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

EXECUTABLE SPECIFICATIONS OF MULTI-USER INTERFACES

Y. Eterovic

July 1992
CSD-920038

UNIVERSITY OF CALIFORNIA
Los Angeles

Executable Specifications of Multi-Application Multi-User Interfaces

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

by

Yadran Eterovic

1992

The dissertation of Yadran Eterovic is approved.

[illegible]

Rajive Bagrodia

[illegible]

Phillip Bonacich

[illegible]

Michel Melkanoff

[illegible]

George Stiny

[illegible]

Gerald Estrin, Committee Chair

University of California, Los Angeles

1992

To Carmen Gloria, Sebastián and Magdalena
for all their love, support, understanding and patience, which made this effort worthwhile.

Contents

List of Figures

ACKNOWLEDGMENTS

I want to express my deepest gratitude to mi advisor, professor Gerald Estrin, for all his guidance during the past five years, without which this work could have not been done at all. Professor Estrin always had time for my questions and always had good answers for them. His continuous, dedicated and sincere concern for my work, but also for me and my family taught me a completely new lesson on being a professor.

I also want to thank the other members of my dissertation committee, professors Rajive Bagrodia, Phillip Bonacich, Michel Melkanoff and George Stiny. They provided interesting comments and suggestions which helped me improve the quality of my work.

Working in the coSARA group has been a pleasure thanks to Mannesh (Manu) Dhagat, Sergio Mujica, Richard Poman Leung, Elsie Wu, Katie Chong, Steven Berson and Ivan Tou. My friendship with Manu was born and grew during this time. My friendship with Sergio became only stronger.

My studies at UCLA were financed in part by Universidad Católica de Chile, by the Government of Chile through its Oficina de Planificación Nacional, and by the Organization of American States.

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

Finally, I want to thank my mother for always being there, geographically distant during these years, but otherwise very close.

VITA

January 26, 1958	Born, Santiago, Chile
1982	Electrical Engineer Catholic University of Chile
1984	Teaching Assistant Computer Science Department University of California, Los Angeles
1985	M.S., Computer Science University of California, Los Angeles
1985-1987	Assistant Professor Department of Computer Science Catholic University of Chile
1985-1987	Assistant Director, Student Affairs School of Engineering Catholic University of Chile
1988-1992	Research Assistant Computer Science Department University of California, Los Angeles

ABSTRACT OF THE DISSERTATION

Executable Specifications of
Multi-Application Multi-User Interfaces

by

Yadran Eterovic
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1992
Professor Gerald Estrin, Chair

This work deals with the problems of modeling, specification, and construction of executable prototypes of multi-application multi-user interfaces, which are graphical, interactive, collaborative, based on the direct-manipulation style of interaction, and dynamically reconfigurable. The contributions of this work with respect to these interfaces are a model and an object-oriented development methodology to build executable prototypes of the interfaces from the specification of their single-user versions. The model and the development methodology support application-independent control mechanisms used for the specification of coordination among multiple users concurrently sharing an application. They also support configuration mechanisms, based on extensions to formal graphical modeling languages, used for the specification of interfaces whose structure and behavior can change dynamically during their operation.

We describe the model in terms of the structure, behavior and communication of screen objects and abstract representations of the applications. By refining the model we develop a small collection of user-application dialogs which cover interactions used in graphical, interactive and direct-manipulation interfaces. The UCLA Structural Model and Graph Model of Behavior modeling languages are used to represent the model and they provide a basis for simulating and testing interface designs during early prototyping phases..

Those languages plus the Object RELation language are used to specify the organization and behavior of the classes composing an interface. Applications take here the form of classes and methods that can be applied to instances of the classes through the interface. Common Lisp User Interface Environment contacts and software modules from an extensible library are used to create instances, and a linking procedure is provided which automatically assigns the appropriate class behavior to the instances. As a result, end users and designers can execute the models by operating real input devices on real screen objects.

Explicit mutual exclusion mechanisms included in models of screen objects, and therefore independent of applications, are used to coordinate activities of multiple users trying to concurrently access the functionality and data of the same application. The regularity of the control structures defining the behavior of the screen objects or coordinating activities among screen objects is used to provide the ability of an executable interface specification to reconfigure itself during a collaborative session, as new applications become active and new users join the session.

1 Introduction

In this work, we deal with the problems of modeling, specification, and development of executable prototypes of multi-application multi-user interfaces, which are graphical, interactive, collaborative, based on the direct-manipulation style of interaction, and dynamically reconfigurable.

The central contributions of this work with respect to multi-application multi-user interfaces are an operational model and an object-oriented development methodology to build executable prototypes of these interfaces from graphical and formal specifications of their single-user versions. The model and the development methodology support application-independent control mechanisms which can be used for the executable specification of coordination among multiple users sharing an application concurrently. They also support configuration mechanisms, based on extensions to formal graphic modeling languages, which can be used for the executable specification of interfaces whose structure and behavior can change dynamically during their operation.

1.1 Definitions and Motivation

The *multi-application multi-user interface* of a collaborative computer system is the component of the system which allows and controls the interaction between multiple users and the multiple applications of the system. Fig. 1 shows the concept of such an interface in the case of three collaborating users concurrently sharing one application, a graphical block diagram editor. Each user's workstation screen displays two windows with functionally identical contents and sensitive to the same set of user's input actions. These interfaces can be characterized as:

- *Graphical*, that is, the visible objects on the workstations' screens represent buttons, menus, dialog boxes, scroll bars, drawing windows, etc., or are elaborate graphical representations of data objects;
- Highly *interactive*, that is, users expect rapid and almost continuous syntactic and semantic feedback in response to their actions;
- Based on the *direct-manipulation* style of interaction, that is, users input their actions by pointing at (the graphical representations of) objects with a mouse's cursor and then clicking or dragging the mouse;
- *Collaborative*, that is, multiple users can concurrently share one or more applications and their data; and
- *Dynamically reconfigurable*, that is, users can join or leave a collaborative session at any time, and applications can become active or inactive, under user control, at any time during a collaborative session, changing the configuration of the interface in both cases.

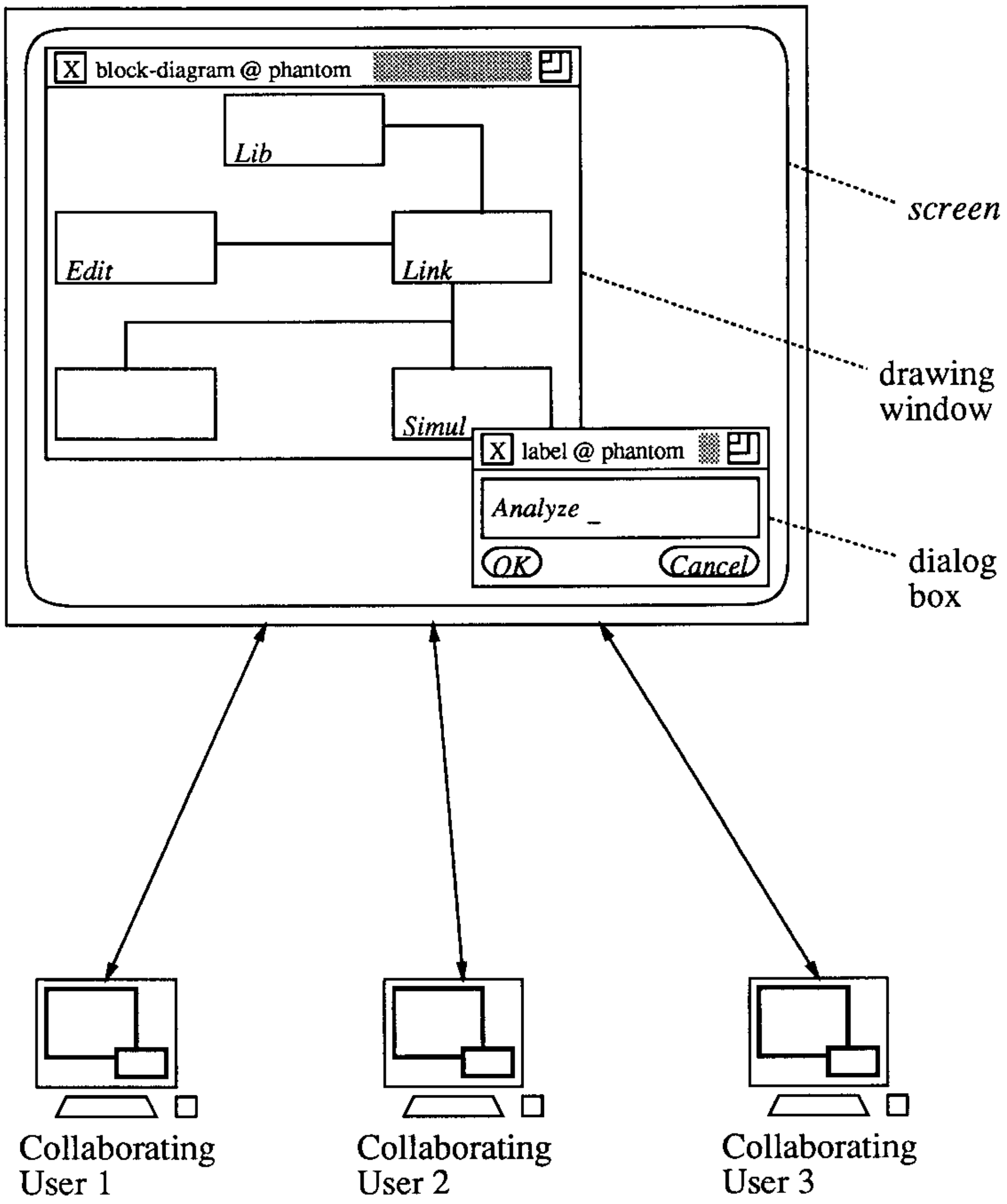


Figure 1: The concept of a multi-application multi-user interface.

In general, the process of building these user interfaces is fairly complex and expensive, and numerous authors agree that the only reliable way to produce them is by an iterative design process, that is, to test prototypes with users and modify the design based on their comments [Harbert et al., 1990; Mulligan et. al, 1991; Myers, 1989b]. If this technique is to be effective, designs must be reasonably easy to construct and execute for evaluation. Both user interface designers and users should be able to: simulate the operation of a proposed user interface, change its definition if the original design is unacceptable, and resume execution using the modified definition.

A user interface development system should be easy to use and fully expressive. Easy-to-use systems usually provide easily manipulated entity libraries but their flexibility is restricted. Some systems provide a low level specification language, which is more difficult to use but has greater flexibility and power ([Myers et al., 1990] cites the Macintosh Toolbox and the X Toolkit as examples of this case). A system that combines both approaches gives designers access to libraries of reusable entities, and at the same time lets them modify and extend those libraries with a lower level specification language. Libraries are used for most work, turning to lower level techniques only to specialize entities or create completely new ones.

1.2 Contributions

The central contributions of this work with respect to multi-application multi-user interfaces are:

- *A formal operational model of the interfaces.* The model explicitly includes the screen objects with which the users interact directly, the application models that support the sharing of information between the interface and the applications, and the communication schemes among these components. Screen objects are represented in more detail than in any other existing model in terms of internal behavior and communication of control and data. Still, they are completely independent of any particular implementation. The application models are defined as abstract, object-oriented representations of the functionality of applications as seen from the interface's point of view, and truly support the shared data model that has been proposed for interfaces based on the direct-manipulation style of interaction [Hudson, 1987].

- *An object-oriented specification and development methodology to build executable prototypes of the interfaces.* For specification purposes, the methodology uses the Object RELation (OREL) modeling language [Mujica, 1991] to describe class diagrams, and the Structural Model (SM) and Graph Model of Behavior (GMB) modeling languages [Estrin et al., 1986] to describe the behavior of classes. For development purposes the methodology uses an extensible library of software modules to assign interpretation to behavioral models, to link class instances to these models, and to map real screen objects to their class instance representations. The same formalisms are used to model a *test environment* which is used to generate inputs and test the behavior of models.

Besides the characteristics mentioned above, the model and the development methodology support:

- Application-independent control mechanisms which can be used for the executable specification of coordination among multiple users sharing concurrently an application. The mechanisms are based on mutually exclusive and/or strictly sequential operation among functionally equivalent screen objects, and can be easily encapsulated in models of the screen objects.
- Configuration mechanisms, based on structural and behavioral extensions to the SM and GMB modeling languages, which can be used for the executable specification of interfaces whose structure and behavior can change dynamically during their operation.

1.3 The Dissertation

Chapter 2 reviews the work done by other researchers in the areas of modeling, specification and construction of user interfaces, presenting the models, notations and environments that are most relevant to our own work. Chapter 3 introduces, by means of a simple example, our operational model of multi-application multi-user interfaces, described in terms of the languages SM and GMB. The model is at the same time a combination, an extension and a refinement of well-known models of mainly single-user interfaces. The model deals simply and explicitly with the aspects of concurrency, collaboration, and reconfigurability found in the multi-user case.

Chapter 4 describes our object-oriented methodology to build executable prototypes of multi-user interfaces from the graphical, formal specifications of their single-user versions. The methodology is based on the operational model introduced in Chapter 3 and on our view of these interfaces and their components as objects belonging to specific classes, participating in several relations, and communicating through different types of messages. The language OREL is used to define the class diagram of a multi-user interface.

Chapter 5 presents the approaches taken in our model to deal with the issues of concurrency and dynamic reconfiguration of multi-user interfaces. First, we introduce a flexible, application-independent mechanism to coordinate the interactions of multiple users concurrently sharing the same application. The mechanism is based on mutually exclusive or strictly sequential operation of equivalent screen objects. Then, based on the internal control structure of the screen objects and on the interaction coordination mechanism, we introduce two dynamic reconfiguration mechanisms that allow an interface to change during its operation as a consequence of variable numbers of users and applications.

Chapter 6 presents a complete, realistic example of the specification and execution of the prototype of a multi-user interface for a graphical application. The same example is used to extend the reconfiguration mechanisms introduced in Chapter 5 by describing the application-independent reconfiguration of the interface between the screen objects and the application. We conclude this work by summarizing our contributions to the field of executable specifications of multi-application multi-user interfaces, pointing out the strengths and weaknesses of our approach, and mentioning some new problems in the field open for research.

2 Related Work

In this chapter we summarize the work done by other researchers in the areas of modeling, specification, and construction of user interfaces. Because the amount of such work is vast, we present only those models, notations, and environments that are most relevant to our own work.

2.1 User Interface Models

2.1.1 The Seeheim Model

A general model of single-user interfaces is the *Seeheim model* [Green, 1985], which divides a user interface into three components, as shown in Fig. 2(a): presentation, dialog control, and application model. These three components communicate by passing *tokens*, which consist of a name and a collection of data values.

The *presentation* deals with: the physical representation of the user interface (including input and output devices such as mouse, keyboard and screen), layout of screen objects, interaction techniques and display techniques. It detects user-generated input events, produced by the operation of the mouse and the keyboard, and displays the screen objects and their contents on the screen. It constitutes the lexical level of the interface.

The *dialog control* deals with the structure of individual commands and the overall dialog between a user and an application. It converts input tokens received from the presentation into calls to application procedures, and generates requests to the presentation to display objects on the screen according to data received from the application. It constitutes the syntactic level of the interface.

Finally, the *application model* is an abstract, implementation-independent description of an application, representing the interface between a user interface and an actual application. It handles invocation of application procedures and communication of requests and data from the application to the user interface. It removes a burden from application designers because frequently required capabilities do not have to be designed for each application.

2.1.2 The Direct Manipulation Model

The *direct manipulation model* of user interfaces [Hudson, 1987] is a modification of the Seeheim model in order to better capture the characteristics of single-user interfaces based on the direct manipulation style of interaction. It consists of only two components, as shown in Fig. 2(b), and does not include an explicit dialog control.

The *presentation* manages the graphical images and input techniques described by a collection of *presentation plans*, which are abstract representations of inputs and outputs including their lexical and syntactic aspects. The *shared data model*

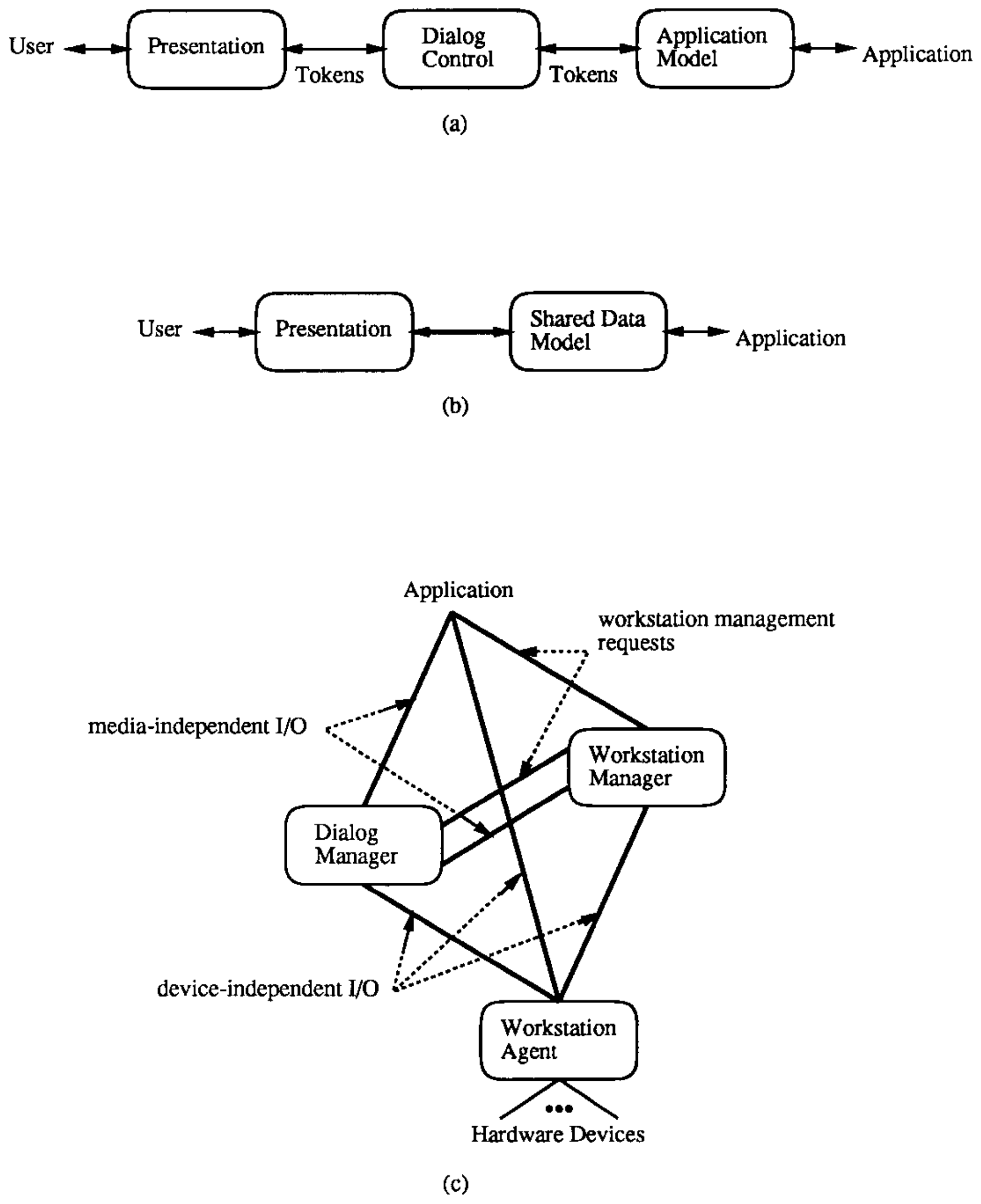


Figure 2: User interface models: (a) Seeheim; (b) Direct Manipulation; (c) Reference.

is a set of structured active shared objects, which do not simply passively store data, but also react to changes in ways that reflect the semantics of the application, allowing a convenient vehicle both for implementing semantic feedback and for automatically notifying the application when objects change.

2.1.3 The Reference Model

The *reference model* [Lantz et al., 1987; Lantz, 1987] is a model for the implementation of interactive software, providing a framework within which alternative implementations can be compared, rather than proposing a specific implementation. It considers more explicitly than the previous models situations in which: the application and the user interface run on different machines; users interact with several applications at the same time; users use the computer to support collaborative work; and the interface can be statically and/or dynamically reconfigured to support different users or application environments.

The reference model consists of three major components, as shown in Fig. 2(c). The *workstation agent* manages the hardware devices, which provide the "raw" interface to the user, presenting a device-independent interface to clients in terms of relatively low-level I/O primitives. It also provides the basic support for multitasking, multiplexing the various devices between multiple clients according to a policy determined by the workstation manager. The *dialog manager* provides "true" dialogs between users and applications. It composes the I/O primitives provided by the workstation agent into interaction techniques and selects particular interaction techniques to satisfy application-specific tasks. Thus, it provides the application with a media-independent interface that supports the invocation of application methods and handles their responses. Finally, the *workstation manager*: handles the meta-dialog that allows users engaged in multiple simultaneous dialogs to switch from one dialog to another, provides (through the dialog manager) the user interface to this meta-dialog, and sets the policy for sharing devices.

2.2 User-Application Dialog Specifications

2.2.1 Interaction Objects

Interaction Objects [Jacob, 1986] is an object-oriented language for specifying the syntax of the dialog between users and applications in a user interface based on the direct manipulation style of interaction. Such an interface comprises a collection of many relatively simple individual dialogs, each of which is described by an interaction object. The language supports multiple inheritance (an object can inherit properties from several other objects), composition (an object can contain one or more smaller objects), and a restricted form of communication between a composite object and its components.

The main elements of an interaction object are input tokens, output tokens and dialog diagrams. A *token* is an abstraction and a definition of a low-level input or output signal, and it may carry data. Examples of input tokens are typing a character on the keyboard and moving or clicking the mouse. Examples of output tokens are highlighting a region of the screen and drawing or erasing a figure. A *dialog diagram* is an extended state transition diagram. The label of a transition in this diagram can be an input or output token or a condition to be tested. In both cases, it can be followed by the name of another diagram to be called as a subroutine, or by a sequence of actions, such as assignments to local variables and calls to application procedures.

The collection of interaction objects of an interface is organized like a system of coroutines. At any point during the operation of the interface only one dialog diagram, and hence only one interaction object, is active. When an input token is produced by the user, the diagram checks to see if its current state has an outgoing transition labeled with that token. In that case, that transition is taken. Otherwise, the diagram is suspended and a different dialog diagram, usually in a different interaction object and currently suspended in the appropriate state, is activated and its transition taken. When a dialog diagram is suspended, it remembers the values of its local variables and its current state.

2.2.2 The Event Model

The *event model* [Green, 1986] is an extension of the concept of input events found in graphic packages, in which the input devices are viewed as sources of events of some predefined types. In the event model there is an arbitrary number of event types, which can be extended by the programmer, and the events can be generated not only by the input devices but also by the dialog itself. When an event occurs, it is sent to one or more *event handlers*, which are processes capable of handling certain types of events. When an event handler receives an event, it executes a procedure called an event procedure. This procedure can assign the value of an expression to a local variable, execute one of two sets of statements based on the value of a condition, create an event and send it to an event handler, create or destroy an event handler, or call an application procedure.

A user interface is described by a set of event handler templates. Each template defines the behavior of a subset of the event handlers the interface can use. It basically contains a list of the events that can be handled by the event handlers created from the template, and one event procedure for each event. Conceptually, all the event handlers of an interface execute concurrently, processing events as they arrive, and the same event can be processed by more than one event handler at the same time. The events sent to the same event handler are processed in the order in which they are sent, but there are no constraints on the time ordering of events sent to different event handlers. Because processing an event, that is, executing the corresponding event procedure, is an atomic operation, an event handler can only process one event at a time, and so it can be viewed as a monitor.

2.2.3 Sassafras's Event-Response Language

The goal of the *Sassafras* UIMS [Hill, 1986] is to provide tools that aid in the development of better user interfaces by allowing rapid iteration of the implementation, testing, and redesign steps involved in the development process. Sassafras makes no effort to make the interface specification process easier than it is with other UIMSs, but concentrates on extending the range of interfaces that can be supported, specially by incorporating concurrency at three levels: input, output, and dialog. The two key components of Sassafras are the Event-Response Language and the Local Event Broadcast Method.

