SARA/CDE," Computer Science Department, University Of California, Undistributed draft, Los Angeles.

H. Lieberman [1981], "A Preview of Act 1," Massachusetts Institute of Technology, Artificial Intelligence Memo No. 625, Cambridge, Massachusetts.

F.H. Lochovsky [January 1987], "Editorial: Introduction to the Special Issue," *ACM Transactions on Office Systems* 5, 1–2.

J.Eliot B. Moss [June 1989], "Addressing Large Distributed Collections of Objects: The Mneme Project's Approach," Object Oriented System Laboratory, Department of Computer and Information Sience, University of Massachusets, COINS Technical Report 89-68, Amherst, MA 01003.

Brad A. Myers [1988], *Creating User Interfaces by Demonstration*, Academic Press, Inc., Boston, MA.

Brad A. Myers [January 1989], "User-Interface Tools: Introduction and Survey," *Software* 6, 15–25.

D.R. Olsen [July 1984], "Pushdown Automata for User Interface Management," *ACM Trans. on Graphics* 3, 177–203.

Maria H. Penedo [1981], "The use of a Module Interface Description in the Synthesis of Reliable Software Systems," Computer Science Department, University Of California, Los Angeles, Ph.D. Dissertation, Los Angeles, California.

A. Purdy, B. Schuchardt & D. Maier [January 1987], "Integrating an Object-Server with other Worlds," *ACM Transactions on Office Systems* 5, 27–47.

T. Reps, T. Teitelbaum & A. Demers [July, 1983], "Incremental Context-Dependent Analysis for Language-Based Editors," *ACM Transactions on Programming Languages and Systems* 5, 449–477.

Carles Rich & Richard C. Waters [November 1988], "The Programmer's Apprentice: A Research Overview," *Computer* 21, 10–25.

S. Sarin & I. Greif [October 1985], "Computer-Based Real-Time Conferencing Systems," *Computer* 18, 33–45.

Rober Scheiffler & James Gettys [1986], "The X Window System," *ACM Transaction in Computer Graphics* 5, 110–1141.

Ben Schneidermann [August 1983], "Direct Manipulation: A Step Beyond Programming Languages," *Computer* 16, 57–69.

R. Seliger [September 1985], "The Design and Implementation of a Distributed Program for Collaborative Editing," Laboratory of Computer Science, MIT, Technical Report TR-350, Cambridge, Massachusetts.

Stephen Slade [1987], *The T Programming Language*, Prentice-Hall, Englewood Cliffs, N.J..

Richard Stallman [March 1981], "EMACS, the Extensible, Customizable Self-Documenting Display Editor," MIT, AI Memo 519a, Cambridge, Massachusetts.

Guy Steele Jr. [1984], *Common Lisp*, Digital Press, Bedford, Ma..

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.

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.

Mark J. Stefik & Daniel G. Bobrow [1983], "The Loops Manual," Xerox PARC , Technical Note CSL-83-7, Palo Alto, California.

W.P. Stevens & G.J. Myers [1974], "Structured Design," *IBM System Journal* 13, 115–139.

Pedro Szekely [January 1988], "Separating the User Interface from the Functionality of Application Programs," Carnegie Mellon University, Technical Report CMU-CS-88-101, Pittsburgh, Pennsylvania.

R. Taylor, L. Clarke, L. Osterweil, J. Wileden & M. Young [April 1986], "Arcadia: A Software Development Environment Research Project," *IEEE 1986 ADA Applications and Environments Conference*, Miami Beach, Florida.

R.H. Thomas, H.C. Fordsick, T.R. Crowley, R.W. Schaaf, R.S. Tomlinson, V.M. Travers & G.G. Robertson [1988], "Diamond: A Multimedia Message System Built on a Distributed Architecture," in *Computer-Supported Cooperative Work: A Book of Readings*, I. Greif, ed., Morgan Kaufmann Publishers, Inc., San Mateo, California, 509–532.

Walter F. Tichy [September 1982], "Design, Implementation and Evaluation of a Revision Control System," Proceedings of the Sixth International Conferencing on Software Engineering, Tokyo, Japan.

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

A. I. Wasserman & D. T. Shewmake [December, 1982], "Rapid Prototyping of Interactive Information Systems," *SIGSoft Software Engineering Notes*.

A.I. Wasserman & P.A. Pircher [January, 1987], "A Graphical, Extensible Integrated Environment for Software Development," *Proceedings, 2nd Symposium on Practical Software Development Environments, ACM SIGPLAN Notices* 22, 131–142.

Anthony I. Wasserman, Peter A Pircher & Robert J Muller [1989], "An Object-Oriented Structured Design Method for Code Generation," Interactive Development Environments, Inc., Unpublished draft.

T. Winograd [December 1986], "A Language/Action Perspective on the Design of Cooperative Work," *Proceedings of the Conference on Computer-Supported Cooperative Work*, Texas.

T. Winograd & F. Flores [1986], *Understanding Computers and Cognition*, Addison-Wesley Publishing Company, Inc..

Duane R. Worley [June 1986], "A Methodology, Specification Language and Automated Support for Computer Aided Design Systems," Computer Science Department, University Of California, Technical Report CSD-860038, Los Angeles.