The *Event-Response Language* (ERL) is a language for specifying the syntax of the dialog between users and applications. Its main elements are incoming events, outgoing events and flags. All input to an ERL processor comes through a queue of incoming events. An event is a signal that something has occurred, and it may carry data. Flags are local variables used to encode the state of the system and to control its execution. An ERL specification consists of a list of rules. Each rule specifies a response to some event or an action to be taken when some state is entered.

A rule consists of a condition followed by an action. The condition is either the name of an event and a list of flags (a regular rule) or simply a list of flags (an ϵ rule). The action is a list of flags, events, and assignments. The rule is firable when all flags in the condition are raised, and the event (if any) is at the head of the event queue. When the rule fires, the flags in the condition are lowered; then, in the order in which they appear in the action, all flags are raised, any events are sent (outgoing events), and the assignments are processed. Execution of an ERL specification proceeds by repeating the following two steps: (1) fire all firable ϵ rules, and (2) fire all firable regular rules and remove the corresponding event from the queue.

The *Local Event Broadcast Method* (LEBM) is a run-time structure that supports communication and synchronization among the components of an interactive system, and that schedules their execution. Its primary elements are modules and clusters. Modules represent units of code (for example, an ERL specification) that exchange information and synchronize other modules via the LEBM. Clusters are groups of modules linked by a single instance of the LEBM. Normally, there is one cluster for each interface. LEBM allows modules in a cluster to communicate with each other by sending events to a controller, where they are queued and processed by a two-step loop: (1) wait until all modules are idle, thus supporting synchronization of modules, and (2) send the first event in the queue to all modules that will accept it.

2.2.4 Garnet's Interactors

Garnet's *Interactors* [Myers et al., 1990; Myers, 1989a], are encapsulations of input device behaviors that allow a high degree of customizing by applications. Interactors are look independent, let designers separate the details of object behavior from the application and from the output graphics, support multiple input devices operating in parallel, let users operate on any number of different applications, and together with Garnet's graphical object system hide the complexities of X Windows graphics and event handling.

There are only six types of interactors, which cover all the kinds of interactions used in graphical user interfaces based on keyboard and mouse: choosing items from a set; moving or changing the size of an object; entering an arbitrary number of new points; calculating the angle at which the mouse moves around some point; capturing all the points the mouse goes through between start and end events; and inputting a small string of text.

All interactors run the same simple state machine that handles the starting, stopping, aborting, and suspending (while outside the active region) activities. The parameters to the interactor determine what events cause the transitions and what actions are taken when they happen, but the designer does not have to program the control flow. Each interactor has default actions when starting, running and stopping, which can be overridden or suplemented by arbitrary Lisp procedures provided by the designer or the application.

2.3 User Interface Toolkits

A user interface toolkit is a library of interaction techniques. An interaction technique is a way of using a physical input device (such as mouse and keyboard) to input a value (such as command, position, name), along with the feedback that appears on the screen [Linton et al., 1989; Myers, 1989a]. Examples of interaction techniques are menus, scroll bars, and dialog boxes.

2.3.1 InterViews

InterViews [Linton et al., 1989] is a library of C++ classes that defines three categories of commonly used user interface objects and the appropriate composition strategies: interactive objects, structured graphics, and structured text. Each category is implemented as a hierarchy of object classes derived from a common base class, which defines the communication protocol for all objects in the hierarchy. Subclasses within each category allow hierarchical composition of object instances, and define the additional protocol needed by the elements in a composition, such as operations for inserting and removing elements and for propagating information through the composition.

Interactive objects such as buttons and menus manage some area of potential input and output on a workstation display. They are derived from the *interactor* class, and are composed by *scenes*, which define specific composition semantics such as tiling or overlapping. Interactors provide operations to handle input events, support nonlinear deformations, and are customizable. Structured graphics objects such as circles and polygons are derived from the *graphic* class, and are composed by *pictures*, which provide a common coordinate system and graphical context for their components. Structured text objects such as words and whitespace are derived from the *text* class, and are composed by *clauses*, which define common strategies for arranging components to fill available space.

Interviews distinguishes between the objects that implement the interface, called *views*, and the objects that encapsulate the underlying data, called *subjects*, thus allowing programmers to present different, independently customizable interfaces to the same data. Views are implemented with interactors, graphics and texts, and define an `update` operation that reconciles a view's appearance with the current state of the subject. Subjects are derived from a subject class and maintain a list of their views. Calling `notify` on a subject in turn calls `update` on its views, which then update their appearance in response to a change in the subject.

2.3.2 CLUE

CLUE, the Common Lisp User Interface Environment [Kimbrough, 1989; Kimbrough and Oren, 1990] is a high-level object-oriented programming interface to the X Window System, extending the basic Common Lisp interface to X (CLX) [CLX, 1989] and based on the Common Lisp Object System (CLOS) [Keene, 1989]. It is a toolkit for constructing X user interfaces, modeled closely on the standard C toolkit (Xt, or the X Toolkit).

A CLUE program contains an event loop, a set of interface objects called *contacts*, and a set of application functions called *callbacks*. Each contact is an interface "agent" that is prepared to present application data, to accept user events which manipulate these data, and then to report the results back to the application using the callbacks. A callback consists of a name and a function. Contacts can be arranged in composites to implement objects such as control panels and dialog boxes. A composite implements a style of layout for its components, so that requests to change the geometry of a component are forwarded to the composite, which actually performs the resulting changes. In this way, the composite can arbitrate the competing requests of several components, implementing constraints among them, and a given layout style can be applied to any collection of contacts and can be changed without the knowledge of individual contacts.

CLUE's main job is handling user input events, such as pressing a key on the keyboard and clicking or moving the mouse. A call to `process-next-event`, at the beginning of the event loop, causes CLUE to read the next event from a given connection to a specific X server, and send it to the contact that is supposed to handle it. The receiving contact then translates the event, first by comparing it with each of the event specifications known to the contact, and then, when a specification is found that matches the event, by invoking the methods associated with this specification and representing well-defined contact behavior. Event specifications and the associated methods can be defined for both classes of contacts and individual contact instances. When a method computes a result that is important to an application, it invokes a callback name, which the application should have associated with a callback function. At this point, `process-next-event` returns.

2.4 User Interface Development Systems

A user interface development system is an integrated set of tools that help programmers design, implement and manage many aspects of interfaces [Linton et al., 1989; Myers, 1989a]. They may contain a toolkit, a dialog-control component to handle event sequencing and interaction techniques, a programming framework to guide and structure the interface code and application semantics, a mouse-based layout editor to specify the location of graphical elements, and an analysis component that evaluates the interface automatically or saves run-time information for later evaluation by the designer.

2.4.1 The User-Interface Design Environment

The *User-Interface Design Environment* (UIDE) [Foley et al., 1989] uses a knowledge-based representation of the user interface's conceptual design. This representation consists of the class hierarchy of objects in the system, their properties and the actions that can be performed on them, and the parameters, preconditions and postconditions for the actions. It can be used for several purposes: to produce a description of the design in the Interface Definition Language; to transform itself and the user interface into another, functionally equivalent interface; to provide input to the Simple User-Interface Management System (SUIMS) to implement the interface; to check the design for consistency and completeness (verify that the knowledge base has enough information for the transformations and the SUIMS to operate); to evaluate the design's speed of use; and to generate runtime help.

Particularly interesting are the transformation system and the SUIMS. UIDE can generate alternative interface designs that are slight variations on one another, by applying transformations to the knowledge-based representation [Foley, 1987]. The implemented transformations include: factoring (e.g., creating a selected object, command, or attribute); establishing a selected set (a generalization of the selected object concept) such that any command applies to all elements of the set; establishing initial values for factored parameters and default values for unfactored parameters; specializing and generalizing commands based on object and command hierarchies; and modifying the scope of some commands.

The SUIMS lets you define aspects of the interface such as presentation style, dialog syntax, and interaction techniques. It uses the knowledge-based representation and a knowledge base containing the interaction techniques necessary for the different interaction tasks. These include selection of commands, classes, instances, attributes and positions, and text and number input. For example, for the instance selection task the system offers the pointing, type-in, and menu interaction techniques. The SUIMS instantiates the objects defined in the knowledge-based representation, maintains their attribute values, and makes them available to the application's procedures through mechanisms provided by the software supporting the representation (the Inference Corp.'s Automated Reasoning Tool). The SUIMS cycles through a set of steps: update the screen, check preconditions and recognize enabled actions, accept the action selected by the user, process each parameter accepting their values in arbitrary order, confirm or cancel the action, execute the action, and evaluate postconditions.

2.4.2 Garnet

Garnet [Myers et al., 1990] helps designers (1) rapidly prototype different interactive, graphical, direct manipulation user interfaces, and (2) explore various interface metaphors during design. It reflects an emphasis on two aspescts of interface design: handling objects' runtime behavior, that is, how they change when the user operates on them; and handling all visual aspects of an interface, including its graphics and the contents of all application-specifc windows. Garnet contains low level tools (an object-oriented programming system, a constraint system, a graphical object system, a system for handling input, and a collection of gadgets) and high level tools (an interface builder (Lapidary), a menu and dialog box creation system, and a spreadsheet for specifying complex constraints).

The programming system supports a prototype-instance (rather than a class-instance) model for objects, which does not distinguish between instances and classes: any instance can be a prototype for other instances. Constraints are arbitrary Lisp expressions, stored in object slots, which represent relationships among objects that are maintained when the objects change. The graphical object system provides default values for all object properties, which can also be specified using the constraint system. It handles object drawing and erasing, minimizing the number of objects that are erased or redrawn, and can group objects into aggregates. If an aggregate is used as a prototype, changes to the aggregate are immediately reflected in all instances, including adding or deleting components, in which case the corresponding components are added or deleted from all instances.

Input handling is supported through the use of interactors [Myers, 1989a], which are encapsulations of input device behaviors that allow a high degree of customizing by applications. Interactors are briefly described in section 2.2.4. Garnet provides gadgets for menus, scroll bars, buttons and gauges, which have a number of parameters to let designers vary many aspects of their appearance and behavior. Lapidary provides a graphical front end to the low level tools. The designer, who does not have to be a programmer, can draw prototypes of application-specific graphical objects: entities the end user will manipulate, feedback showing selected objects, and dynamic feedback of objects. The designer can then specify the runtime behavior of these objects, using constraints and abstract descriptions of their response to the input devices. Lapidary generalizes from specific example pictures, letting the designer specify the graphics and behaviors using dialog boxes and by demostration. The dialog box creation system creates a dialog box or menu from a textual specification of its contents. This specification is look and feel independent and includes only the string labels to appear in the object and the type of input required. The spreadsheet lets designers enter arbitrary Lisp contraint expressions, and then monitor and debug their interface designs by watching spreadsheet values while the interface is running.

2.4.3 The UofA* UIMS

The goals of the UofA* UIMS [Singh and Green, 1991] are to help in the initial design of user interfaces, and to increase the ease of exploring different alternatives in interface designs. The approach taken is to enable designers to quickly produce and then refine working prototypes of the interfaces, and to quickly produce different interfaces for the same application. The UofA* UIMS consists of three subsystems: *Diction* accepts a high-level description of the semantic commands supported by the application and produces the interface's dialog control component and a second output; *Chisel* uses the second output from Diction, plus a display device description and user's preferences, to produce the interface's presentation component; *vu* is an interactive graphical facility that allows designers to refine the presentations produced by Chisel.

The command description accepted by Diction contains declarations of all the commands supported by the application. For each command, its name, parsing sequence (prefix, postfix, or nofix), selection type (arbitrary or fixed number of arguments), and arguments are specified. An argument specification lists the argument's name, type, and initial (for global arguments) or default (for local arguments) value. The dialog control components produced by Diction consist of program modules, one per command, called event handlers, which are used in the event model to describe these components. The event model and event handlers [Green, 1986] are briefly described in section 2.2.2.

Chisel's input consists of a dialog requirements file prepared by Diction, a device description selected from a library, and an optional user's preferences specification, the only input which has to defined by the designer. Once given all the inputs, which have the form of Lisp functions, Chisel selects interaction techniques, determines their attributes, and places the corresponding interaction objects (physical realizations of the interaction techniques) on the screen.

2.4.4 Chimera

Chimera [Wood and Gray, 1992] is a UIMS written in C and PostScript, built to facilitate rapid prototyping and dynamic reconfiguration of user interfaces. It consists of *Chisl*, a specification language, and *Chip*, an interpreter for Chisl. An interface specification is built up from several dialog units. Each unit consists of declarations of global and local variables, definitions of presentations and interactive objects, and options. Interactive objects are the means by which users interact with the dialog. They can be editable text-fields, buttons or any graphics image definable in PostScript. And they have three attributes, which can be changed at run-time, specifying if the object is visible, if it can be moved by a user, and if the user may interact with it. Options define the flow of control of the dialog. An option consists of a condition, involving the unit's variables and interactive objects, and a list of actions, specifying changes in the variables, presentation, and interactive objects' attributes.

2.4.5 Integral Help

Integral Help [Fenchel, 1982] is a technique by which interactive applications can be designed to automatically provide detailed syntactic assistance to their users. It has been applied to the SARA system and is based on the idea that the same program that recognizes valid inputs (the parser) is used to generate the assistance information. The SLR(1) grammar specification of the user input language is processed by an extended parser, called the parser/help generator program, to produce: (1) parse tables used to recognize the input to the application program and invoke appropriate semantic processing routines; (2) a help data base that contains information associating syntactic and semantic help information with the states of the parser and nonterminals of the grammar; and (3) a hard copy user reference manual clearly describing each language construct and command, and providing many examples.

If while using an application the user requests help information or makes a syntactic error, the help system is invoked. This system interacts with the parser to determine the state of the parse. If the current state has an associated nonterminal abstraction, that abstraction is used to produce an error or help message; otherwise, the procedure for the state most recently placed on the parser's stack is repeated. In the case of an error, the system presents the erroneous input line with a pointer to the invalid input symbol and prints a message containing the nonterminal abstraction. The parser discards the appropriate portion of the input line and then recovers to a state indicated by information associated with such nonterminal abstraction, or with the state most recently placed on the stack. If the user requests help anywhere on an input line, the system behaves as it does for errors to determine the nonterminal abstraction to be used. The user may select the type and amount of help information, making incremental requests at various levels of detail. The user can request help information for any symbol of the grammar at any time: for nonterminals, the current context is indicated by presenting the associated group of productions; for terminals, the production in which the terminal appears is presented. The help and error routines can be extended to provide varying degrees and types of assistance depending upon the experience levels and desires of each user or group of users.

2.5 coSARA's Formal Graphical Modeling Languages

2.5.1 OREL

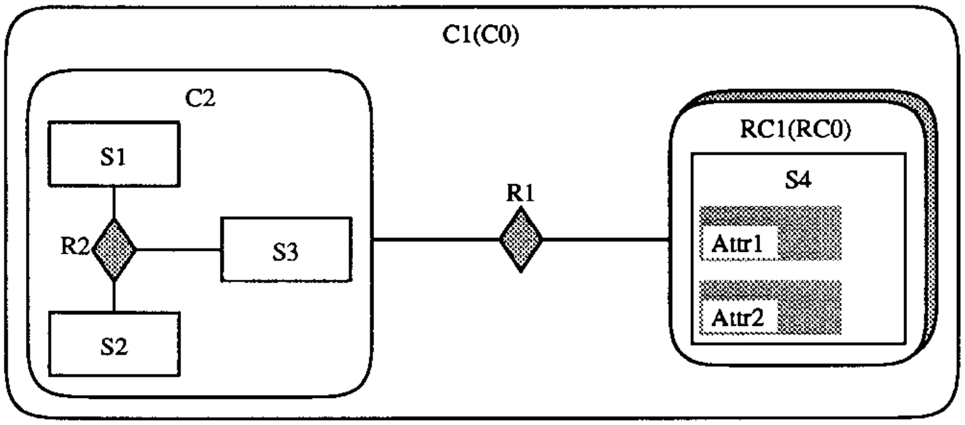
Object RELation (OREL) [Mujica, 1991], is a formal, graphical language, which provides six primitives to model object-oriented data: (1) simple classes, (2) composite classes, (3) recursive composite classes, (4) class attributes, (5) class inheritance, and (6) relations among classes. Fig. 3 shows the graphical representations of these primitives and exemplifies their use. A composite class is a class whose objects (class instances) contain a dynamically variable number of objects of another class, called the component class. A composite class can actually have several different component classes. A recursive composite class is a composite class in which one of the component classes is the composite class itself. A relation between two or more classes is a special class whose objects are collections of lists of objects of the classes participating in the relation. Each list is called a tuple and usually contains one object of each participant class.

A tool called the *OREL compiler* translates an OREL class model into the corresponding CLOS (Common Lisp Object System) code [Keene, 1989], including the class definitions, the functions to instantiate the classes (create objects), the methods to assign values to the objects' attribute slots and to read these values, the methods to add and remove the component objects of a composite object, the methods to create tuples and to add them to the relations, and the methods to find specific tuples in a relation and to remove them from the relation.

2.5.2 SM and GMB

Structural Model (SM) [Estrin et al., 1986] is a formal, graphical language, which provides three primitives to model the structure of a concurrent system: *modules*, which represent the system's components and subcomponents; *sockets*, which represent the modules' communication ports; and *interconnections*, which represent the communication paths between sockets in different modules.

Graph Model of Behavior (GMB) is a formal, graphical language to model the behavior of a concurrent system. GMB, supported by the GMB graphical editor, models three related domains of the behavior of a system: control, data, and interpretation, as shown in Fig. 4. The control domain is represented by a *control graph*, which models the flow of control among the events that occur in



C1 and C2 are composite classes (rounded-corner rectangles). C1, a subclass of class C0 (not shown), consists of classes C2 and RC1, and relation R1. C2 consists of classes S1, S2 and S3, and relation R2.

RC1 is a recursive composite class (rounded-corner rectangle with shadow). It is a subclass of class RC0 (not shown) and consists of classes RC1 (itself) and S4.

S1, S2 and S3 are simple classes (rectangles) without attributes, that is, representing classes already defined.

S4 is a simple class with 2 attributes: Attr1 and Attr2 (shaded rectangles).

R1 and R2 are relations (shaded diamonds). R1 relates 2 classes: C2 and RC1 R2 relates 3 classes: S1, S2 and S3.

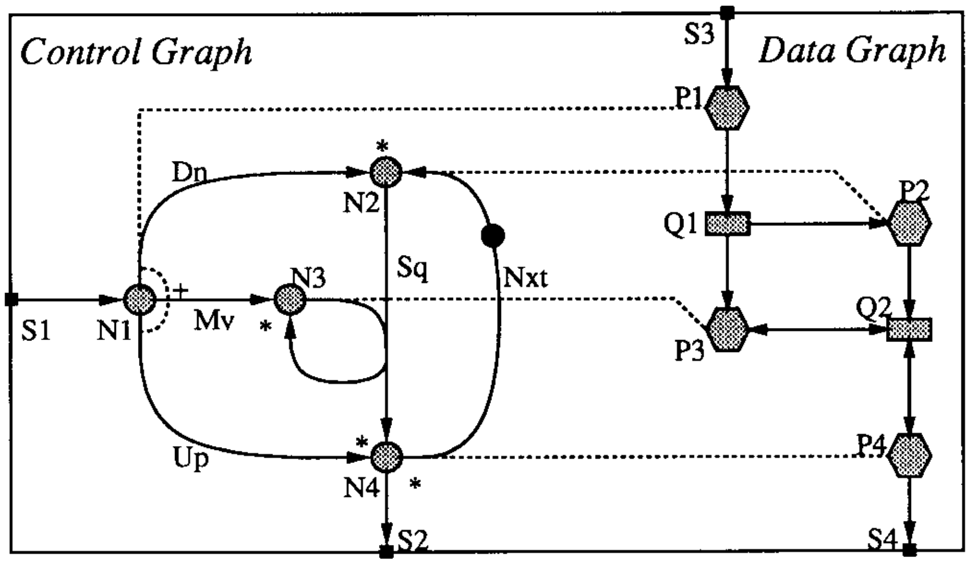
Inheritance is represented by enclosing the name of the superclass or superclasses in parentheses.

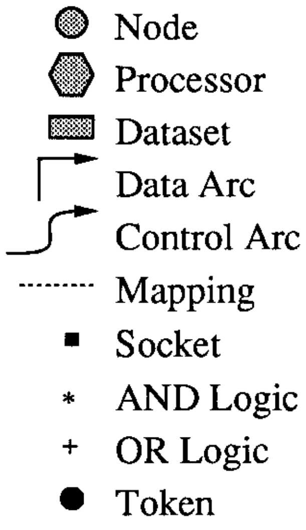
Figure 3: The primitives of OREL and an example of their use.

the system, similar to a Petri net. The events are represented by the *nodes* of the graph (`N1`, ..., `N4`), while the partial ordering of their activity is determined by directed *control arcs* connecting them (`Dn`, `Mv`, `Up`, `Sq`, `Nxt`). Control arcs can have multiple sources and multiple destinations; for example, arc `Sq` has two sources (nodes `N2` and `N3`) and two destinations (nodes `N3` and `N4`).

The data domain is represented by a *data graph*, which models the flow of data between computation units and data storages. The computation units are represented by *processors* (`P1`, ..., `P4`), the data storages by *datasets* (`Q1`, `Q2`), and the direction of flow of data is determined by directed *data arcs* connecting the processors to the datasets (unnamed in the figure). Finally, the interpretation domain associates with the data graph the values stored in the datasets and their types, and the computations implementing the activity of the processors, including control flow decisions, processing delays, and data transformations.

The *token machine* is an interpreter of GMB models. Its formal definition can





Interpretation

```
   DATASETS
Q1: point
Q2: (point . point)

   PROCESSORS
P1:
   (let ((event (ReceiveThru "S3")))
      (WriteTo "Q1" (data event))
      (typecase event
         (initiation (PlaceToken Dn))
         (continuation (PlaceToken Mv))
         (termination (PlaceToken Up))))
P2:
   (let ((p (ReadFrom "Q1")))
      (WriteTo "Q2" (list p nil)))
P3:
   (let ((fig (ReadFrom "Q2"))
         (p (first fig))
         (q (second fig)))
      (EraseOld p q)
      (setq q (ReadFrom "Q1"))
      (DrawNew p q)
      (WriteTo "Q2" (list p q)))
P4:
   (SendThru S4 (ReadFrom "Q2"))
```

Figure 4: The primitives of GMB and an example of their use.

be found in [Vernon, 1983]. For the purpose of this presentation, however, the meaning of a GMB model can be briefly and more informally defined as follows. Each control graph node has associated an input logic expression, in terms of the node's input control arcs, and an output logic expression, in terms of the node's output control arcs. These expressions can involve the operators AND and OR, represented in the figure by the symbols * and +, respectively. For example, the input logic expression for node N4 is Up * Sq, and the output logic expression for node N1 is Dn + Mv + Up. Furthermore, each control graph node is mapped to one data graph processor. In the figure, nodes N1, N2, N3 and N4 are mapped to processors P1, P2, P3 and P4, respectively.

A node's input logic expression defines the required distribution of *tokens*, or units of control, in the node's input control arcs for it to be enabled. For example, node N4 will be enabled when both arcs Up and Sq have one token each at the same time. Enabled nodes are scheduled for activation. When a node becomes active, or fires, the enabling tokens are removed from the corresponding arcs and the interpretation of the processor mapped to the node is executed. Thus, given the situation shown in the figure, if a token is placed on arc Dn, then node N2 becomes enabled; its input logic expression is Dn * Nxt and there is a token initially placed on arc Nxt. When N2 fires, the interpretation of processor P2 is executed: P2 reads the value currently stored in dataset Q1, makes a list containing this value and nil, and writes the value of this list into dataset Q2.

Finally, for an active node, its output logic expression defines which combinations of the node's output control arcs can receive tokens when the execution of the interpretation of the processor mapped to the node is finished. If there is more than one possible combination, because the expression involves the operator OR, then the combination that will actually receive the tokens—one token on each arc in the combination—is specified as part of the processor's interpretation. For example, the interpretation of processor P1 specifies which one of node N1's three output control arcs should receive a token, based on the type of some event.

3 An Operational Model of Multi-Application Multi-User Interfaces

Numerous authors agree on the importance of the design of the underlying user interface model as the first step in the design and implementation of a user interface development system [Green, 1985; Green 1986; Hartson, 1989; Hudson, 1987; Lantz et al., 1987]. The structure of a development system and the services it provides often follow the structure of a model. If such a model is divided into a number of components, the development system can supply design and implementation tools for each of these components. If the model seems logical, designers will have less trouble learning how to use these tools.

The Seeheim model [Green, 1985; Green, 1986] shows how the software implementing the interface is organized at run time and how a design system must be structured to provide this organization. It is based on the concept of dialog independence in which dialog and application are loosely coupled by a control component that defines relationships between them, and transmits tokens back and forth at run-time. Both the Seeheim model and the concept of dialog independence have been useful as frameworks. However, problems have been revealed, as more complex interfaces supporting multiple windows, a direct manipulation style of interaction, and semantic feedback have been investigated and developed. Functional distinctions between user interface and application have been unclear, and it has been found that shared data models are essential.

The Direct-Manipulation model [Hudson, 1987] eliminates the centralized, explicit dialog control component of the Seeheim model by incorporating the multiple, simpler syntactic components into the presentation component (for example, the interaction objects [Jacob, 1986]). It also replaces a simple application model component by a semantically richer shared data model which supports semantic feedback to the user and application notification of data modifications. The Reference model [Lantz et al., 1987; Lantz, 1987] allows the model components to directly communicate between them, considers more explicitly the cases of distributed and collaborative applications and the binding of media-independent I/O to device-dependent I/O, and proposes a hierarchy of dialog managers rooted at a workstation manager component.

We attempt to combine, extend and refine the advantages of the Direct-Manipulation and Reference models in order to provide an operational model of multi-application multi-user interfaces which are useful from the design and implementation points of view. We take special consideration to include, in a way that is at the same time simple and explicit, the aspects of concurrency, collaboration, and reconfigurability.

In the following sections we present a high-level structural view of our operational model. We use the special case of a single-application single-user interface to define the fundamental structural concepts of the model, and then we use the general case of a multi-application multi-user interface to introduce the model's communication concepts. We then use a simple example to illustrate how the model guides the specification of an interface. We present the specification of both the structural and the behavioral aspects of a counter's interface. Next, we provide detailed behavioral models of both the lexical and the syntactic components of our interface model. We describe one generalized lexical component, in charge of handling user inputs and producing system outputs, and nine fundamental syntactic components, in charge of defining legal sequences of user inputs and communicating the successful reception of these sequences to the application. Finally, we present the components and structure of a test environment for multi-user interface specifications.

3.1 The Particular Case of a Single Application and a Single User

In order to present our operational model of a multi-application multi-user interface, we describe first the model as it applies to the special case of one user interacting with one application. Then the case involving multiple applications and multiple users is presented as a generalization of this special case.

We view a single-application single-user interface (UI_1) as consisting of a collection of user interface objects that interact with a user and an application, thus allowing and controlling the interaction between the user and the application. These objects, that we call *screen objects*, essentially represent the various windows on the user's workstation screen, including the various types of interactions that they support. To invoke the functionality of an application a user has to activate these screen objects, by operating input devices, such as mouse and keyboard, usually while pointing at the objects. The application shows the results of executing its functionality by changing the graphical appearance of the screen objects.

Our operational model of a UI_1, whose high-level view is shown in Fig. 5, is based on four types of abstractions: interactors, contacts, dialogs, and an application model (the model is represented in the SM language, described in section 2.5.2). *Interactors* are structured, abstract represention of the screen objects. The *application model* is a structured, object-oriented representation of services or functionality that an application provides to the user through the UI_1. As we will see later, we allow the application model to change according to the number of users sharing the application, while the application itself does not have to change.

Each interactor is an abstraction of things like screen buttons, menus, dialog boxes, scroll bars, or drawing windows and is modeled as consisting of one contact connected to several dialogs. The *contact* models input and output aspects of the interactor. Thus, the full collection of contacts of a UI_1 models the input and output aspects of the whole UI_1. Each one of the *dialogs* connected to the

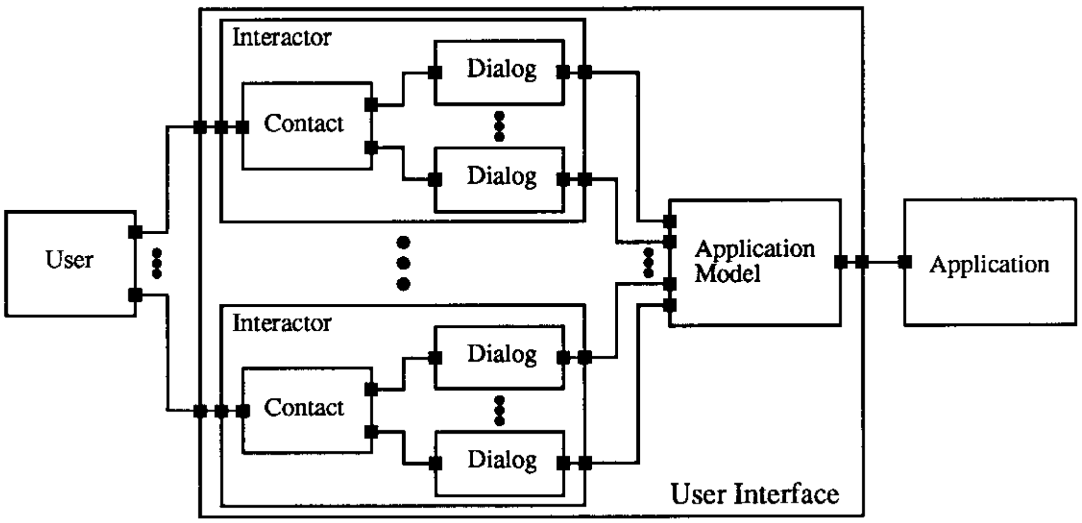


Figure 5: High-level structural model of a single-application single-user interface.

contact models the syntax of a particular type of interaction supported by the interactor. Again, the full collection of dialogs of a UI_1 models the syntax of the whole interaction between a user and an application, represented in our model as the syntax of the interaction between the contacts and the application model. For example, if the interactor represents a drawing window, then the contact may communicate with one dialog that handles rectangles and another that handles polylines.

3.2 The General Case of Multiple Applications and Multiple Users

The high-level view of the operational model of a multi-application multi-user interface for the general case in which K users concurrently share N applications is shown in Fig. 6. In this case, each application is represented within the interface by one application model. However, each application-specific collection of interactors is replicated in agreement with the number of users concurrently sharing the application. Thus, each application model communicates, in general, with K equivalent collections of interactors simultaneously.

Although it is possible to partition the collection of interactors associated with each user according to the applications that the user is executing, we have decided not to do so explicitly because there are situations in which the same interactor can be used by two applications simultaneously. Therefore, the interactors in front of each user simply represent the union of all the interactors needed to execute the N applications. Thus, each user-specific collection of interactors, as a whole, communicates with all the application models, although each individual interactor

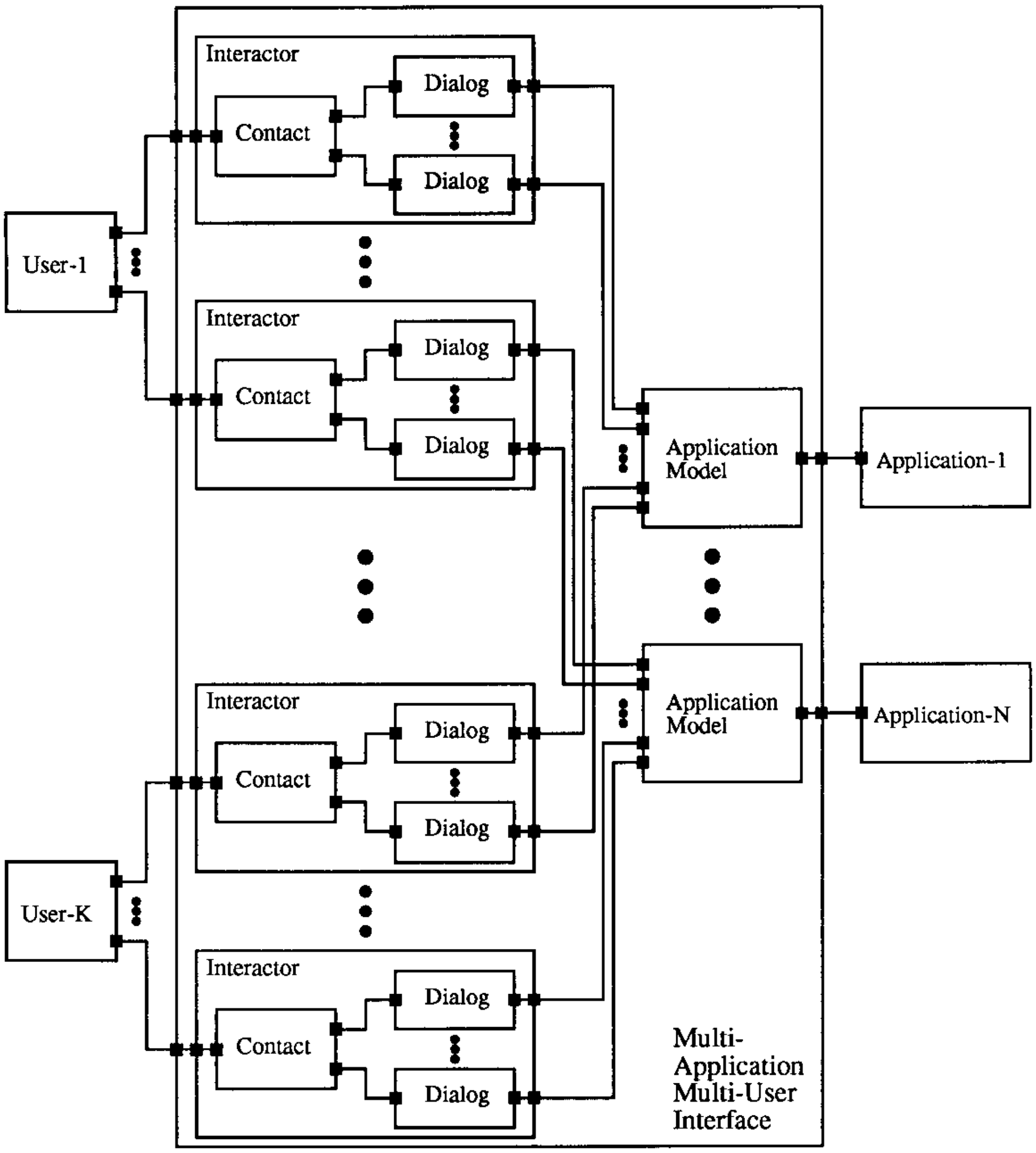


Figure 6: High-level structural model of a multi-application multi-user interface.

within the collection actually communicates with only one application model.

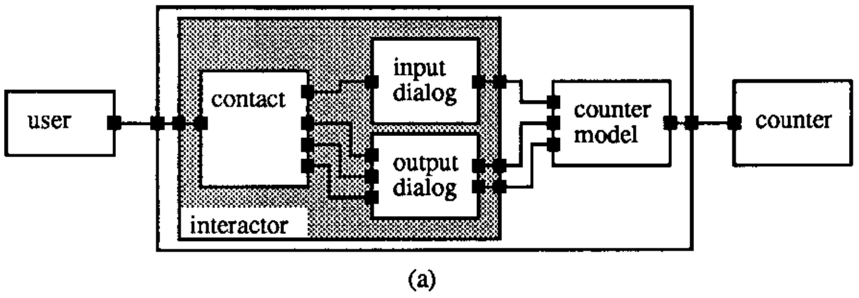
Users produce *actions* on the interactors by operating input devices. Examples of actions are pressing a key on the keyboard and moving or clicking the mouse. Within the interactors, these actions are assumed to be identified by contacts and sent to appropriate dialogs. Each dialog then determines whether a sequence of actions received from a contact is valid. Dialogs translate valid sequences of actions into *commands* which they send to the application models. Examples of commands include creating, modifying, and deleting application data objects.

The application models process the commands received from the interactors and then send *calls* back to the interactors, providing them with information about how to respond to the users. The dialogs in the interactors convert these calls into *responses* which are sent to the contacts, which finally display them to the users. Examples of responses are opening a dialog box, highlighting a screen region, and drawing or erasing a figure. As we will see later on, it is also possible and convenient that a dialog could send responses to a contact while it is still processing the actions that the contact is sending to it, that is, before transmitting the corresponding command to an application model and receiving a call.

3.3 Specifying a Simple User Interface: High-Level Structural Model

Designers graphically specify a multi-application multi-user interface for a collaborative application by connecting contacts to dialogs, thus constructing interactors, and then connecting the interactors to the application model. At the present level of detail, contacts, dialogs and application models are represented in the SM modeling language by *modules*. The points in each module through which information is sent or received are represented by *sockets*. A connection is represented as an undirected arc between a socket in a contact and a socket in a dialog, or between a socket in a dialog and a socket in the application model. Contacts, dialogs and application models usually have several sockets, used to send or receive information of different types.

For example, consider a very simple application which counts the number of times that users click the mouse while pointing the mouse's cursor at a particular screen button. The application displays the current value of the count on the screen button. In this case, as shown in Fig. 7, each user communicates with one interactor, representing the screen button. Within each interactor, the contact communicates with two dialogs: an input dialog which recognizes each click as a valid sequence of actions, and an output dialog through which the application model writes and erases numbers on the screen button. Finally, all the dialogs communicate with the same application model, representing the functionality of the counter.



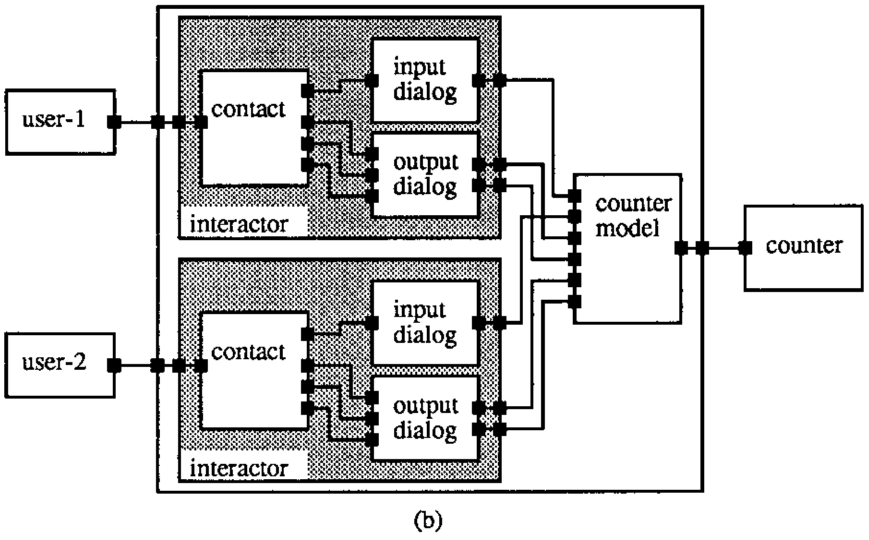


Figure 7: Structural model of the counter's user interface: (a) one user; (b) two users.

The contact in front of each user handles the input and output aspects of the screen button. It is sensitive only to the click action, and is able to display and erase numbers on the screen region corresponding to the screen button. When the contact detects a click produced by a user, it sends a signal to the input dialog connected to it. The syntax of this interaction is very simple, because each click by itself consitutes a valid sequence of actions; therefore, the input dialog simply sends a signal to the counter model every time it receives a signal from the contact.

When the counter model receives a signal from the input dialog, it increments the value of an internal variable by one, and sends to the output dialog the new value of the variable and another signal indicating that the value was sent. Finally, the output dialog sends to the contact: first, one signal requesting the contact to clear the display area of the screen button; then the new value; and then a second signal requesting the contact to display this value.

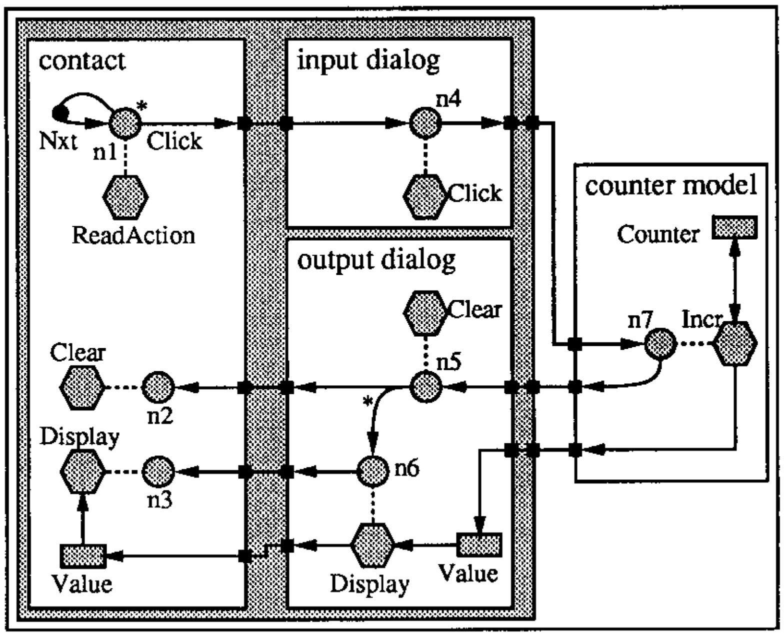


Figure 8: Behavioral model of the counter's user interface for one user.

3.4 Specifying a Simple User Interface: Behavioral Models

The behavior of each contact, dialog, and application model of a multi-application multi-user interface is specified in the GMB modeling language. In general, a behavioral model contains one (node, processor) pair for each basic activity represented in the model, as shown in Fig. 8 for the counter example. For example, a basic activity of the contact in this case is detecting user-generated mouse clicks, and a basic activity of the counter model is incrementing by 1 the value of the counter. We have chosen to abstract the explicit input from a user to module `contact` in Fig. 7(a) by implicitly incorporating the user behavior in the pair (`n1`, `ReadAction`) in module `contact` in Fig. 8; thus, pair (`n1`, `ReadAction`) captures the detection of the clicks. Adding 1 to the counter is captured by the pair (`n7`, `Incr`) in module `counter model`.

At this point it is important to notice that the exact nature of the functionality represented by processor `ReadAction` depends on the stage of the interface development process. In an abstract sense, as we explain in Section 3.5, it simply encapsulates the mechanism by which, somehow, the corresponding screen object receives input from a user. Thus, during the stages of simulation and tests, `ReadAction` *simulates* a user clicking the mouse button by executing an interpretation that produces tokens and places them on the contact's output control arc `Click`. We will describe this capability in more detail and for a more general case in Section 3.8.

On the other hand, during the execution of a prototype of the interface, in which users click real mouse buttons on real screen buttons, the nature of ReadAction depends on the type of the window manager system or toolkit used to implement the real screen buttons. The interpretation of ReadAction could be the actual mechanism, on a per screen object basis, that first detects a click and then produces the corresponding token. Or ReadAction could remain essentially idle, because a different, centralized mechanism used by the window manager to detect the clicks is also used, through a software extension, to produce the tokens. We will describe this approach in Section 4.3b.

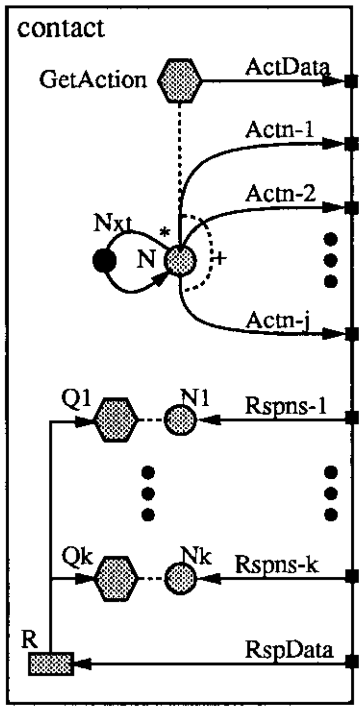
Nodes in the same module or across modules are linked together by directed control arcs according to the partial ordering among the activities. For example, the node associated with the activity of incrementing the counter (n7 in counter model) is preceded by the node associated with the activity of recognizing a complete, valid sequence of actions (n4 in module input dialog); and in module output dialog, the node associated with the activity of requesting the contact to display the new value of the counter (n6) is preceded by the node associated with the activity of requesting the contact to clear the display area (n5).

Processors can be linked to datasets in the same module or across modules by directed data arcs according to the type of access, write or read or both, that a processor has over the data stored in a dataset. For example, in contact, processor Display representing the activity of displaying the value of the counter on the screen has read access over the dataset holding such value (Value); and in counter model, processor Incr representing the activity of incrementing the counter has both read and write accesses over the dataset representing the counter (Counter).

3.5 The GMB Model of a Contact

In an operational model of a multi-application multi-user interface, contacts describe the behavior of the input and output aspects of the interface's screen objects (buttons, menus, dialog boxes, scroll bars, and drawing windows), which are represented in the model by interactors. Therefore, the collection of contacts of an interface model describes the behavior of the input and output aspects of the interface itself. The GMB model of a contact, shown in Fig. 9, consists of an input model and an output model. The input model represents the ability of the modeled screen object to:

1. Detect user-generated input actions; and
2. Communicate the occurrence of these actions to the rest of the interface, in particular to the screen object's components that handle the syntax of



```
    PROCESSORS
GetAction:
    (let ((action (ReceiveAction)))
      (SendThru "ActData" (DataOf action))
      (typecase action
        (Actn-1 (PlaceToken "Actn-1" "Nxt"))
        (Actn-2 (PlaceToken "Actn-2" "Nxt"))
        ...
        (Actn-j (PlaceToken "Actn-j" "Nxt"))))

    (defun ReceiveAction ()
      (loop (if (> (random 10) 6) (return)))
      (let ((action nil)
            (type nil)
            (data (cons (random 100) (random 100))))
        (case (random j)
            (0 (setq type Actn-1))
            (1 (setq type Actn-2))
            ...
            (j-1 (setq type Actn-j)))
        (setq action (make-action type data)
        action))

Q1:
    (let ((data (ReadFrom "R")))
      ...)
...
Qk:
    (let ((data (ReadFrom "R")))
      ...)
```

Figure 9: GMB model of a generic contact. The interpretation depends on the specific contact and is only sketched. The semantics is explained in the text.

the user-application interaction. These components are represented in the interface operational model by the dialogs connected to the contact.

The output model represents the ability of the modeled screen object to produce responses to the users, from requests generated by the application and by the screen object's syntax components. The responses are produced by invoking procedures which usually change the appearance, or graphical state, of the screen object.

In the input model of Fig. 9, node `N` is initially enabled by the token on its only input control arc `Nxt`. Therefore, `N` will eventually fire, removing the token on `Nxt` and activating processor `GetAction`, mapped to `N`. Once `GetAction` is active, it essentially waits until the next action is received. As we explain below, we simulate the generation of actions by executing the function `ReceiveAction`. When invoked, this function loops for a while as determined by some selected distribution, then creates an object by invoking `make-action` and assigns to it random data and type, and finally returns the object.

When an action is received, GetAction sends the action's screen position, the value of (DataOf action), through its output data arc ActData, then places a token on one of the output control arcs, and finally places a token back on Nxt, enabling N again. By sending the action's screen position through ActData, the contact is effectively storing this information in the Pos dataset of one of the dialogs connected to the contact. The specific output control arc on which a token is placed is chosen by GetAction acccording to the type of the action assigned in the make-action object creation noted above. The new token on the output control arc signals that the specific action represented by the arc has occurred, and eventually will be processed by the connected dialog as we will explain in Section 3.6.

At this point we should notice that because a contact is only a model of the input and output aspects of a screen object in general, it does not prescribe any specific way in which actual user input actions are detected, or even any specific way in which these actions are represented to include type and data. We have provided a method for generating events to drive input dialog modules and thereby enable simulation test of interface behavior. The interface designer is free to specify distributions and types of input events useful in such tests. Section 3.8 discusses more generaly how to create and use such test environments during the simulation phase of interface design.

In Section 4.3 we will see that for the purpose of building an executable prototype of such an interface, driven by users operating mouse and keyboard, we provide means to map real screen objects provided by a particular window toolkit to the interactors representing them in the specification. The mapping does then depend on the toolkit and knows about the details of representation and detection of actions, and therefore can effectively produce the data and the tokens specified in the interpretation of GetAction.

The output model of Fig. 9 is essentially always ready to receive data through its input data arc RspData, which is written into the dataset R, and then to receive a token on one of its input control arcs Rspns-1 ... Rspns-k. Both the token and the data are usually generated by one of the dialogs connected to the contact. The token enables the corresponding node N1 ... Nk, which will eventually fire, removing the token and activating the corresponding processor Q1 ... Qk. The processor will then execute its interpretation, which usually takes the form of modifying the graphical appearance of the screen object, and includes reading the data stored in dataset R.

3.6 A Classification of Dialogs

The graphic primitives provided by the GMB modeling language represent low level basic behavior. Based on our experience developing and using the multi-user interfaces of several applications available in our collaborative design environment at UCLA [Mujica, 1991], we have extended the basic GMB notation with a collection of reusable modules that encapsulate the behavior of contacts and dialogs commonly found in these interfaces. The applications include a general purpose drawing editor, specialized drawing editors to produce object-oriented data models and SM and GMB models, and a selection tool and a zoom tool for graphical representations. The number of different dialogs needed to implement the multi-user interfaces of these applications is relatively small, and the syntax of the valid sequences of actions is relatively similar for the different dialogs.

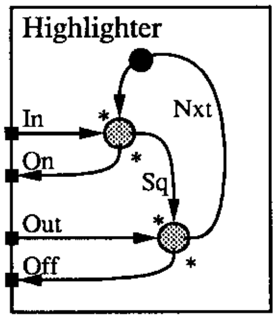
3.6.1 Elementary Dialogs

These dialogs are characterized by the fact that their communication with the contacts and the application models is very simple. Incoming actions and outgoing commands and responses consist exclusively of control signals without any accompanying data, implemented in the control flow models shown in Fig. 10 by the placement of tokens on control arcs. In the following descriptions, we refer to the actions, responses and commands precisely by the name of the corresponding control arc on which the token is placed.

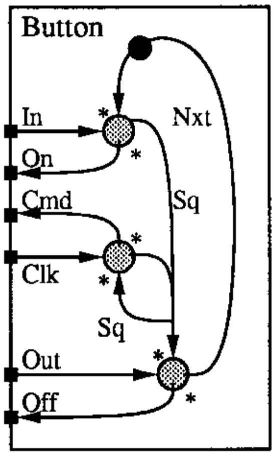
Besides receiving actions and producing and sending responses and commands, the dialogs have a local state that really defines what are the next valid incoming actions. Each dialog's local state is defined by the presence or absence of a single token on either control arc `Nxt` or `Sq`. If the token is on `Nxt` then the only valid incoming action is the first one in the sequence of actions described by the dialog, that is, the action that starts the dialog. If the token is on `Sq` then any of the other actions in the sequence is valid.

- *Highlighter*: Shown in Fig. 10(a), represents a very simple dialog associated with most screen objects. A screen object is highlighted while the mouse points at it; otherwise, it is unhighlighted. The dialog defines a sequence of two actions, `In` and `Out`, which are received from a contact in the form of control signals. The actions always have to occur in alternating order (to be processed successfully), as evidenced by the fact that the dialog switches between its two possible local states after processing each of them. The dialog responds to each action by sending a different control signal, `On` or `Off`, back to the contact.

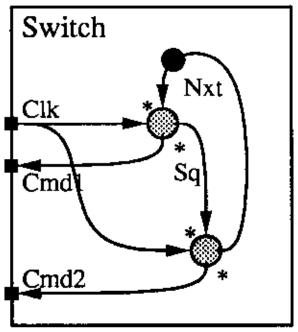
- *Button*: Shown in Fig. 10(b), represents a very simple dialog associated with screen buttons. While pointing at a button with a mouse, a user can click the mouse an arbitrary number of times, always sending the same command to the application. The dialog defines a sequence of three actions, `In`, `Clk` and `Out`, which are received from a contact in the form of control signals. `In` and `Out` actually correspond to the actions of a highlighter. Because



(a) Highlighter.
Defines a sequence of 2 actions received from a contact as tokens placed on arcs In (first action) or Out (second action). Responds to the In action by placing a token on arc On, and to the Out action by placing a token on arc Off, both arcs connected to the contact. The 2 actions always have to occur in alternating order.



(b) Button
Defines a sequence of 3 actions received from a contact as tokens placed on arcs In (first action), Clk (second action) or Out (third action). Responds to the In action by placing a token on arc On, and to the Out action by placing a token on arc Off, both arcs connected to the contact. Responds to the Clk action by placing a token on arc Cmd, connected to the application model. The In and Out actions can occur only once each; the Clk action can occur an arbitrary number of times.



(c) Switch.
Defines an elementary sequence of 1 action received from a contact as a token placed on arc Clk. Responds to this action alternatively, placing a token on arc Cmd1 or arc Cmd2.

Figure 10: Behavioral models of: (a) Highlighter; (b) Button; (c) Switch.

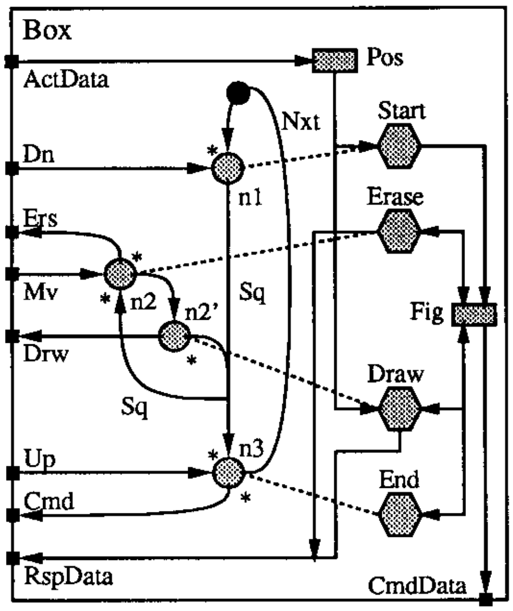
highlighters are always useful with buttons, we included them directly in this dialog. They represent the mouse beginning and finishing to point at a button and therefore can occur only once each during the same execution of the dialog. The dialog responds to each of them by sending a different control signal, `On` or `Off`, back to the contact. `Clk` represents clicking the mouse while pointing at the button and therefore can occur an arbitrary number of times. Everytime, the dialog responds by sending the control signal `Cmd` to the application model. The number of valid occurrences of the different actions is controlled by the way in which the dialog changes its local state, placing tokens on either arc `Sq` or `Nxt`, after processing each action.

- *Switch*: Shown in Fig. 10(c), represents a very simple dialog associated with a screen button that sends to the application model one of two different commands alternatingly, every time a mouse is clicked while pointing at the button. The dialog defines an elementary sequence of one action, `Clk`, received from a contact in the form of a control signal. The dialog responds to this action by sending to the application model one of two control signals, `Cmd1` or `Cmd2`, depending on its current local state. The fact that these two signals are always produced in alternating order is guaranteed by the way in which the dialog switches back and forth between its two possible local states after processing each `Clk`.

3.6.2 Drawing Dialogs

These dialogs are characterized by the fact that they define new structured data for the application models, based on information received from the contacts. Their communication with the contacts and the application models, therefore, involves not only control signals implemented as tokens placed on control arcs, but also data sent or received through input and output data arcs and stored in datasets. Furthermore, the dialogs are able to collect individual pieces of data received from the contacts, and structure them in an appropriate way, usually in the form of a list, before sending a command to the application model. During the collection of the data, the dialogs send several useful responses to the contacts with the purpose of providing feedback to the users.

- *Box*: Represents a very common dialog needed to input rectangles and to create rectangle-based figures. The user marks the position of the rectangle's top lefthand corner by pressing down the mouse's button, then drags the mouse towards the position of the rectangle's bottom righthand corner producing a rubber band effect, and finally releases the mouse's button. The dialog, shown in Fig. 11, defines a sequence of three input actions, `Dn`, `Mv` and `Up`. These actions are received from a contact in the form of a screen



DATASETS

Pos: (x.y)
Fig: ((x.y).(u.v))

PROCESSORS

```
Start:
  (let ((P (ReadFrom "Pos")))
     (WriteTo "Fig" (cons P nil)))
Erase:
  (let ((R (ReadFrom "Fig")))
     (SendThru "RspData" R))
Draw:
  (let ((Q (ReadFrom "Pos"))
        (R (ReadFrom "Fig")))
     (setf (cdr R) Q)
     (WriteTo "Fig" R)
     (SendThru "RspData" R))
End:
  ()
```

SEMANTICS

In the initial state, all processors are idle, control arc Nxt has exactly one token on it, all other control arcs have no tokens, and datasets Pos and Fig are empty.

The dialog begins when a screen position is written into dataset Pos, through arc ActData, and a token is placed on arc Dn. Node n1 is enabled and eventually fires, removing the tokens on Nxt and Dn and activating processor Start. Start reads the value in Pos and writes it into dataset Fig. When Start becomes idle again, a token is placed on arc Sq. The dialog now is ready to continue, if a token is received on arc Mv, or to finish, if a token is received on arc Up.

The dialog continues if a screen position is written into Pos and a token is placed on Mv. Node n2 is enabled and eventually fires, removing the tokens on Sq and Mv and activating processor Erase. Erase reads the coordinates of the current rectangle in Fig and sends them through arc RspData. Then it becomes idle again, a token is placed on arc Ers, and node n2' is enabled.

Node n2' eventually fires, activating processor Draw. Draw reads the new value in Pos, reads the coordinates of the current rectangle in Fig, updates these coordinates with the value from Pos, writes the new coordinates back into Fig, and sends them through RspData. When Draw becomes idle again, tokens are placed on arcs Drw and Sq, allowing the dialog to repeat this sequence of activities, if a token is received again on Mv, or to finish, if a token is received on Up.

The dialog finishes if a token is placed on Up. Node n3 is enabled and eventually fires, removing the tokens on Up and Sq, and placing tokens on arcs Cmd and Nxt and thus returning to its initial state.

Figure 11: Behavioral model of the Box dialog.

position written into dataset Pos plus a token placed on the appropriate input control arc. While Dn and Up can occur only once each during the same execution of the dialog (at the beginning and at the end, respectively), Mv can occur an arbitrary number of times, after Dn and before Up.

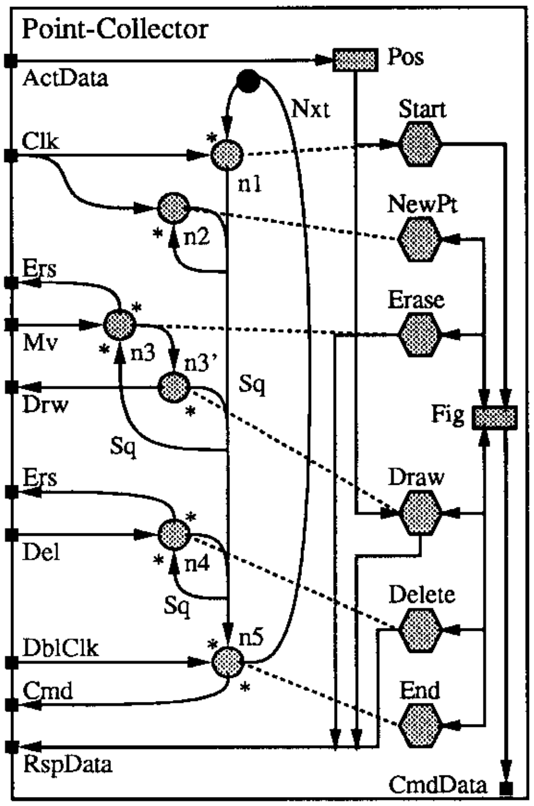
- *Point-collector*: Represents a very common dialog needed to input sequences of screen positions and to create polylines and other types of figures based on such sequences. The user marks each position by clicking the mouse's button. While the mouse is being moved from one position to the next, the dialog produces a rubber band effect for a line segment. By typing "d" on the keyboard at any time, the current last point of the sequence is deleted. The dialog, shown in Fig. 12, defines a sequence of four actions, Clk, Mv, Del and DblClk. These actions are received from a contact in the form of a screen position written into dataset Pos plus a token placed on the appropriate input control arc. Del does not include a screen position. While DblClk can occur only once during the same execution of the dialog (at the end), all the other actions can occur an arbitrary number of times. The first occurrence of Clk, which starts the dialog marking the first position, is interpreted by processor Start; subsequent occurrences of Clk, which mark all the other positions, are interpreted by processor NewPt.

We said that the dialogs receive the individual pieces of data associated with each input action as a value written into the Pos datasets. They construct the structured data that is being defined by the user through these input actions, according to the interpretation associated with each processor, shown in complete detail in the figures. During the construction process the data is stored and updated in the Fig datasets, which by the time the dialogs are finished contain the complete structures. At this time the dialogs send the corresponding command to the application model by placing a token on the Cmd output control arcs. The application model can read the value of the structured data associated with the command through the CmdData data arcs.

The concept of a local state described for the elementary dialogs also applies to these dialogs: if there is a token on arc Nxt the dialog is ready to begin; if there is a token on arc Sq the dialog is ready to continue or to finish. When the dialog begins, it removes the token from Nxt and eventually places a token on Sq. If the dialog continues, it removes the token from Sq and eventually replaces it. If the dialog finishes, it removes the token from Sq and replaces the token on Nxt, returning to its initial state.

3.6.3 Editing Dialogs

These dialogs are characterized by the facts that: (1) they can modify data that already exist in the application models, according to information received from



DATASETS

```
Pos: (x.y)
Fig: List of (x.y)
```

PROCESSORS

```
Start:
  (let ((P (ReadFrom "Pos")))
    (WriteTo "Fig" (list nil P)))
NewPt:
  (let ((L (ReadFrom "Fig")))
    (setf (cadr L) (car L)
          (car L) nil)
    (WriteTo "Fig" L))
Erase:
  (let ((L (ReadFrom "Fig")))
    (SendThru "RspData"
      (cons (cadr L) (car L))))
Draw:
  (let ((L (ReadFrom "Fig"))
        (Q (ReadFrom "Pos")))
    (setf (car L) Q)
    (WriteTo "Fig" L)
    (SendThru "RspData"
      (cons (cadr L) Q)))
Delete:
  (let* ((L (ReadFrom "Fig"))
         (S (cons (2nd (cdr L)) (1st (cdr L)))))
    (setf L (cons nil (cddr L)))
    (WriteTo "Fig" L)
    (SendThru "RspData" S))
End:
  (let ((L (ReadFom "Fig")))
    (setf L (cdr L))
    (WriteTo "Fig" L))
```

SEMANTICS

Similar to that of the box dialog for several input actions. Reception of the first token on arc Clk is analogous to reception of a token on arc Dn; reception of a token on arc DblClk is analogous to reception of a token on arc Up; and reception of a token on arc Mv is analogous to reception of a token on arc Mv in the box dialog.

This dialog, however, can continue not only by receiving a token on Mv, but also by receiving a new token on arc Clk or by receiving a token on arc Del. If it receives a new token on Clk while there is a token on arc Sq, node n2 becomes enabled and is eventually fired, removing the enabling tokens and activating processor NewPt. NewPt updates the sequence of coordinates in dataset Fig, including now the last screen position written into dataset Pos.

If the dialog receives a token on Del, node n4 is enabled and eventually fires, activating processor Delete. Deletes updates the sequence of coordinates in Fig by removing the last one entered, then sends this value and the new last coordinate in Fig through dataset RspData.

Figure 12: Behavioral model of the Point-Collector dialog.

the contacts, (2) the modification dialog reads data from the selection dialog, and (2) the results of the modifications are reflected on the screen objects, according to information received back from the application models.

- *Point-Selector*: Selection of graphical objects based on a single screen position, usually defined by clicking or double clicking the mouse while pointing at the object of interest, very similar to the Button dialog. Once the screen position is defined, the dialog: communicates with the application model to read the coordinates of all the figures already stored in the application model's dataset; then decides which figure has been selected, stores it in the dialog's own dataset, and highlights its representation on the screen object.

- *Box-Selector*: Selection of graphical objects by drawing a rectangle that encloses them; the rectangle is drawn using the same press-drag-release sequence described in the case of the Box dialog. Once the rectangle is defined, the dialog: communicates with the application model to read the coordinates of all the figures already stored in the application model's dataset; then decides which figures have been selected, stores them in the dialog's own dataset, and highlights their representations on the screen object. Thus, what the Point-Selector dialog does for one figure, the Box-Selector dialog does for a collection of figures.

- *Move-Shape*: This dialog changes the position, size or shape of a selected figure or collection of figures. In practice, there are three different dialogs, one for each type of change. Their control and data graphs are the same, and in fact are equivalent to those of the Box dialog, but they differ in the interpretations associated with the data graphs. The user presses the mouse's button while pointing at some "handle" in the figure. A figure's handles are shown when the figure is highlighted, as a consequence of selecting it. Then the user drags the mouse until the new desired position, size or shape is achieved, at which point he or she releases the button. Finally, the dialog communicates the new coordinates of the figure back to the application model.

- *Text-collector*: To enter a small, possible multiline, string of text by typing it on the keyboard.

3.6.4 Comments

Interaction Objects [Jacob, 1986], which are described briefly in section 2.2.1, was one of the first dialog specification languages to recognize the fact that a user interface based on the direct-manipulation style of interaction is composed, from a syntactic point of view, of a collection of many, relatively simple individual dialogs. However, Jacob's language does not provide a basic collection of dialogs as primitive constructions. Garnet's Interactors [Myers et al., 1990; Myers, 1989a], which are described briefly in section 2.2.4, on the other hand, reflect the fact that not only these interfaces consist of a collection of individual dialogs, but also that there are few really distinct dialogs. Therefore, Garnet provides only six types of interactors, representing encapsulated dialog behavior, as primitive constructions to specify user-application dialogs.

Our own encapsulated dialogs are very similar to Garnet's interactors in terms of the high-level functionality supported. The main differences are that while the interactors include menus and angle detectors explicitly, our dialogs include switches and selectors. A more important difference, though, is the fact that our dialogs support in a natural way communication of control and data, not only with screen objects and applications, but also between dialogs. This property allows designers to:

1. Specify feedback to the users without involving the applications beyond what is strictly necessary, as we do for the case of the Move-Shape dialog that communicates with a selector dialog.

2. Explicitly specify sequencing and mutual exclusion among dialogs, required to provide application-independent coordination in collaborative environments, as we show in Chapter 5.

3.7 The GMB Model of a Generalized Dialog

A dialog models the syntax of a specific type of interaction between a user and an application, carried out through a screen object. It defines the structure of a valid sequence of actions produced by the user, and received by the dialog from a contact connected to it, including the points in the sequence at which: commands are sent to the application, calls are received from the application, and responses are sent to the user through the contact. From the dialog point of view, the application is represented by an application model. In general, upon the succesful completion of such a sequence of actions, commands, calls and responses, the dialog:

1. Informs the application model that an instance of the interaction represented by the dialog has occurred; and

2. Makes available to the application model all the corresponding relevant data.

The detailed behavior of a number of dialogs commonly found in graphical user interfaces based on the direct-manipulation style of interaction has been presented in Section 3.6. The similarities among the control graphs and among the data graphs of these dialogs, have led us to define a generalized dialog. The GMB model of the generalized dialog is shown in Fig. 13. It describes a sequence of incoming actions and outgoing responses and commands, organized in 3 major parts: initiation, continuation, and termination. Respectively, they specify what happens—in terms of invoking an application model's functionality, accessing an application model's data, and producing responses to a user—when the dialog begins (involving nodes `n1` and `n1'` and the corresponding processors, control arcs and data arcs), while it is in progress (involving nodes `n2` and `n2'`), and when it finishes (involving node `n3`).

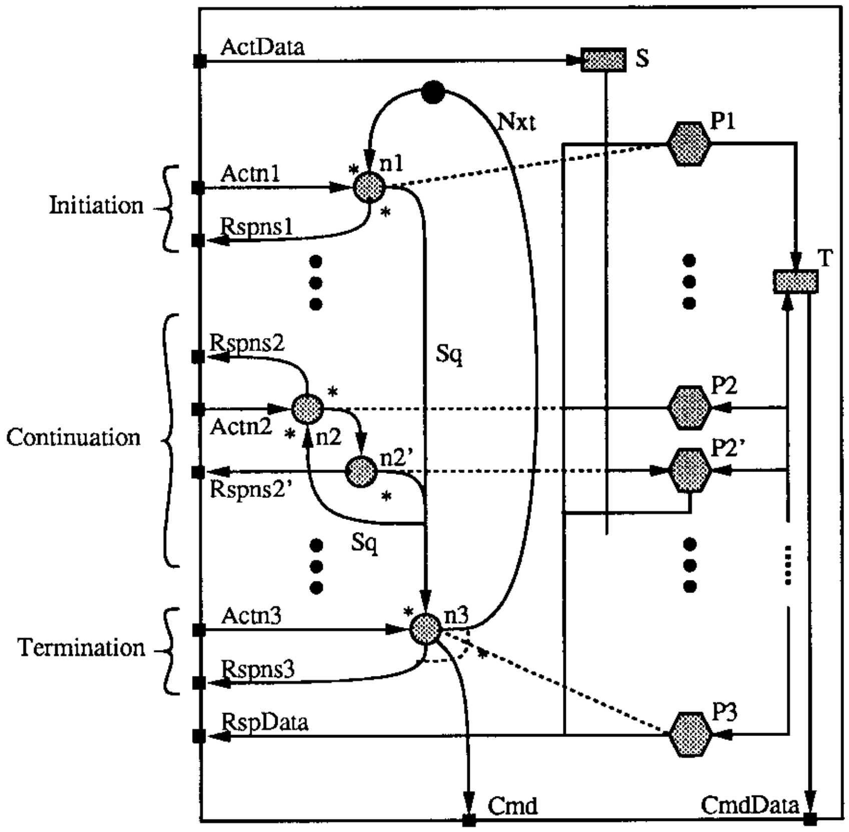


Figure 13: GMB model of a generalized dialog. The interpretation depends completely on the actual dialog instance. The semantics is explained in the text.

The activities of entering and leaving a dialog, that is, initiation and termination, respectively, take place once each during the same execution of the dialog. Each activity is started by placing a token on a specific input control arc (i.e., arc `Actn1` for initiation, arc `Actn3` for termination) when a specific user-generated action occurs. The tokens are produced by the contact connected to the dialog. For example, if the dialog to produce a polyline on a drawing window begins with a single click of the mouse and finishes with a double click, the contact would place a token on `Actn1` when the first click occurs and would place a token on `Actn3` when the double click occurs.

While the dialog is in progress, that is, during the continuation activity, tokens can be placed on any one of several input control arcs in the central step of the dialog (arc `Actn2` is one of them, the others are captured in the ellipses). This represents the fact that several different actions can occur in any order and be repeated an arbitrary number of times, after beginning and before finishing the dialog. For example, while drawing a polyline, after the first click and before the final double click, a user has three options: (1) move the mouse to produce a rubber band line segment between the current last point of the polyline and the current position of the mouse's cursor; (2) press the key "d" on the keyboard to delete the current last segment and last point of the polyline; or (3) click the mouse again to define a new point and a new line segment of the polyline. These three actions can be repeated an arbitrary number of times and in any order while the polyline is being drawn.

3.8 A Test Environment for Multi-User Interfaces

An important advantage of using the SM and GMB languages to model a multi-user interface is that it is possible to construct a *test environment* for the model using the SM and GMB languages themselves [Razouk et al., 1979], and thus to test the model by executing the same language interpreter (see section 2.5.2) on both the test environment and the model simultaneously. That is, before building an actual interface, or even an executable prototype of it, in which users interact with real screen objects through real input devices, it is possible to build SM modules whose behavior, specified in terms of the GMB, can be used to simulate the generation of input actions (which in reality are a consequence of the interaction between users and screen objects) and output responses (which in reality are generated by the screen objects when they receive requests from the applications). The objective of these modules is to allow the designers of a multi-user interface to test their specifications in a systematic and controlled way.

In particular, we propose the construction of a simple, modular and hierarchical test environment for multi-user interface specifications based on the operational model, as follows. First, within each interactor, the corresponding collection of dialogs is tested by the interactor's contact. Then, each collection of interactors representing the screen objects of the user interface of an individual user is tested by a module representing the user and controlling the activation of the contacts. Finally, a complete multi-user interface, in which equivalent interactors belonging to different users communicate among themselves, is tested by a special module representing the collection of users and controlling the activation of the user modules. The structure of our test environment, for the specific case of two users concurrently sharing an application through an interface consisting of two interactors, each containing two dialogs, is shown in Fig. 14.

Besides defining the input and output behavior of screen objects, contacts have

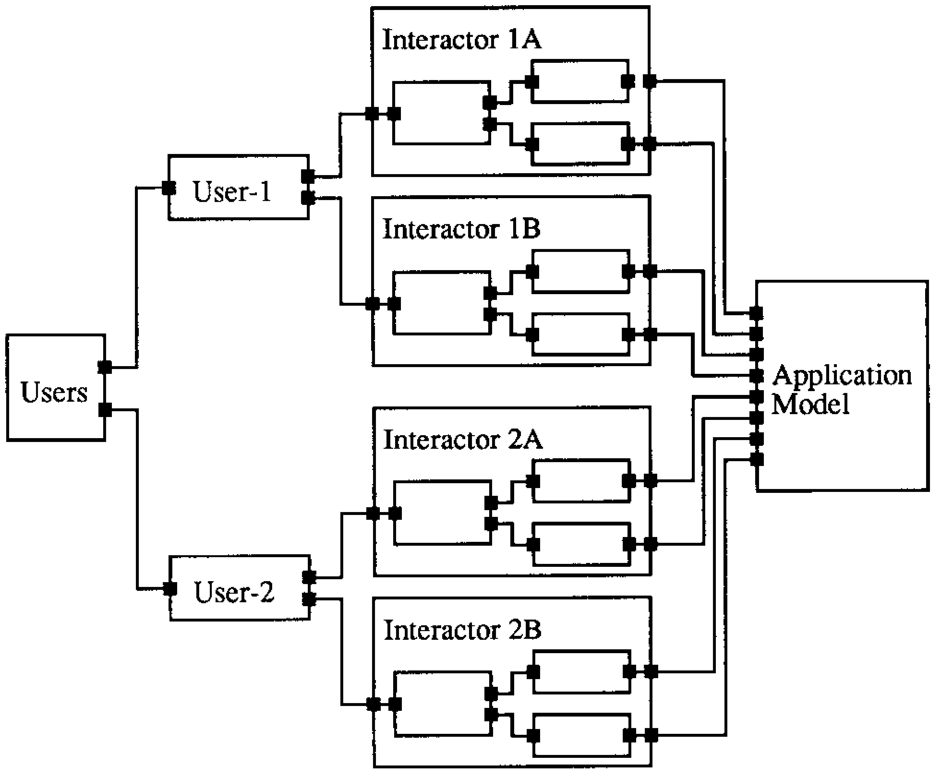
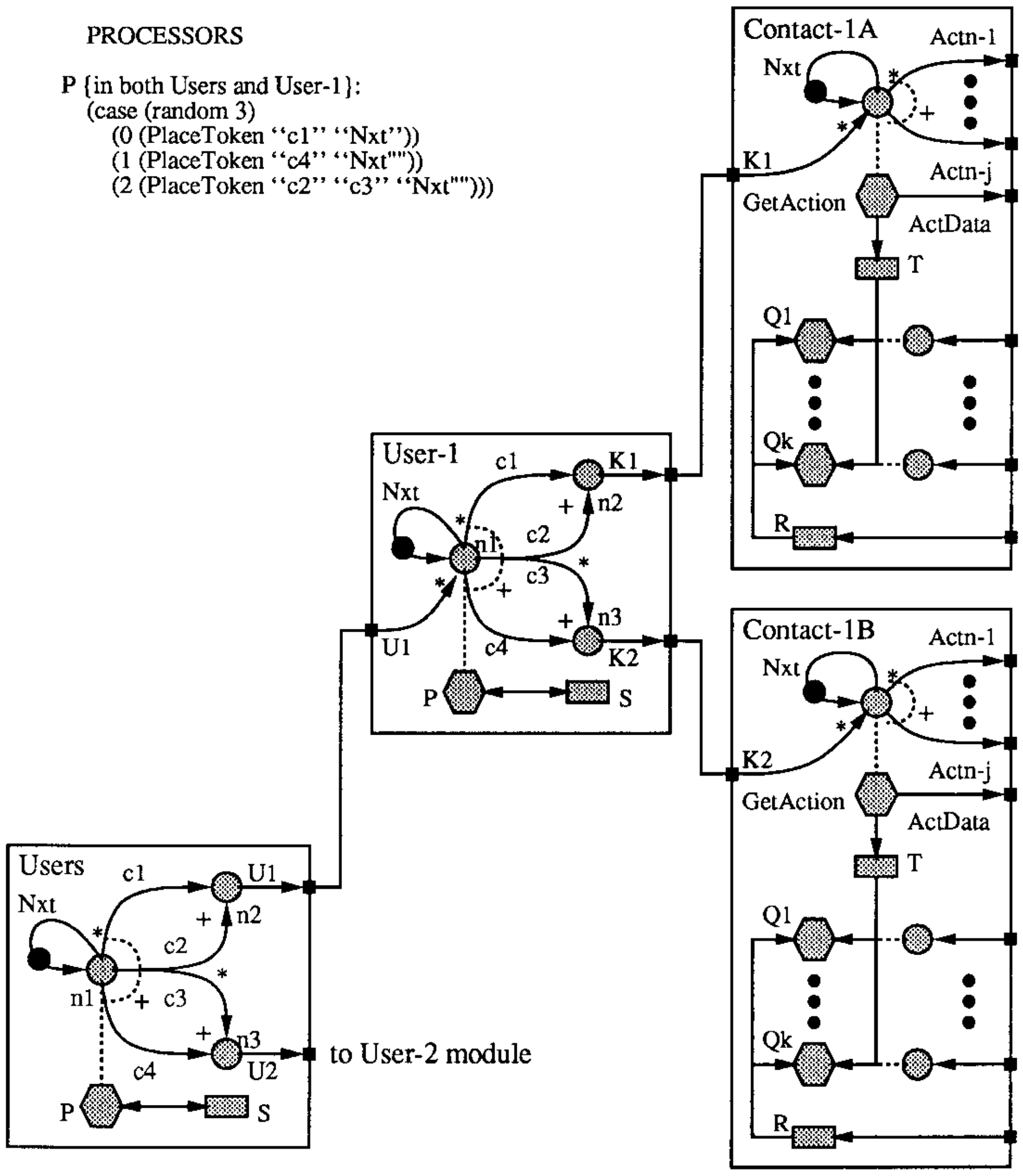


Figure 14: Structural model of the test environment.

a very simple control logic in which all control arcs: (1) are simple, that is, they have only one source and only one destination, and (2) connect a socket to a node or a node to a socket, so that no node has to wait for a token produced by another node in the same contact to become enabled. Thus, contacts represent the most appropriate components of our multi-user interface operational model which can also be used as building blocks of a test environment for an interface specification.

When a contact like the one shown in Fig. 9 participates in a test environment, the occurrence of input actions is simulated by processor `GetAction`, the occurrence of output responses is simulated by processors `Q1` ... `Qk`, and a new dataset `T` is used to check whether the data associated with the output responses corresponds to the data associated with the input actions. The new contacts `Contact-1A` and `Contact-1B`, user module `User-1`, and module `Users`, representing the interaction among the multiple users, are shown in Fig. 15.

The interpretation of `GetAction` writes data through the contact's output data arc (`ActData`) and into the new dataset `T`, and places tokens on the different output control arcs of the contact (`Actn-1` ... `Actn-j`). The tokens are placed on the arcs in the order in which they would be produced if users were interacting with screen objects. Therefore, tokens are sent to each dialog connected to the contact in the correct sequence required by that dialog, although the different



```
PROCESSORS

P {in both Users and User-1}:
   (case (random 3)
      (0 (PlaceToken "c1" "Nxt"))
      (1 (PlaceToken "c4" "Nxt""))
      (2 (PlaceToken "c2" "c3" "Nxt"")))
```

Figure 15: Behavioral model of the test environment.

dialogs can be activated in random order.

The interpretation of processors `Q1` ...`Qk` can have access to real screen objects with graphical capabilities, or to a simple message window. In the first case, a processor can send to a window a response of the form `(draw-line (10 20) (80 95))`, which will be interpreted by the window so that users will be able to "see" the response. In the second case, a processor can simply print out information regarding its activation, for example, can print the message `Drawing line from (10 20) to (80 95)`. In either case, before producing a response, processors compare the data stored by the dialogs in dataset `R` with the data stored by `GetAction` in the new dataset `T`. If these two pieces of information coincide, the processors proceed as described. Otherwise, they can take different actions depending on how users want to handle errors in the interface specification.

To test the operation of multiple interactors representing the screen objects in the interface of an individual user, we introduce modules `User-1` and `User-2`, which controls the activation of the contacts in the interactors. The behavioral model of `User-1` is shown in Fig. 15. Essentially, the input logic expression of node `N` in each contact depends now not only on control arc `Nxt`, but also on a control arc K_i coming from `User-1`. Because the interpretation of processor `P` in `User-1` decides which of the multiple K_i arcs will receive a token during a particular activation of `P`, `User-1` also decides which of the interactors become active at that time, and thus simulates a user that operates simultaneously on a number of screen objects. Specifically, if a token is placed on control arc `c1`, then only `Contact-1A` becomes active; if a token is placed on arc `c4`, then only `Contact-1B` becomes active; and if tokens are placed on both arcs `c2` and `c3`, then both `Contact-1A` and `Contact-1B` become active.

Finally, to test the operation of a complete multi-user interface, we introduce module `Users`, which controls the activation of modules `User-1` and `User-2`, essentially in the same way in which these modules control the activation of the contacts connected to them. The behavioral model of `Users` is shown in Fig. 15.

3.9 Comparison with Related Works

Our model is the first model to deal explicitly, at the level of control and data flows, with some of the complexities involved when a user interface controls and allows interaction between a group of collaborating users and a collection of applications and the applications' data, which are being concurrently shared by the users. Our model is a combination, an extension and a refinement of the advantages of the Direct-Manipulation model [Hudson, 1987] and the Reference model [Lantz et al., 1987; Lantz, 1987] of user interfaces. It follows the trend of these other models with respect to the decentralization of dialog syntax management, providing a precise framework for this management to be handled by the individual screen objects composing the user-visible end of the interface. Our model refines the direct-manipulation model because it represents explicitly the way in which a screen object actually deals with both the input/output and the dialog syntax management aspects of an interface. Our model extends the Direct-Manipulation model in the direction of the Reference model, because it deals explicitly with multiple users and multiple applications, through the idea of replication of the input/output and dialog syntax management aspects according to the number of users, and through the concept of a single application model for each application.

4 An Object-Oriented Approach to Prototyping Multi-User Interfaces for Collaborative Applications

In general, building multi-user interfaces is a fairly complex and expensive process. As we said in Chapter 1, we require that these interfaces be graphical, interactive, collaborative, based on the direct-manipulation style of interaction, and dynamically reconfigurable. Formal graphical descriptions, appropriate abstractions, and the ability to build executable prototypes, as has been suggested by numerous authors [Goguen and Moriconi, 1987; Harel et al., 1990; Foley et al., 1989; Linton et al., 1989; Lewis et al., 1989; Singh and Green, 1991; Wellner, 1989], are three approaches whose advantages can be combined to help in this process.

In this chapter we describe an object-oriented methodology to build executable prototypes of multi-user interfaces from the graphical, formal specifications of their single-user versions. Numerous authors agree on the advantages of using an object-oriented approach for the design and implementation of user interfaces [Barth, 1986; Fisher, 1987; Hartson, 1989; Jacob, 1986; Linton et al., 1989; Kimbrough, 1989]. Our design and development method is based on two basic ideas about multi-user interfaces:

- Our operational model of a multi-user interface consisting of two major types of components is shown in Fig. 16: (1) a collection of *interactors*, representing screen objects such as buttons, menus, dialog boxes, scroll bars, and drawing windows; and (2) an *application model*, representing the services provided by the application to the users through the interface. Each interactor in turn consists of one *contact*, modeling its input and output aspects, connected to a collection of *dialogs*, defining the syntax of the various types of user/application interactions supported by the interactor.

- An object-oriented approach to the specification and construction of multi-user interfaces, evolved from the coSARA tool model [Mujica, 1991], in which an interface and its components are seen as objects belonging to specific classes, participating in several relations, and communicating through different types of messages. Graphical formal languages are used for specifying the structure, relationships, behavior and communication schemes of the classes. An appropriate tool is used to link the actual objects (class instances) to their behavioral models for execution.

The design and development method works in four major steps. First, designers build a class-oriented model of the multi-user interface, which we call the *class diagram*. Then, they build behavioral models for their application model, for the contact and for the dialogs contained in each interactor. Next, designers

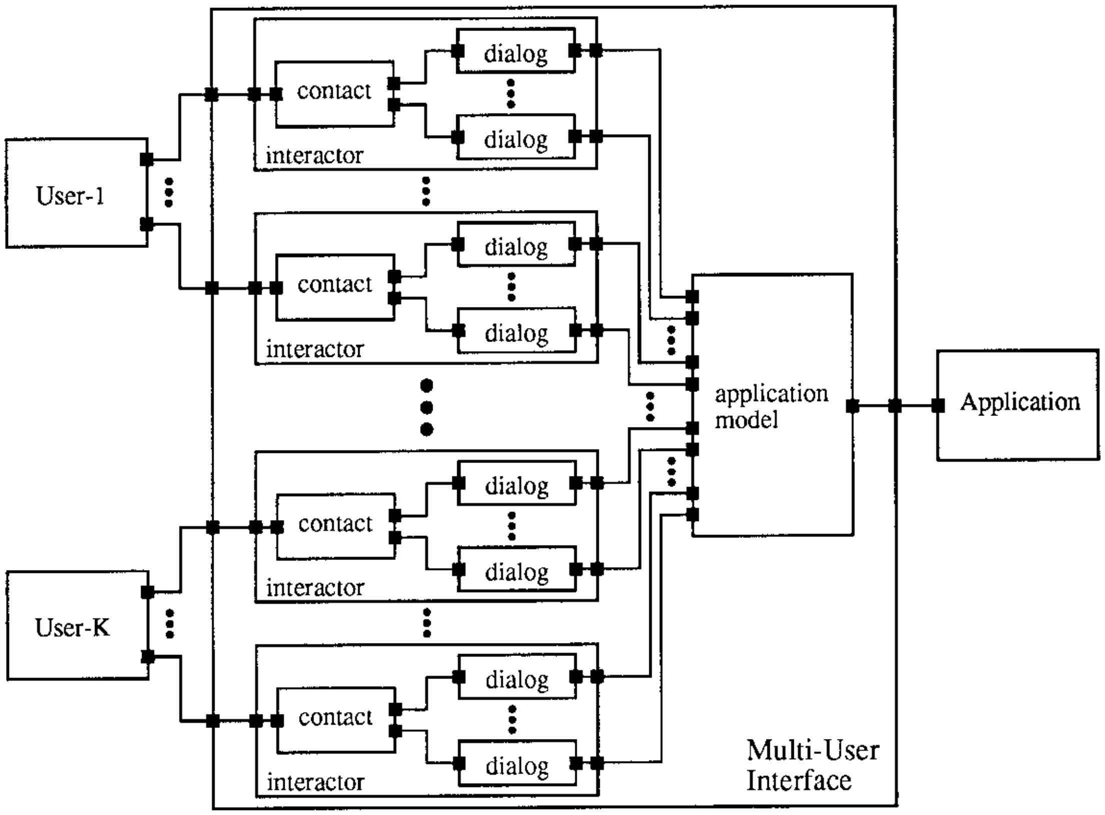


Figure 16: High-level structural model of a multi-user interface between K users and one application.

create instances of the application model and of the interactors. Next, they link actual screen objects to instances of the interactors, thus building an executable prototype of the interface. Finally, they test the behavior of the prototype and make modifications, repeating the process until the prototype interface satisfies their intent.

We illustrate the method by applying it to the design of the multi-user interface of a simple drawing tool, which allows multiple users to concurrently draw and move rectangular blocks on a window. Fig. 17 shows a sample window during use of the tool. Operationally, a block is drawn by pressing down the mouse's left button at the position of the rectangle's top lefthand corner, and then releasing the button after dragging the mouse to the desired position of the rectangle's bottom righthand corner; a block is moved by pressing down the mouse's right button while pointing the mouse's cursor to a block's edge, then releasing the button after dragging the block with the mouse to a new location.

In the following sections we present the class diagram of the drawing tool's multi-user interface, built in terms of the OREL modeling language; and the behavioral models for each of the classes represented in the class diagram, described

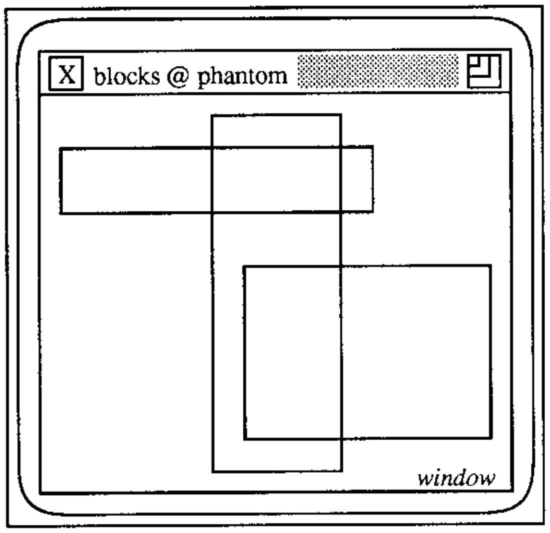


Figure 17: The drawing tool during operation.

in terms of the SM and GMB modeling languages. Then, we describe the processes by which these models are converted into executable prototypes, through class instantiation and association of class behavior with the corresponding instances; and we illustrate the execution of the final prototype through an example consisting on drawing a block on the drawing window.

4.1 The Class Diagram of the Multi-User Interface

The first step of our design method is to construct the class diagram of the multi-user interface. The class diagram of an interface is a description of the collection of classes (their structure and relationships) needed to support the operation of the interface. OREL, a language for object-oriented modeling [Mujica, 1991], which incorporates relations and is supported by the OREL graphical editor, provides six primitives to describe class diagrams: (1) simple classes; (2) composite classes; (3) recursive composite classes; (4) class attributes; (5) relations among classes; and (6) class inheritance. The meaning and graphical representation of these primitives are given in section 2.5.1.

Fig. 18 shows the OREL class diagram for the drawing tool's multi-user interface. The user interface is represented as a top-level composite class, `UI`, containing four component classes and two component relations. Two of the component classes are `Tool`, which represents the application model component of our operational model, and `Interactor`, which represents all the interactors, or screen objects, through which each user executes the application and observes the effects of this execution.

According to the operational model, the application model (`Tool` in this case) and the interactors communicate via commands and calls, which explains the

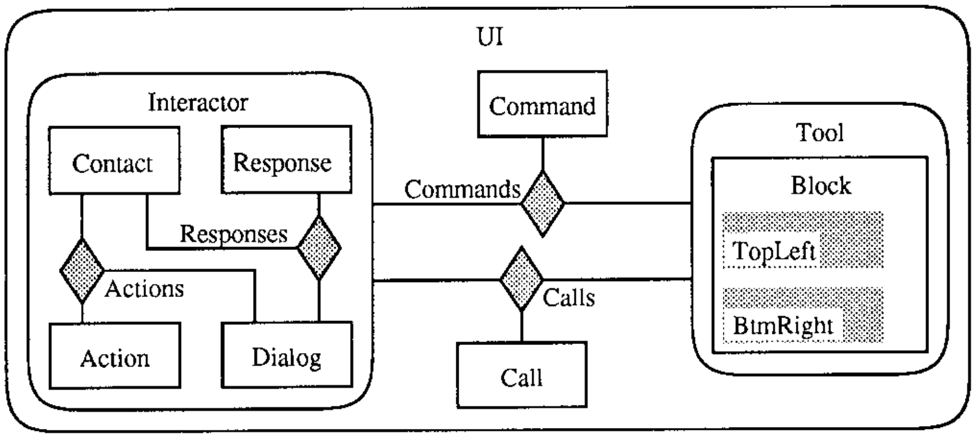


Figure 18: The OREL class diagram of the drawing tool's user interface.

other components of the class `UI`. Classes `Command` and `Call` represent all the different types of messages that can be exchanged between the interactors and the application model. Relations `Commands` and `Calls` define precisely which commands and which calls are sent or received by each interactor. For example, each tuple of `Commands` contains three objects, of classes `Command`, `Interactor` and `Tool`, respectively.

`Tool` is a composite class whose only component class, `Block`, represents the class of objects handled by the drawing tool. The objects' properties, that is, the positions of the top lefthand and bottom righthand corners of the blocks, are the attribute slots of this simple component class: `TopLeft` and `BtmRight`. An instance of `Tool` contains a variable number of instances of `Block`, which changes as users draw more blocks. Each instance of `Block`, that is, each block drawn by the users, contains its own values for the attributes `TopLeft` and `BtmRight`.

`Interactor` is also a composite class. It contains four component classes and two component relations. Together, these components represent the following facts:

1. Each interactor contains one contact, represented by the simple class `Contact`, and several dialogs, all of them represented by the simple class `Dialog`; and

2. Within an interactor, the contact communicates with the dialogs via actions and responses.

Simple classes `Action` and `Response` represent the messages that the contact can exchange with the dialogs, while relations `Actions` and `Responses` define which actions and responses correspond to each dialog. Thus, for example, in a particular interactor each tuple in the relation `Responses` consists of three objects: the only contact of the interactor, one of the several dialogs of the interactor, and one of the responses that the dialog can send to the contact. `Responses` contains as many tuples as the total number of different types of responses that can be generated by all the dialogs in the interactor.

A tool called the *OREL compiler* translates an OREL class diagram, such as the one shown in Fig. 18, into (the actual CLOS code is presented in Appendix A):

- the appropriate class definitions, e.g., for classes `Block`, `Tool` and `UI`, and for the relation `Calls`;
- the functions to instantiate the classes, e.g., functions `make-Block`, `make-Tool`, `make-UI` and `make-Calls`;
- the methods to assign values to, and read the values of, the objects' attribute slots, e.g., for slots `TopLeft` and `BtmRight` in objects of the class `Block`;
- the methods to add or remove the component objects of a composite object, e.g., to add `Block` objects to a `Tool` object;
- the methods to create tuples and add them to the relations, e.g., the tuples consisting of an interactor, a command and the application model, which can be added to the relation `Commands`; and
- the methods to find specific tuples in a relation and remove them from the relation.

4.2 Modeling the Behavior of the Multi-User Interface

Once we have constructed the multi-user interface class diagram, we have to specify the behavior of the different classes represented in the diagram, that is, the behavior of the instances of these classes. Class behavior and the behavior of user interface objects have been defined before using formal, graphical notations, in particular extended state transition diagrams and Statecharts (also a form of extended state transition diagrams incorporating concurrency) [Harel et al., 1990; Harel, 1987]. For example, the work by Booch [1991], Interaction Objects [Jacobs, 1986], Objectcharts [Coleman et al., 1992], and Statemaster [Wellner, 1989].

We have chosen to use the SM and GMB modeling languages [Estrin et al., 1986], following the work done in the coSARA system [Mujica, 1991], and because:

- It is possible to construct a test environment for our model using the SM and GMB languages themselves [Razouk et al., 1979], and therefore it is possible to test the model by executing the token machine on the test environment and the model at the same time, as we showed in Section 3.8;

- It is possible to automatically synthesize SM and GMB models from semi-formal statements of requirements [Lor and Berry, 1991]. This can be useful when trying to model new types of interactors and specially new applications.

4.2.1 The Interface's Structural Model

The structural model of a multi-user interface is a high-level representation of the classes composing the interface, including their hierarchical organization and communication patterns. The three graphical primitives of the SM modeling language [Estrin et al., 1986] are used to describe such a model: (1) *modules* represent application and user interface components; (2) *sockets* represent the modules' communication ports; and (3) *interconnections* represent communication paths between sockets in different modules. The model is derived from the interface's class diagram, so that the nesting of modules reflects the composition of classes.

The structural model of the drawing tool's interface is shown in Fig. 19. At the top-level we find a module with no sockets, `UI`, which represents the interface class. It contains two modules, `Interactor` and `Tool`, representing the classes of the only interactor required by the application and of the application model, respectively. The structure of each of these classes is then modeled by hierarchically decomposing modules into submodules, representing component classes or encapsulated functionality, until the behavior of each module is precise enough to be defined by a single behavioral model. In the case of the interactors, this occurs at the level of individual contact and dialog modules whose behavior is then represented in the terms described in Chapter 3. In the case of application models, the decomposition process finishes at the level in which each module encapsulates the behavior of the application in response to a specific command from some interactor, as we will see in the next section.

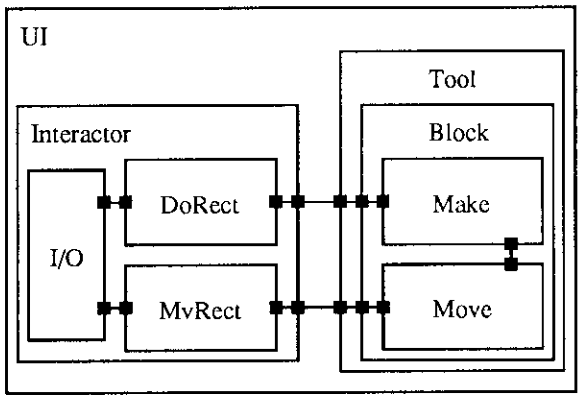


Figure 19: The structural model of the drawing tool's user interface.

Module `Interactor` contains submodules `I/O`, which represents the interactor's contact, and `DoRect` and `MvRect`, which represent the two interactor's dialogs to produce and move blocks, respectively. Module `Tool` contains submodule `Block`, which represents the class of objects handled by the application and which, in turn, has been partitioned into submodules `MakeFig` and `MoveFig`. These submodules represent the functionality of the application divided according to the commands it can receive from the interactors.

It is important to notice that although the structural model is derived from the corresponding class diagram, it is not an equivalent representation:

1. There are no modules representing classes `Command`, `Call`, `Action` and `Response`. The reason is that these classes do not really present a behavior of their own. Their instances are rather passive entities that are sent and received by other, active entities, in particular the application model and the interactors. Each of these active entities has a particular way of representing the passive entities that they send and receive.

2. There are no modules representing the relations. Because all the relations in the class diagram define which object communicate with which other object and by sending what types of messages, we have chosen to use the explicit communication primitives of SM, that is, sockets and interconnections, to represent the relations. This needs not be the case for other types of relations.

3. The class `Dialog` is represented by two different modules. This is a consequence of the fact that what could otherwise be a complex dialog between user and application, allowing the user to create and move objects and allowing the application to graphically represent to the user the results of these actions, can be conveniently divided into two much simpler dialogs for which we already have defined the appropriate behavioral encapsulations.

4. The class `Block` is represented by a module containing two different communicating submodules, which are explained in the next section.

Finally, it is also important to notice that in the structural model, whenever two unrefined modules communicate with each other, the communication is represented by a single interconnection from a socket in one of the modules to a socket in the other. It is possible that in the process of refining those modules into submodules or behavioral models, this single interconnection be replaced by two or more interconnections, as we will see below.

4.2.2 The Interface's Application Model

The exact nature of the application model component in the Seeheim model of user interfaces [Green, 1985; Green, 1986] has received much attention recently [Hudson, 1987; Hartson, 1989; Hurley and Sibert, 1989; Wood and Gray, 1992]. This is a consequence of the fact that the concept of separation of concerns between the user interface software and the application software, which is strongly supported by the application model component of the Seeheim model, often conflicts with the requirements of modern, high-quality user interfaces that provide semantic feedback to the users. The Tailor tool, using a methodology called CREASE [Hurley and Sibert, 1989], and the fully-explicit dynamic and reconfigurable linkage component of the Chimera UIMS [Wood and Gray, 1992] are two approaches to dealing with this problem.

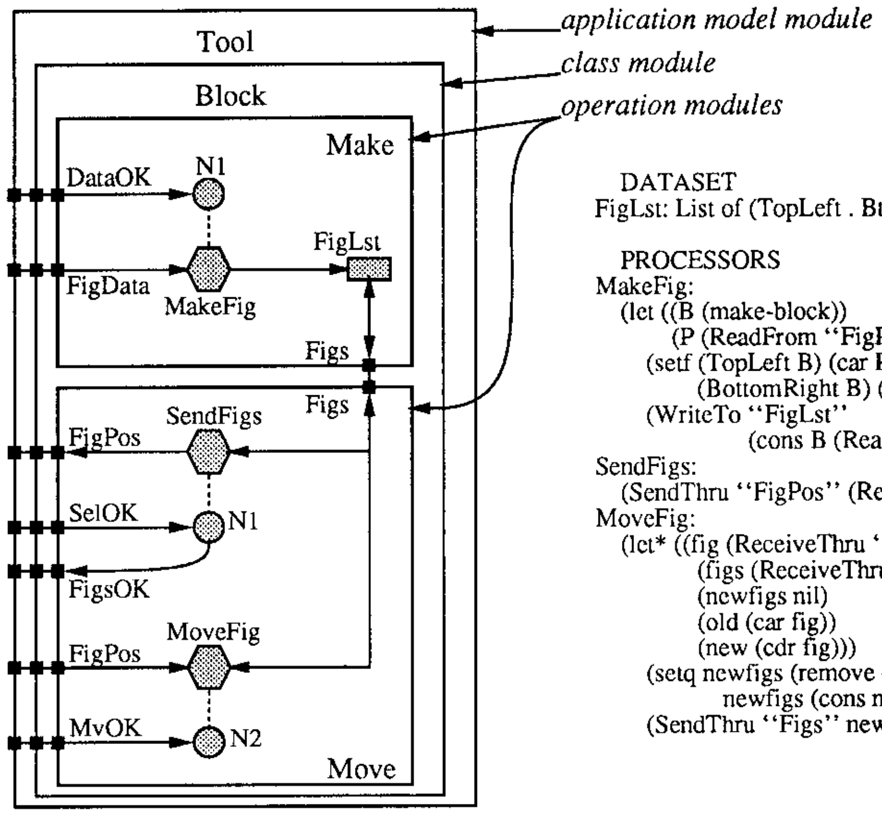
In our operational model of multi-application multi-user interfaces, the application model component is specified using an object-oriented approach from the point of view of the functionality that the application provides to a user through the interface. The full behavioral model of the application model in the case of the drawing tool is shown in Fig. 20. From a high-level perspective, the specification of the module representing this component in the behavioral model of the interface is derived from the composite class specification for the same component in the class diagram of the interface:

1. The application model module contains one submodule for each component class in the application model composite class; these are called *class modules*. Thus, for example, in the case of the drawing tool, module `Tool` contains one class module, `Block`, representing the class of all blocks handled by the application.

2. Each class module, in turn, contains one submodule for each operation that users can perform through the interface on the instances of the corresponding class, plus one submodule representing the instantiation operation for the class, which in general is also available to the users. These submodules are called *operation modules*. In the case of the drawing tool, users can create and move blocks. Therefore, class module `Block` contains operation modules `Make` and `Move`.

From a low-level perspective, the final aspect that remains to be specified is the actual behavior of each operation module, in terms of a control graph, a data graph, and an interpretation associated with the data graph.

1. Each class module contains a dataset to store all the objects belonging to the corresponding class. The module representing the instantiation operation for the class is the appropriate module to store such dataset. Thus, dataset `FigLst` is defined in module `Make`. As we can see from the interpretation associated with `FigLst`, it consists of a list of Lisp dotted pairs whose components represent the `TopLeft` and `BtmRight` attribute slots of the class `Block`. Thus, each pair in the list represents a different block.



```
  DATASET
FigLst: List of (TopLeft . BtmRight)

  PROCESSORS
MakeFig:
  (let ((B (make-block))
        (P (ReadFrom "FigPos")))
    (setf (TopLeft B) (car P)
          (BottomRight B) (cdr P))
    (WriteTo "FigLst"
                 (cons B (ReadFrom "FigLst")))
SendFigs:
  (SendThru "FigPos" (ReceiveThru "Figs"))
MoveFig:
  (let* ((fig (ReceiveThru "FigPos"))
          (figs (ReceiveThru "Figs"))
          (newfigs nil)
          (old (car fig))
          (new (cdr fig)))
    (setq newfigs (remove old figs)
           newfigs (cons new newfigs))
    (SendThru "Figs" newfigs))
```

Figure 20: Object-oriented behavioral model of the drawing tool's application model.

2. The instantiation operation module also contains a (node, processor) pair which becomes active every time the data necessary to create a new object is ready in one of the interactors connected to the application model module. The availability of the data is signaled by a token placed on a control arc connected to the node; the data is then read by the processor through a data arc. Thus, module `Make` contains node `N1`, processor `MakeFig`, control arc `DataOK`, and data arc `FigData`.

3. For the remaining operation modules we can only say that each of them should be able to read and write the dataset in the instantiation operation module, and should contain one (node, processor) pair for each command that can be received from the interactors and is part of the operation represented by the module. Each pair is activated and receives data directly from the interactors, and can send control and data back to the interactors. The commands represented in module `Move` are the collection of data for the selection of the block that is about to be moved, represented by pair (`N1`, `SendFigs`), and the communication of the new position for the selected

block, represented by pair (`N2`, `MoveFig`).

4.2.3 The Behavior of the Interactors: Contacts and Dialogs

The behavior of contacts and dialogs has been presented in detail in Chapter 3. Therefore, the description given here is very brief. A contact models the actions, produced by the users, to which a screen object responds (e.g., pressing a key on the keyboard, clicking or moving the mouse, etc.), and the responses that the screen object produces due to requests received from the dialogs (e.g., highlighting a screen region, drawing or erasing a figure, etc.). Dialogs model the syntax of the communication between the users and the application, specifying which sequences of actions received through the contacts are valid, and the points in these sequences at which information is sent back to the contacts and commands are sent to the application model. Each dialog describes one valid sequence of actions.

Contacts and dialogs are represented by SM modules, which communicate via sockets and interconnections. Their behavior is described using GMB models. The multi-user interface of the drawing tool contains one interactor per user, each interactor containing one contact and two dialogs. Module `I/O` in Fig. 22(d) represents the window where users perform the actions and observe the responses to these actions. The actions are handled by processor `GetAction` and result in information being sent to the dialogs through `GetAction`'s output data arc, `Pos`, and node `N1`'s output control arcs, `LDn`, `LUp`, `RDn` and `RUp`. Both the data arc and the control arcs are directly connected to the dialogs. The responses that the contact produces are the result of the activation of the processors `DrawFig` and `EraseFig`, due to requests received from the dialogs through the contact's input data arc `Pts` and input control arcs `Draw` and `Erase`.

Modules `DoRect` and `MvRect` in Fig. 22(a) define the action sequences required to produce and move rectangles. The behavior of `DoRect` is shown in Fig. 22(e) and explained in more detail later in the chapter. Essentially, it expects two actions in sequence, and when it gets the second it informs both the application model and the contact that a new rectangle has been defined. Modules representing contacts and dialogs that behave in this way can be obtained from the system's Library, or they can be defined by the interface designers.

4.3 The Instantiation and Behavior-Linking Process

The third step in our design and development method is to create the actual objects (class instances) that participate in the multi-user interface, that is, the interactors and the application model, and to link them to the behavioral models of the corresponding classes.

Before we actually describe this process in more detail, it must be pointed out that designers have first to complete the behavioral models of the previous step by associating them with the appropriate interpretation code. This interpretation code, that is, the executable code defining the functions carried out by the processors and the data types of the datasets, resides in the system's Library in the form of software modules, and is associated with the models using the GMB graphical editor, the same editor used to produce the models. Through a special purpose dialog box, this editor requests the names of the software modules' from the designers, then finds them in the Library, and finally links them to the corresponding processors and datasets. The system's Library contains a collection of general purpose software modules, as well as modules generated by the OREL compiler from class models. Any module specific to an application or user interface which is not yet part of the Library and is not generated by the OREL compiler from the class diagram, has to be programmed by the designers and stored in the Library before it can be associated with the models.

At this point, the GMB models represent a complete specification of the behavior of the application and its multi-user interface. A multi-user interface object can be created now by instantiating the top-level `UI` class. As we said before, the interface class definition and the function to instantiate it are produced by the OREL compiler from the interface's class diagram (for example, the diagram shown in Fig. 18). The behavior of this interface object is then defined in terms of the behavior of its components, an application object and a collection of interactor objects. These objects have to be instantiated, though, before they can have any behavior. Fig. 21 shows a dataflow diagram of this instantiation process and the following linking processes by which behavior is associated with the instances.

An application object is instantiated using the appropriate class definition and instantiation function produced by the OREL compiler from the class diagram (subprocess A in the figure). Its behavior is defined simply by linking it to the GMB model of the application model, that is, by storing in one of the object's slots a pointer to the model.

A slightly different process has to be carried out for the interactor components of the multi-user interface. The idea is again that interactor objects are instantiated from the `Interactor` class definition produced by the OREL compiler (subprocess B in the figure), and then each of them is linked to the corresponding GMB model. We notice, however, that the interactors in the operational model and class diagram of the interface are really abstract representations of the actual screen objects that will eventually be used for user-application interaction. In order to keep the specifications independent of a particular window manager software, the interactors themselves do not become screen objects through the design method. Instead, they are mapped to screen objects created from a collection of software modules which reside in the Library and can be used by the designers (subprocess C in the figure). These software modules include class definitions and the corresponding instantiation and manipulation methods written in CLUE (the Common Lisp User interface Environment), implementing screen

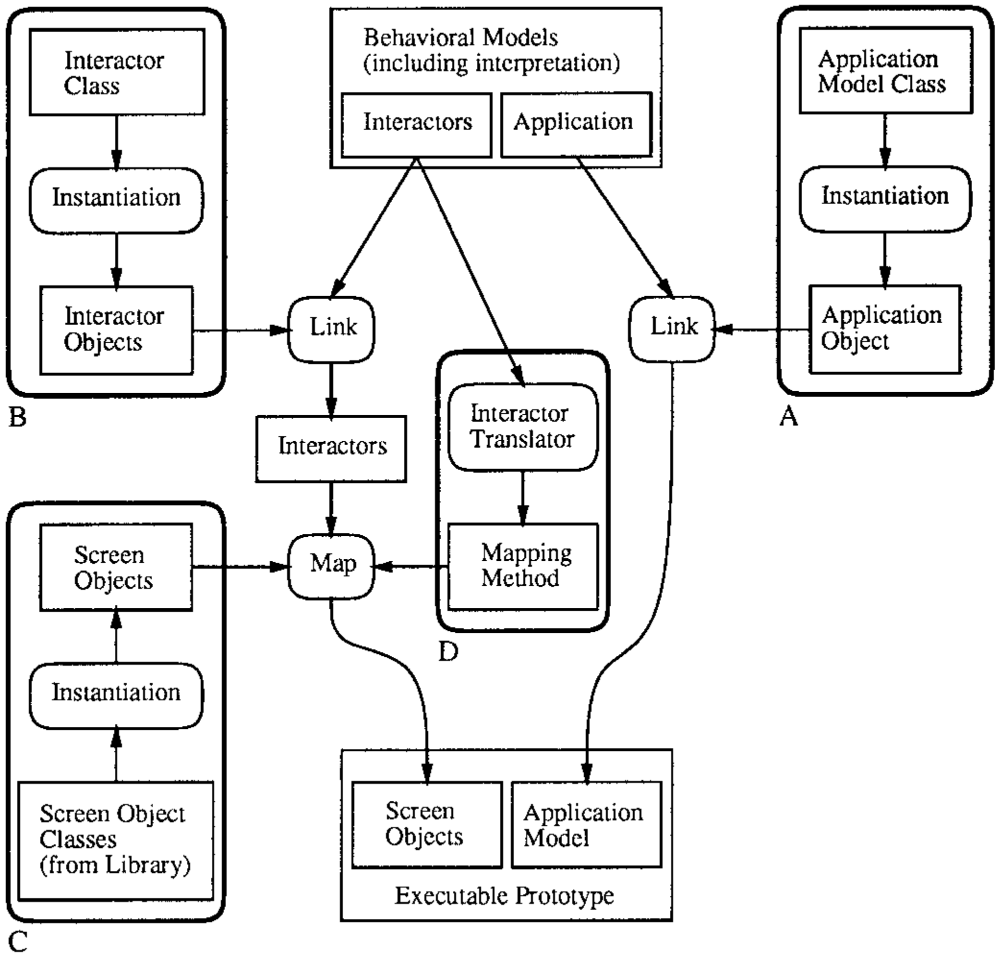


Figure 21: Producing executable instances of an application and its user interface, from object-oriented specifications.

objects such as: buttons, text collectors, drawing windows, menus (a composition of buttons), dialog boxes and fill-in forms (two types of compositions of buttons and text-collectors), and canvases (a composition of buttons, text-collectors and drawing windows).

Mapping a real screen object, once it has been instantiated, to the corresponding interactor means effectively making the screen object sensitive to the user-generated actions specified in the interactor's contact by its output control arcs. More specifically, the idea is that when the screen object detects the occurrence of one of those actions, a token should be placed on the appropriate control arc of the corresponding interactor's contact. This is accomplished through a two-stage subprocess (subprocess D in the figure). First, a tool called the *Interactor Translator* is executed using the GMB models of the contacts as input (the actual CLOS code is presented in Appendix B). Essentially, this tool produces the interactors' mapping method, which has to be executed next, once the screen objects have been instantiated. When executed, this mapping method requests an actual screen object for each interactor in the multi-user interface's behavioral model, and then adds a mapping to each of these screen objects. These mappings constitute concrete implementations of the `GetAction` processes in the contacts (described in the previous section and also in section 3.5), translating the actions produced by the users into data and tokens placed on the contacts' output data arcs and output control arcs.

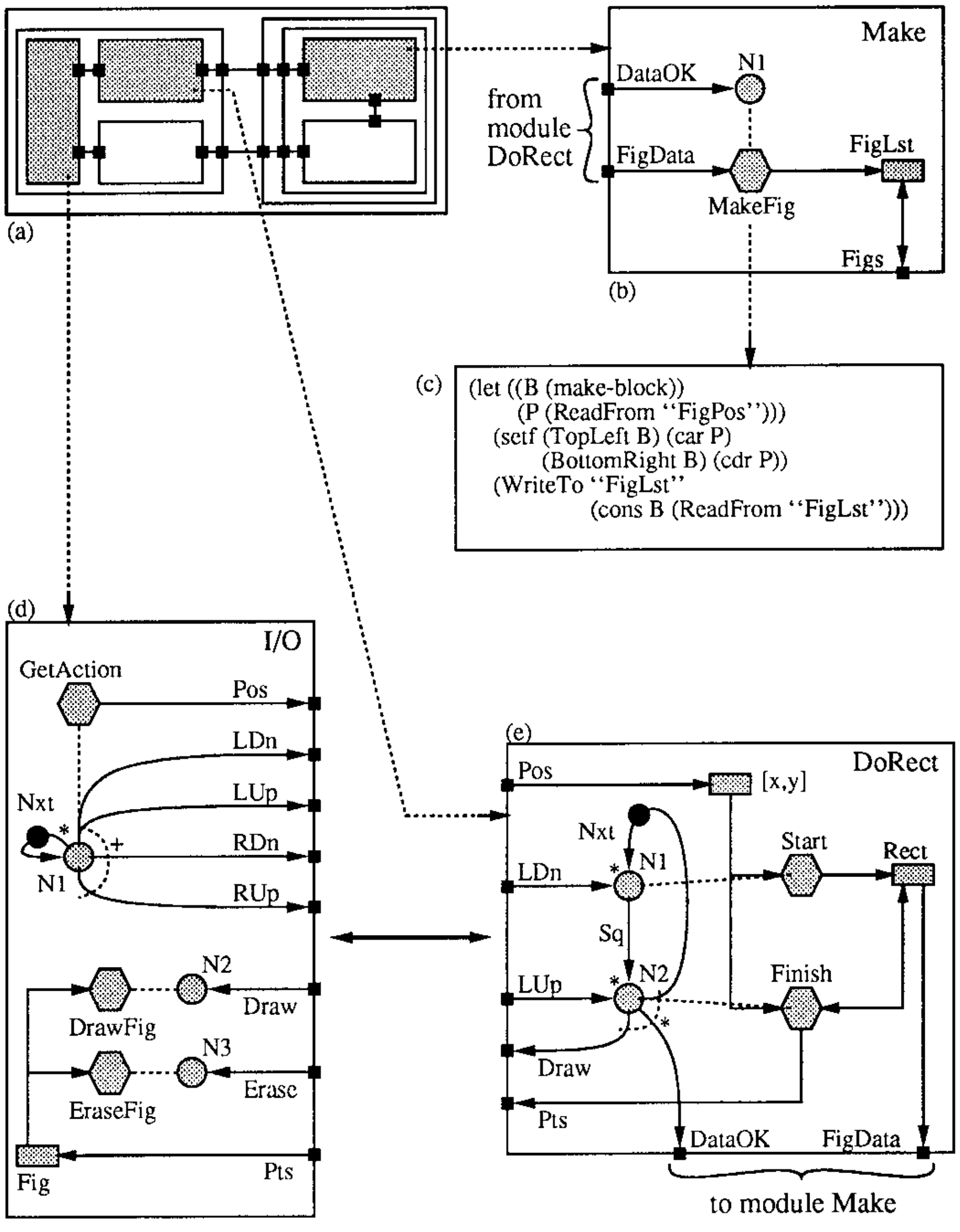
The exact nature of the mappings, and therefore the code implementing the Interactor Translator and mapping method, depends on the particular software system actually being used to manage the windows. On the other hand, the application, the application model and the interactors, and hence the multi-user interface, are completely independent of such details. In our case, we run CLUE on top of both CLOS (the Common Lisp Object System) and CLX (the Common Lisp interface to the X window system). CLUE provides a callback mechanism through which the occurrence of X events, that is, the user generated actions, are automatically translated into the invocation of dynamically defined functions. After this mapping process is done, the application object and the corresponding multi-user interface can be executed by the system's token machine, as we explain below. It is important to notice, though, that this execution is actually driven by users operating real input devices on real screen objects.

4.4 The Execution Process

The *token machine* is an interpreter of GMB models. Its formal definition is found in [Vernon, 1983], but for the purpose of this exposition the brief definition provided in section 2.5.2 is enough. We describe now the activities that take place in the user interface models of the drawing tool, shown in Fig. 22, when a user draws a rectangle on the drawing window, defining a block.

We explained in the beginning of this chapter that to draw a rectangle on the window, the user has to press down the mouse's left button at the position of the rectangle's top lefthand corner, then release the button after dragging the mouse to the position of the rectangle's bottom righthand corner. The contact `I/O` in Fig. 22(d) defines the input/output behavior of the drawing window. When the user presses down the button, processor `GetAction` sends the window position of this action through arc `Pos`, and then places a token on arc `LDn`. When the button is released, `GetAction` sends again the action's window position through arc `Pos`, but now the token is placed on arc `LUp`.

`GetAction` is originally activated by the initial token on arc `Nxt`, which enables and eventually fires node `N1`. When `GetAction` becomes active, it waits until an action occurs, at which time it operates as indicated. Besides placing a token on either arc `LDn` or `LUp`, `GetAction` always places a token on `Nxt` to be able to



```
(let ((B (make-block))
      (P (ReadFrom “FigPos”)))
  (setf (TopLeft B) (car P)
        (BottomRight B) (cdr P))
  (WriteTo “FigLst”
           (cons B (ReadFrom “FigLst”))))
```

Figure 22: The models of the drawing tool: (a) Sketch of SM model; (b) GMB model of `Make`; (c) Interpretation for processor `MakeFig`; (d) GMB model of `I/O`; (e) GMB model of `DoRect`.

process the next action.

The tokens and data produced by the contact travel to the dialog DoRect. It is DoRect, as shown in Fig. 22(e), that really requires a sequence of two actions to produce a rectangle. Each action comes in the form of a window position initially stored in dataset [x,y] and a token placed on either arc LDn, for the first action, or LUp, for the second. When the first action is received, processor Start becomes active; its interpretation simply transfers the position stored in [x,y] to dataset Rect.

When the second action is received, processor Finish becomes active. It also transfers the new position stored in [x,y] to Rect, which now stores the two positions that together define the rectangle. But Finish does something else: it informs both module Make and the contact I/O that a new rectangle has been defined. To inform Make, so that Make can create the corresponding semantic block, Finish simply places a token on arc DataOK. To inform the contact, so that it can draw the rectangle on the screen, Finish sends both pieces of data in Rect through arc Pts and then places a token on arc Draw. This token will eventually cause processor DrawFig in the contact to become active and do the drawing.

In module Make, as shown in Fig. 22(b), the token on DataOK enables and eventually fires node N1, thereby activating processor MakeFig. MakeFig's interpretation, shown in Fig. 22(c), consists of a sequence of calls to the methods produced by the OREL compiler, which are stored in the system's Library. First, MakeFig creates a new block, i.e., a new instance of the class Block. Then, it reads the positions of the block's top-left and bottom-right corners from DoRect through arc FigData, and stores them into the corresponding slots of the block (see Fig. 17(b)). Finally, MakeFig stores the new block by adding it to dataset FigLst.

Data arc Figs is connected to module Move, as shown in Fig. 20. Its purpose is to give Move access to all the existing blocks, so that when a user moves one of them on the window, Move can update the values of the corresponding slots TopLeft and BtmRight.

4.5 Comparison with Related Work

Our multi-user interface design method has evolved from the coSARA tool model and associated process to build and extend tools for the coSARA environment [Mujica, 1991]. While in coSARA the model and the process are centered around the tool, or application, our method is centered around the application's interface. The tool model describes the class diagram for a generic application as consisting of the simple classes Application, Screen Object and Message, all participating in the relation Callbacks. Class Application contains an attribute slot DCM. The intended meaning of this diagram is that an application object communicates directly with a collection of actual screen objects implemented as CLUE contacts in order to interact with the users. The communication between the application object and the screen objects is via messages in the form of CLUE callbacks, although there is no distinction between messages going in one direction, for example, from the application to the screen objects, from those going in the opposite direction (while CLUE callbacks provide a convenient mechanism to inform applications about the results of user actions on the screen objects, they do not provide a natural way for applications to send data back to the screen objects). A relevant characteristic of the tool model is that the application itself is in charge of defining the syntax of the interaction with the user. This definition is stored in the slot `DCM` and, in principle, is produced automatically from a GMB-based specification of the interaction. Thus, in the tool model, applications are completely aware of the details of the ways in which users access their functionality, while the model itself and its associated building process are strongly based on CLUE.

Our design method, on the other hand, provides for applications and screen objects which are much more independent of each other. To begin with, our method is not concerned with applications themselves but only with their abstract representations for the purposes of their interaction with users. These representations, that we call application models, are only aware of their own functionality and of a few, well defined ports (the sockets in the SM-GMB models) through which this functionality can be accessed from the outside, and through which they can send information back to the outside. At the other end of the picture, users can interact directly with any type of user interface toolkit. This interaction is abstracted by our representations of screen objects, which we call interactors. Interactors provide a standard way of representing user actions and application responses, and handle all the syntax involved in the interactions, including syntactic feedback to the users. Actually, even the syntactic components of the interactors, the dialogs, are not aware of the specific actions that users produce on the screen objects by manipulating input devices, or of the specific graphics that the screen objects can handle. Both of these aspects, which are defined by the particular toolkit handling the windows and input devices, are abstracted by the input/output component of each interactor, the contact.

5 Concurrent Application Sharing and Interface Reconfigurability

Two important characteristics of multi-application multi-user interfaces are those of supporting:

1. *Multiple users concurrently sharing multiple applications and their data.* During a session on a collaborative computer system, when two or more users concurrently share an application, it is likely that at some point during the interaction some of them may attempt to access the same application's data at the same time in conflicting ways. For example, two users may want to move the same figure to two different positions.

2. *Dynamic reconfigurability of the interface's structure and behavior.* As new users join an ongoing collaborative session, participating users leave the session, new applications are activated, and/or active applications are quit, the configurations of the user interfaces of the individual applications and the configuration of the system's user interface must change accordingly.

Therefore, there are two important problems to be addressed when modeling and specifying multi-application multi-user interfaces:

1. Include the necessary *control mechanisms* to avoid potential conflicts due to concurrent access to application data or functionality; and

2. Include the necessary *configuration mechanisms* to allow changes in the number of users and active applications during the operation of an interface.

These mechanisms should be reflected explicitly in the model and, later on, enforced automatically during the simulation of the model or the execution of the prototype derived from the model. The models of contacts, dialogs, and application models presented so far have the potential to be used in the specification of a multi-application multi-user interface with the characteristics just described, because the underlying formal notations—in particular, the graph model of behavior—are able to model concurrency, communication and synchronization.

5.1 Specification of User Coordination for Concurrent Application Sharing

We have seen that when the single-user version of a user interface consists of a collection of contacts connected to a collection of dialogs which in turn are connected to one application model, then in the corresponding multi-user version both the contacts and the dialogs are replicated for each user and the application model is extended to be able to communicate with all the new dialogs, such that: (1) there is one instance of each original contact for each user; (2) there is one instance of each original dialog for each user; and (3) there is still only one instance of the application model.

Fig. 23 shows the structural model of the drawing tool's multi-user interface introduced in chapter 4, for the case in which K users are sharing the tool concurrently and there is no coordination among their activities at the level of the interface. Each user can attempt to access the tool's functionality and/or data at any time. If the users do not want their accesses to conflict, they have to coordinate their activities by essentially social interaction.

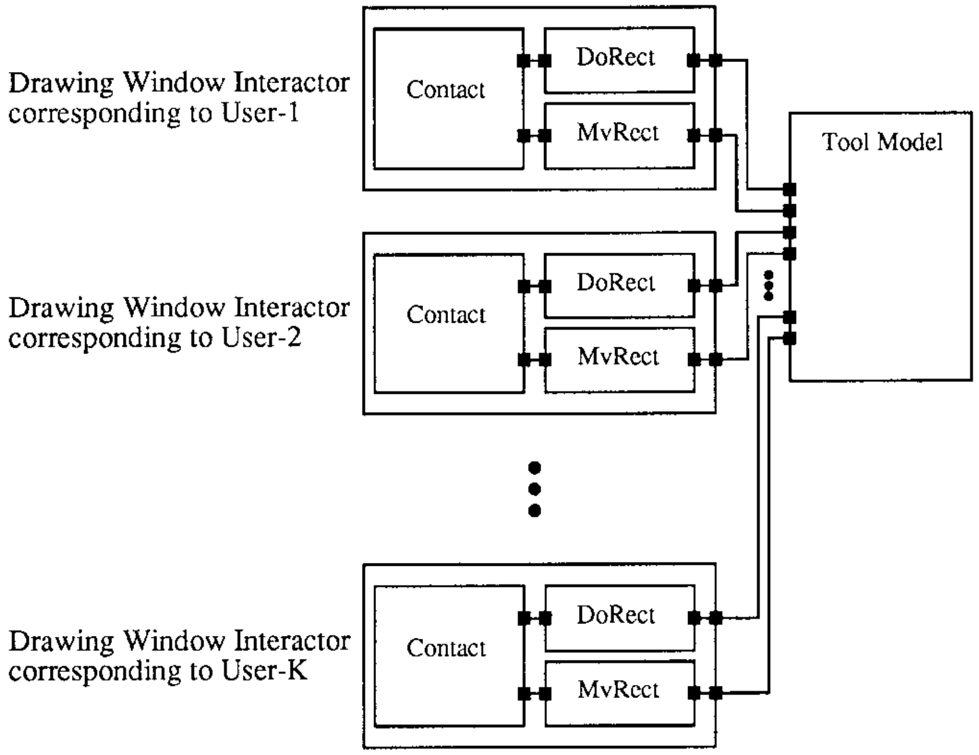

Drawing Window Interactor corresponding to User-1
Contact
DoRect
MvRect
Tool Model
Drawing Window Interactor corresponding to User-2
Contact
DoRect
MvRect
Drawing Window Interactor corresponding to User-K
Contact
DoRect
MvRect

Figure 23: Multi-user interface for K users with no explicit coordination.

5.1.1 Application-Independent Coordination

Concurrent access by multiple users to the data and functionality of an application is controlled primarily by the application itself, as it has all the necessary knowledge to decide what can be accessed safely and when. However, the multi-user interface specification can help in this task by preventing some potentially conflicting accesses from actually occurring. This is possible because the entities that actually perform the read and write operations on application data directly available to the interface are the processors in the application model. These processors have read and/or write accesses to the datasets which store the data in the application model. Thus, to prevent two or more processors with write access to the same dataset from attempting to write to that dataset at the same time, we essentially have to prevent the processors from being active simultaneously or during overlapping periods of time. We can achieve this by making the activation of these processors be mutually exclusive, that is, by making the firing of the corresponding nodes be mutually exclusive.

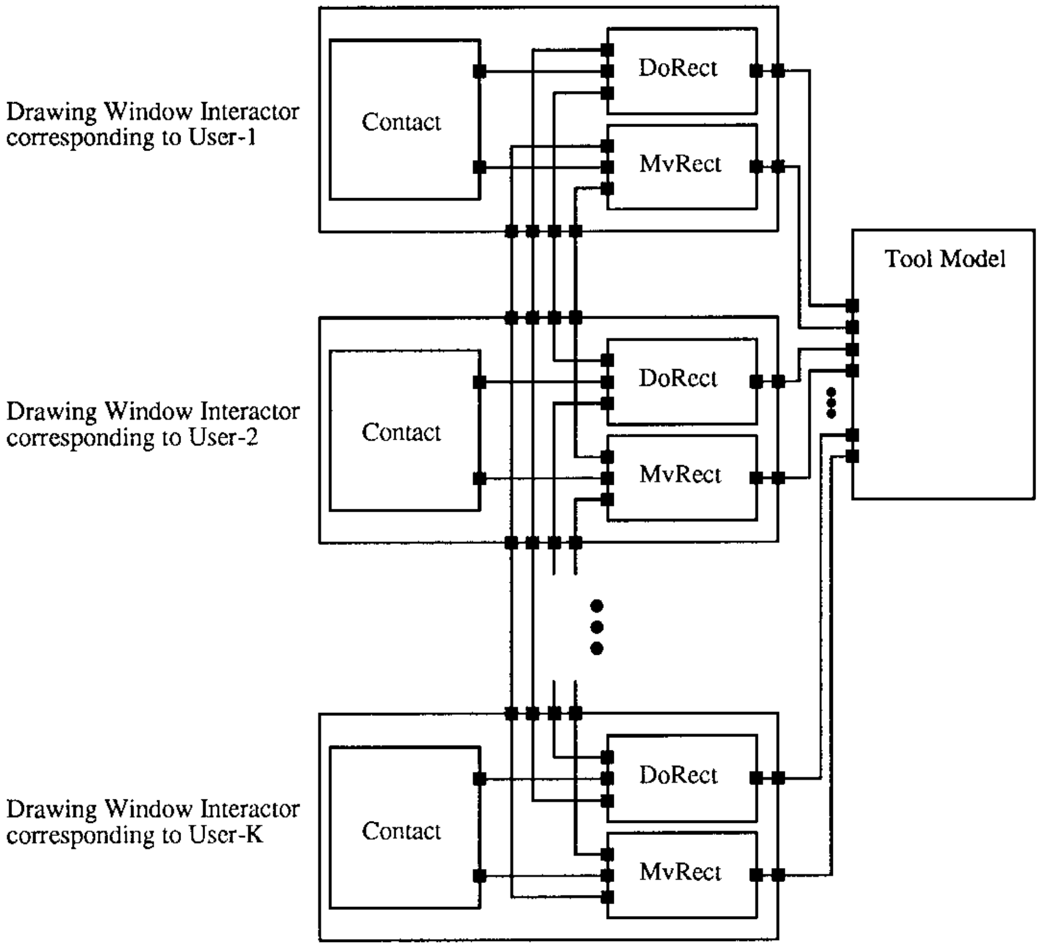
In principle, it is possible to add to the control graphs of the application models the extra logic necessary to enforce mutually exclusive operation among those processors with write access to the same dataset. However, it is more convenient to include this extra logic at the level of the dialog modules that generate the input control signals to the application models, for two reasons:

- *Flexibility.* There is only one type of mutually exclusive operation that can be imposed on the processors of an application model, namely, make all processors with write access to the same dataset operate in a mutually exclusive way among them. On the other hand, we can impose mutually exclusive operation among equivalent dialog modules associated with different users at different levels within the modules. For example, we can subject to mutually exclusive operation across equivalent dialogs: the execution of the complete dialogs, or the execution of the final step of the dialogs, or the execution of each step of the dialogs.

- *Application independence* and *encapsulation.* The structure and behavior of an application model depends to some extent on the actual application it represents. Therefore, any type of mutually exclusive operation that can be imposed on the processors of an application model has to be specified explicitly for each different application model at the time the model is built. On the other hand, the structure and behavior of the dialogs is known well in advance if we consider both the generalized dialog model and the specialized dialog models. Therefore, the logic of the different types of mutually exclusive operation can be incorporated into these models as standard components of the models.

Fig. 24 shows again the high-level structural model of the drawing tool's interface. This time, however, we include a number of extra interconnections between equivalent dialog modules in different interactor modules, corresponding to different users. These new interconnections represent the paths for the communication of control between the dialogs involved, for the purpose of coordinating the interaction of the different users. The details of how this control information is handled inside the dialogs for three different types of interconnection configurations are described below:

1. The control mechanism affects the complete dialogs, enforcing sequential execution;
2. The control mechanism affects the complete dialogs, enforcing mutually exclusive execution; and
3. The control mechanism affects only the exit subgraphs of the dialogs, enforcing mutually exclusive execution.



Drawing Window Interactor corresponding to User-1
Contact
DoRect
MvRect
Tool Model
Drawing Window Interactor corresponding to User-2
Contact
DoRect
MvRect
Drawing Window Interactor corresponding to User-K
Contact
DoRect
MvRect

Figure 24: Multi-user interface for K users with (some type of) coordination among equivalent dialogs.

5.1.2 Sequential Execution Among Equivalent Dialogs

In order to make the presentation that follows simple, let us consider a collaborative application whose interface to each user consists of one window supporting only one type of interaction, that is, one interactor which contains one contact and only one dialog connected to that contact. Fig. 25 shows the behavioral models of contacts and dialogs for this case, when K users are concurrently sharing one application.

The K dialogs have their control graphs interconnected in a loop and initialized in such a way that they can only be activated sequentially, one after the other in a predefined order. The dialog currently enabled, that is, the dialog with a single token on its control arc `Prev` (for example, the dialog corresponding to user-1 in the figure), will be active continuously from the time it is entered, when it receives a token on arc `Dn`, until the time it is exited, when it receives a token on arc `Up`. At this time the dialog will, among other activities, deposit a token on its control arc `Nxt`, equivalent to control arc `Prev` in the next dialog in the loop, which thus becomes enabled and can be activated (the dialog corresponding to user-2 in the figure).

5.1.3 Mutual Exclusion Among Equivalent Dialogs

Strict sequential execution among a group of collaborating users is not always necessary or desirable. A different approach is to let any user, but only one at a time, have access to a specific dialog. Fig. 26 shows the behavioral models of contacts and dialogs for this case, when K users are sharing one application concurrently.

The K dialogs have their control graphs interconnected and initialized in such a way that only one dialog can be active at any given time. The dialog is active continuously from the time it is entered until the time it is exited. Only then can another interconnected dialog become active. This type of behavior is achieved by interconnecting together all the `Nxt` control arcs, thus making each of them part of a single, more complex control arc, and then initializing this new control arc with only one token on it. It is important to notice that, as long as this single token is available on the arc `Nxt`, it can be "grabbed" by any of the K interconnected dialogs; in principle, it will be grabbed by the first dialog to receive a token on its control arc `Dn`. Therefore, this control structure does not prescribe any particular order for the activation of the dialogs, as did the structure described in the previous section.

The first step in implementing this control structure, is to augment each local arc `Nxt` as follows: we add an extra destination head going outside of the dialog module through a new socket, `NxtOut`, and we add an extra source tail coming from outside of the dialog module through a new socket, `NxtIn`.

The second step consists of interconnecting the K `Nxt` control arcs by connecting the new destination of one arc to the new source of another, as follows: socket `NxtOut` in the dialog associated with user K is directly connected to socket `NxtIn` in the dialog associated with user $K-1$; socket `NxtOut` of this dialog is directly connected to socket `NxtIn` in the dialog associated with user $K-2$; ...; finally, socket `NxtOut` in the dialog associated with user 1 is directly connected to

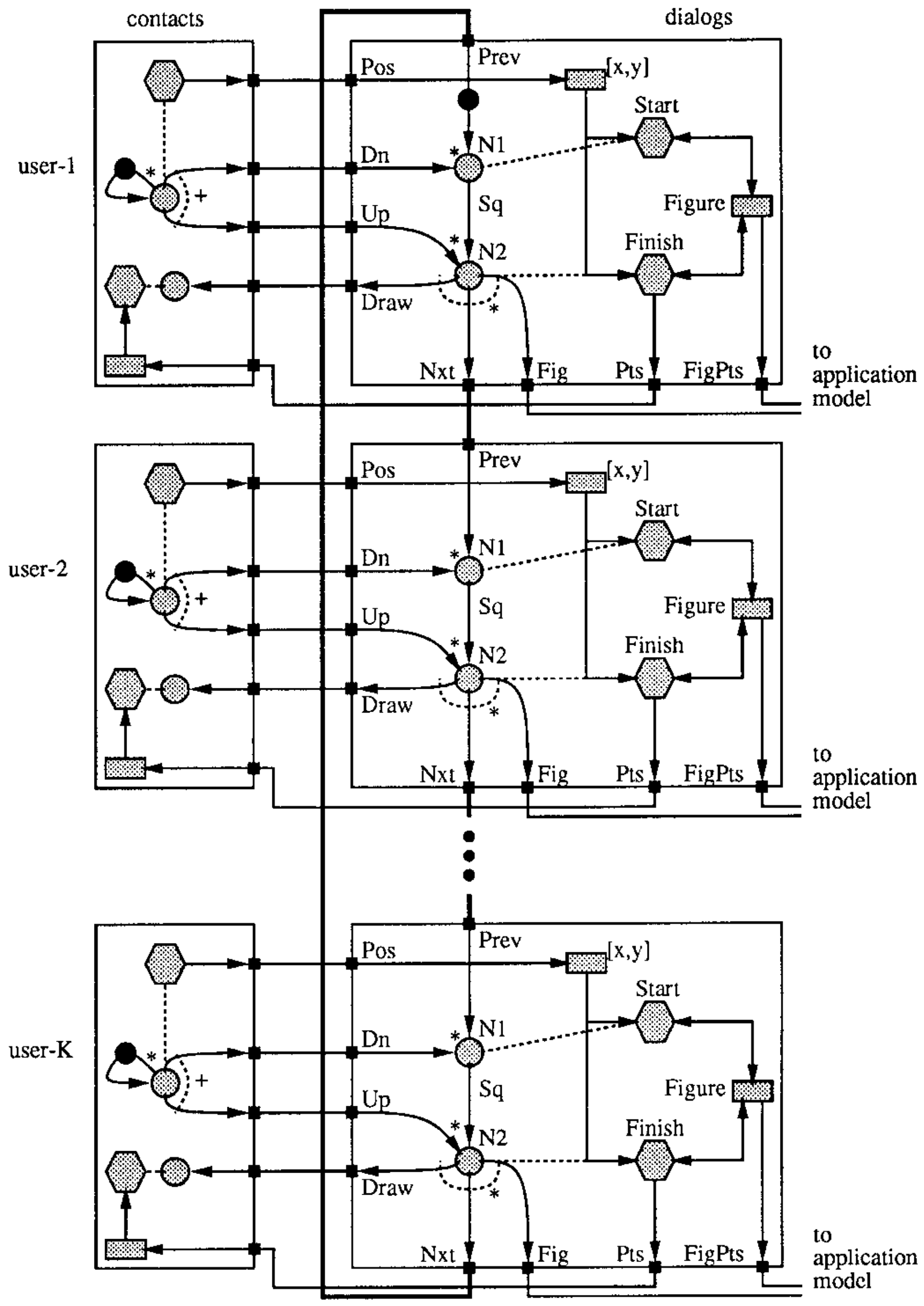


Figure 25: Sequential dialogs (for simplicity, modules enclosing contact-dialog pairs, representing interactors, are not shown).

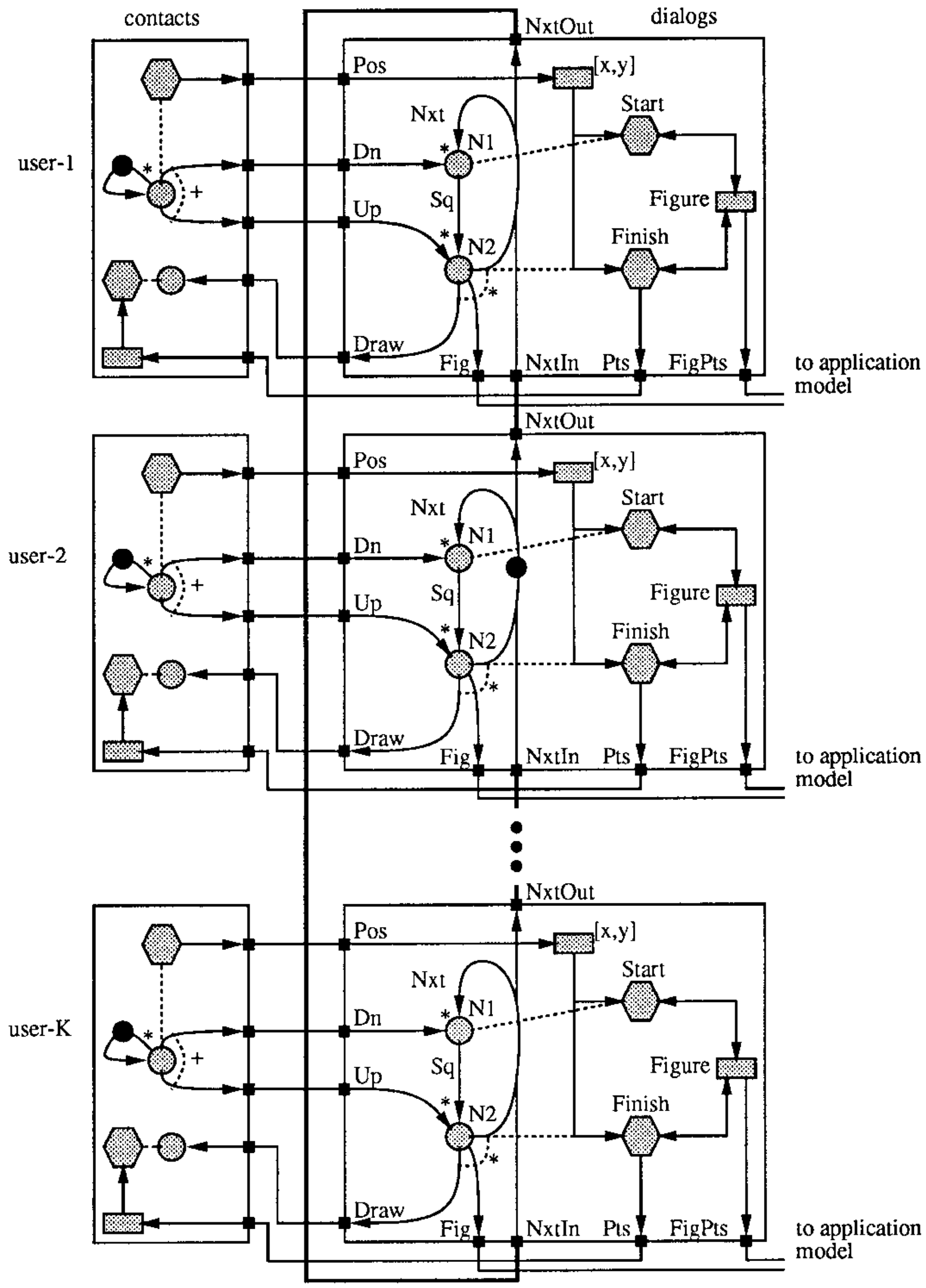


Figure 26: Mutually exclusive dialogs (interactors are not shown).

socket NxtIn in the dialog associated with user K.

5.1.4 Mutual Exclusion Among Dialogs' Exit Subgraphs

In many dialogs, the communication between the dialog and the application model occurs only at one point in the sequence of actions defining the dialog. A typical case is that in which the communication takes place right after the execution of the dialog is finished, that is, after the execution of the interpretation of the processor representing the last step in the sequence of actions described by the dialog. At this point, a token is placed on the only control arc coming out of the node mapped to that processor and going into the application model through an interconnection.

Therefore, in these cases two or more users can concurrently execute different copies of the same dialog, interacting with the same application, until but not including the dialog's last step. In order to guarantee conflict-free accesses to the application model after the execution of the last steps in these several copies of the same dialog, these steps have to be made operationally mutually exclusive. Fig. 27 shows again the behavioral models of contacts and dialogs for the case in which K users are concurrently sharing the same collaborative application, through an interface which consists of one contact and one dialog for each user.

In this case, the control graphs of the K dialogs are interconnected in such a way that several or all the dialogs can be active simultaneously, but only one dialog at a time can access the application's functionality or data as a consequence of executing that dialog's final step. As soon as that dialog has sent its request and the appropriate data to the application model, any one of the other interconnected dialogs can execute its final step and then access the application. To achieve this type of behavior we use an interconnection scheme similar to that used in the previous case, except that now we only involve the last node of each of the dialogs' control graphs being interconnected together, instead of the whole control graphs.

Essentially, we create a new complex control arc, Mx, which extends across the modules representing the dialogs through sockets and interconnections, and which is initialized with only one token on it. Within each dialog module, this new arc has two sources and two destinations. Node N2 itself, associated with the last step in the dialog sequence, represents one of the sources and one of the destinations. This augments N2's input and output logic expressions with arc **Mx** logically AND-ed to both of them. Two new sockets, MxIn and **MxOut**, represent the other source and the other destination, respectively. Finally, socket MxOut in the dialog associated with user i is interconnected to socket MxIn in the dialog associated with user $i-1$, for $i = K, K-1, \ldots, 2$; and socket MxOut in the dialog associated with user 1 is interconnected to socket MxIn in the dialog associated with user K.

Here again we should notice that as long as there is a single token on arc **Mx**, it can be grabbed by any of the K interconnected dialogs. In principle, the token

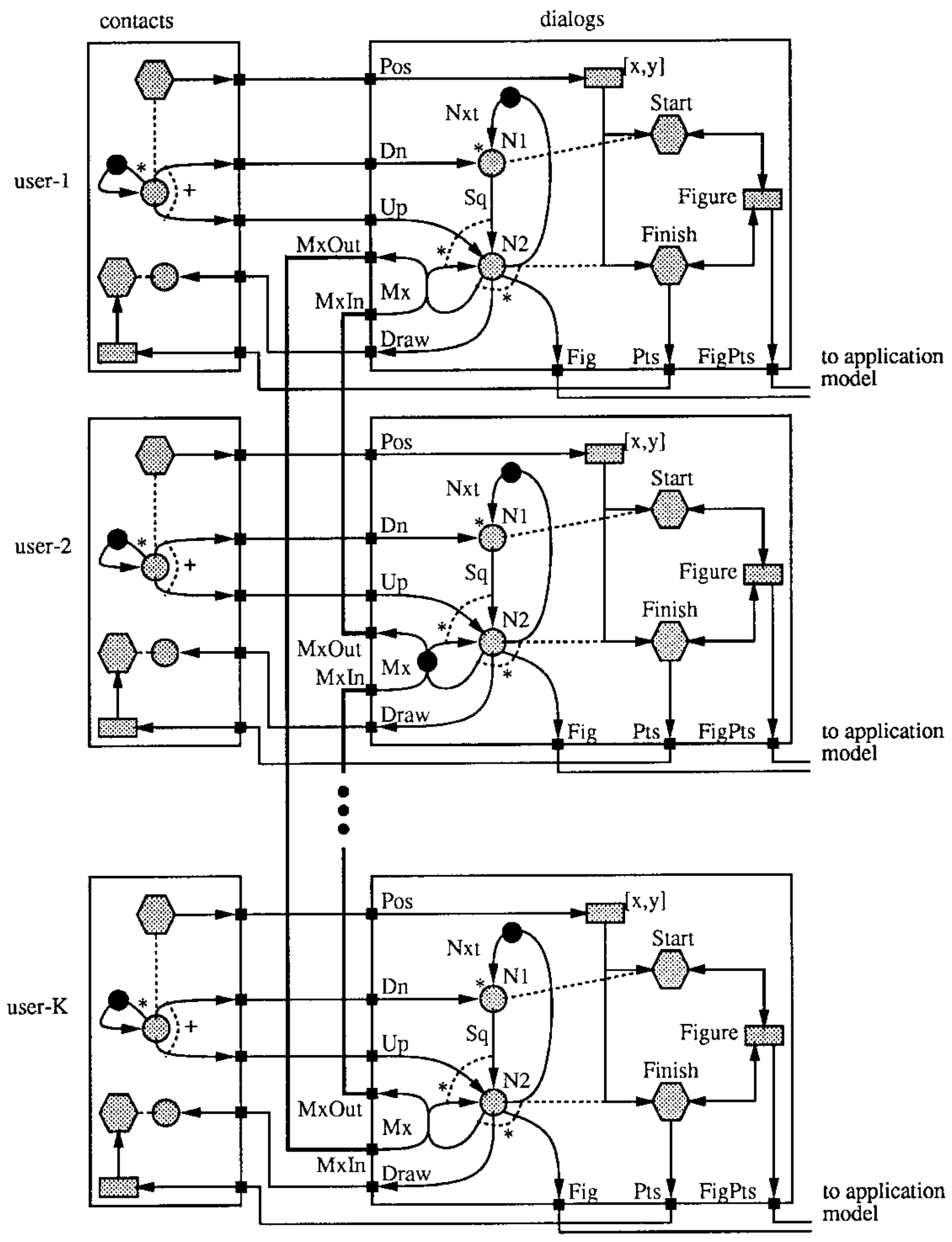


Figure 27: Mutual exclusion among dialogs' exit subgraphs (interactors are not shown).

will be grabbed by the first dialog to place tokens on both of its arcs Up and Sq. Thus, this control configuration does not prescribe any particular order for the termination of the dialogs.

5.2 Dynamic Reconfiguration of Multi-Application Multi-User Interfaces

An important problem to address when modeling multi-application multi-user interfaces is that of modeling changes in the structure and the behavior of the interface, that occur during its operation. In a multi-application system, the activation of a new application is likely to change the configuration of the system's interface to reflect the fact that the application is now available. For example, if the application provides its own collection of windows for user interaction, a new copy of these windows has to be created for each user currently in the session; or if the system's interface provides a menu of active applications, the menu has to be updated to include one more option. Similarly, in a multi-user system, in which several users are sharing the same application concurrently, the participation of a new user will change the application's interface. For example, if the interface provides a different, although equivalent, collection of screen objects to each user, it needs to be updated to handle one more collection of screen objects for the new user.

Throughout their development, SM and GMB models have assumed a fixed structure and behavior, which cannot change during the simulation of the models or during the operation of the prototypes built from the models. With that restriction we could model the multi-application multi-user interface of a system for a given number of applications and a given number of users, but we could not model situations in which a new application becomes available during a session, a currently available application is deactivated, a new user joins a session in progress, or a user currently participating in a session leaves it. It is very important that we be able to produce models that, first, reflect the types of changes allowed and, second, can still be subject to control flow analysis and animated simulation. Within the framework defined by the original SM and GMB modeling languages plus the extensions presented in Sections 3.5, 3.6 and 3.7 in terms of specialized modules representing contacts, dialogs and application models, we have found two modeling levels at which dynamic reconfiguration can be modeled properly without destroying the analysis or simulation capabilities:

- Behavioral changes occurring locally at the level of a dialog module, which can be modeled by adding or removing *subgraphs* (that is, coherent collections of nodes, processors, datasets and control and data arcs) to or from the control and data graphs of the dialog module.

- Structural and behavioral changes occurring at the level of a collection of communicating dialogs, which can be modeled by adding or removing the complete model of a dialog module, including its connections to other modules.

5.2.1 Dynamic Reconfiguration of Dialogs

An individual dialog can be altered dynamically during the operation of the user interface to which the dialog belongs, as a consequence of:

- A refinement procedure being applied to the interface; or
- A change in the functionality of the applications being executed through the interface.

Adding a rubber band effect to a dialog that produces rectangles, and adding a capability to delete the last segment to a dialog that produces polylines are examples of dialog refinement. Adding one more option to a menu dialog is an example of change due to increasing the applications' functionality. Looking at the generalized dialog model of Fig. 13, we see that these types of changes affect only the central step, or continuation, of the dialog, and are reflected in the number of one-node or two-node subgraphs, which can remove tokens from the arc **Sq** to activate some processor, and then can place tokens back on arc **Sq** when the processor becomes idle again. Consequently, we want to be able to model a dialog module in such a way that:

1. It is possible to add or remove one-node and two-node subgraphs in the dialog's central step;
2. The subgraphs added or removed are in what we call an *idle* state, that is, none of the nodes are enabled and all the processors are idle;
3. The subgraphs are added or removed together with the corresponding sockets, processors, datasets, control arcs and data arcs;
4. The changes occur during the execution of the user interface to which the dialog belongs, or during the simulation of this user interface's model; and
5. The changes reflect the fact that more or less capabilities are available, in the dialog itself or in the application, as the execution or simulation progresses.

We have to ensure that the specification of such a dynamically reconfigurable model does not affect the control flow of the complete specification in such a way that it is no longer possible to perform control flow analysis or animated simulation, or in such a way that it invalidates the results of analysis of the original graph. We approach this problem by considering the two types of effects that a change in the number of one-node or two-node subgraphs can produce in the flow of control with respect to the abilities to analyze and simulate the models:

1. The effect on the control flow internal to the graph that is being reconfigured, specifically with respect to the control arc Sq; and

2. The effect on the control flow of the rest of the specification, transmitted through the control arcs Cmd and Nxt.

First, let us study the effect on the control flow internal to the graph that is being reconfigured. Fig. 28 shows the control graphs for two instances of the generalized dialog, with 2 and 3 one-node subgraphs in its central step, each with a single token on its arc Sq.

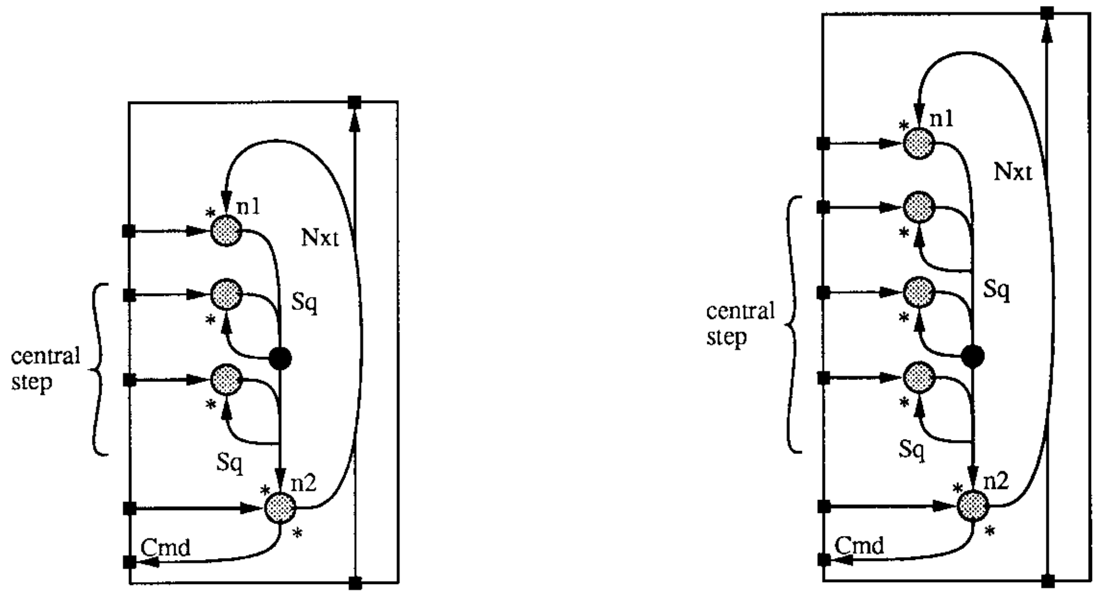


Figure 28: Comparing the control graphs of two dialogs, with two and three subgraphs in the central step: internal effect.

In this case, one more or less subgraph represents essentially one more or less way in which: (1) a token currently placed on arc Sq can be removed, eventually activating some processor; and (2) a token can be placed back on Sq, when the execution of the processor's interpretation is finished. But there is no way in which, by adding a subgraph that is in an idle state, Sq can suddenly contain two or more tokens simultaneously. This is because at most one of the subgraphs, among the original ones plus the new one, can become busy at any given time by removing the token on Sq; and this subgraph is then the only one that can place a token back on Sq when it becomes idle again.

Similarly, the only problem with removing a subgraph could occur if the subgraph removed were busy and "holding" the token from Sq, in which case the final step of the dialog would never be executed because node n2 would never become enabled. But we assume that the removed subgraph is idle. We guarantee this assumption by forcing the token machine interpreter to wait until it is true before proceeding with the modification. Therefore, after the number of subgraphs in the central step of the dialog has been changed by 1, there is always at most one token on Sq and, if there was a token on Sq when the change occurred, that token will eventually be back on Sq.

Second, let us study the effect on the control flow of the rest of the specification. Fig. 29 shows the control graphs for two instances of the generalized dialog, with 2 and 3 nodes in its central step, each with a single token on its arc **Nxt**. We have seen in the previous case that the reconfiguration has no effect in the internal control flow of the dialog. This essentially means that from the point of view of node n2, the presence or absence of a token on arc Sq behaves in the same way before and after the change. So, after the change, n2 and its mapped processor can still place only one token on arc Nxt and one on arc Cmd per each execution of the dialog, as was also the case before the change. But again the dialog can only be executed if the original token on Nxt is removed from there first by node **n1** and its mapped processor. Therefore, there is always at most one token on **Nxt**, both before and after the change, and thus the change has no effect on the control flow external to the dialog.

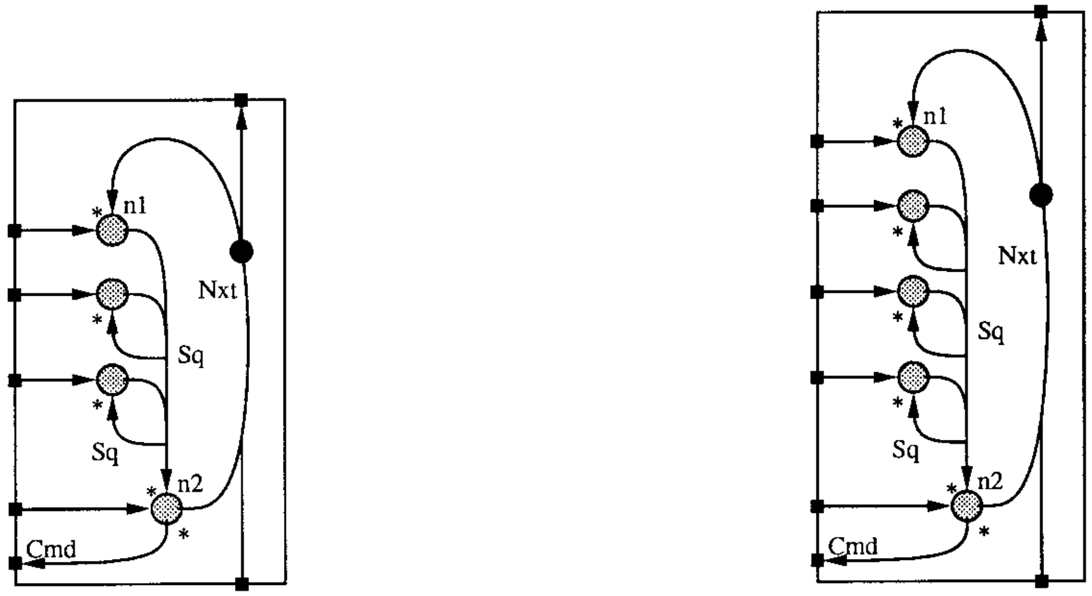


Figure 29: Comparing the control graphs of two dialogs, with two and three nodes in the central step: external effect.

5.2.2 Dynamic Reconfiguration of a Collection of Interacting Dialogs

The model of a multi-application multi-user interface operating in a collaborative system can change dynamically during its operation as a consequence of:

- The activation of a new application which provides its own application model and collection of interactors; or
- The participation of a new user who requests access to some or all the applications currently running in the system.

Reconfiguring an interface when a new application becomes active is certainly possible if the application's model and interactors do not communicate with any of the already active applications or their interactors. The only action that needs to be taken is to inform the token machine about the new control graphs that have to be interpreted. Furthermore, the result of the previous section shows that it is also possible to reconfigure the interface when some of the existing modules are affected by the new application, provided that those modules and the type of modification affecting them have the control structures described there. For example, if the interface includes a menu of active applications, then the activation of a new application means that the menu has to be updated to include one more entry. This reconfiguration can proceed along the lines presented in the previous section. Therefore, we will study now the reconfiguration of an interface when a new user joins an ongoing collaborative session.

In this case, the computer system has to provide the new participant with a user interface to each application currently running in the system. From the point of view of our operational model of multi-application multi-user interfaces, this new interface will consist of a new collection of interactors, each interactor being connected to one of the already existing application models. The first issue to notice here is that we are not replicating application models, only the collections of interactors connected to them. This is advantageous from the point of view of the design of the applications themselves, because each application designer then knows in advance that at any time the application will be communicating with just one application model (representing the application within the interface). This means, however, that the application models have to change in order to accommodate the new interactors on one side, and still provide the same interface to the corresponding applications, on the other.

Let us consider now the mechanisms described earlier in this section to provide some type of control over potentially conflicting accesses by multiple users to the same application. We presented a scheme in which all interactors that represent equivalent functionality (one interactor from each collection of user-specific interactors) are connected among them through each of their dialog modules, as shown in Fig. 24. The connections, which are actually among the control graphs of the dialogs, impose strictly sequential or mutually exclusive operation among either the complete dialogs or just the exit subgraphs of the dialogs, as shown in Figs. 25, 26 and 27, respectively. Consequently, we want to be able to model a multi-application multi-user interface in such a way that:

1. The model reflects explicitly that it is possible to connect new interactors to the appropriate collection of equivalent interactors already operating in a mutually exclusive fashion;

2. The new interactors are in an idle state at the time of the connection and there presence reflects the fact that a new user has joined the collaborative session in which the interactors are running.

We have to ensure, though, that such a model can still be subject to animated simulation and analyzed for control flow properties, as is the case of a non-reconfigurable model. The general approach we take here is similar to the approach taken for the case of reconfiguring individual dialogs: we show that from the point of view of control flow it does not make a difference having K or $K + 1$ equivalent interactors connected together in a mutual exclusion loop. We observe that the only control flow interaction among all the interactors in the loop is through the single token on either arc `Nxt` or `Mx`, depending on the type of mutual exclusion operation being considered. Therefore, the meaning of a new interactor in the loop is simply that now there is one more element in the loop that can grabb this token (either a complete dialog or just its exit subgraph), become active, and then have the token placed back on `Nxt` or `Mx`.

5.3 Extended Dialog and Multi-User Interface Models

Based on the analyses presented in the previous two sections, we can extend now the models of both the generalized dialog in particular and the multi-user interface itself to include the control graph, data graph, and interpretation necessary to allow the models to change as described in Section 5.2, when the communication between dialogs, for the purpose of controlling multiple user access to the same application, has the structure described in Section 5.1. Essentially, the new models include in both cases a (node, processor) pair associated with those reconfigurations which add elements to the original models, a (node, processor) pair associated with those reconfigurations that remove elements from the original models, and a dataset which stores the information necessary to perform the reconfigurations, that is, the original models.

The new generalized dialog model is shown in Fig. 30. It includes the source and destination extensions to control arc `Nxt` necessary for mutually exclusive operation. Similar generalized models can be provided for the cases in which the dialogs operate in strictly sequential order, as shown in Fig. 25, or when the mutually exclusive operation affects only the dialogs' exit subgraphs, as shown in Fig. 27.

When a token is received on control arc `Add`, a new one-node or two-node subgraph, depending either on the default case for the specific dialog or on data explicitly provided in the new dataset `Config`, is connected to `Sq`, as shown in

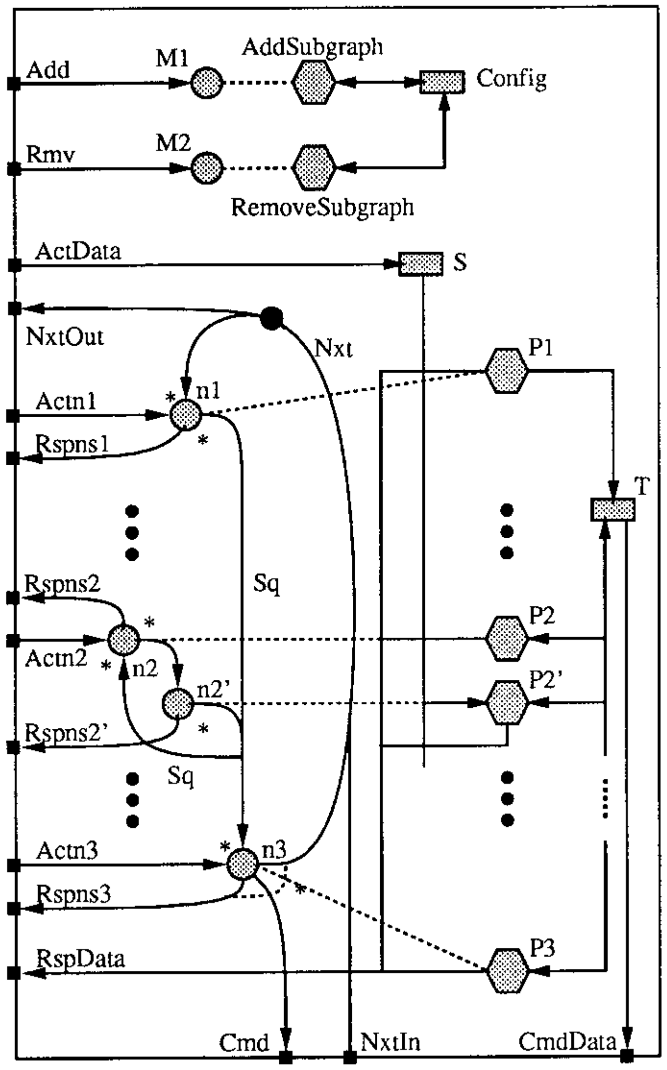


Figure 30: GMB model of a generalized dialog, extended for mutually exclusive operation and dynamic reconfigurability.

Figs. 28 and 29. At the same time, one or two new processors are connected to datasets S and T. The new processors are mapped to the new nodes and their interpretation is found as the current value of S, or can be defined interactively by the users. The outside interconnections, through which the new subgraph receives and sends control signals, have to be defined graphically by the users. If a token is received on control arc Rmv, then again the users, or the current value of dataset S, specify which one-node or two-node subgraph, processors and corresponding connections have to be deleted from the model.

The new multi-user interface model is shown in Fig. 31. We notice that it is no longer necessary to explicitly represent multiple users, because all users participating in a collaborative session, besides the first one, can be incorporated to the session through a reconfiguration process to add users. The reconfiguration process can be initiated by any one of the users currently in the session.

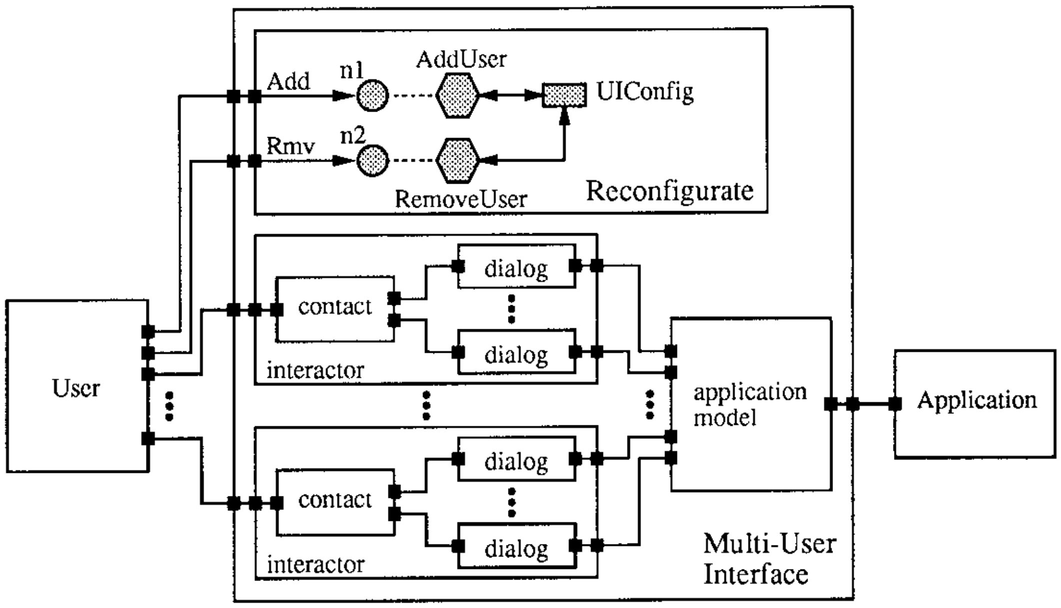


Figure 31: Multi-user interface model extended for dynamic reconfigurability.

Essentially, when a token is received on control arc Add of module Reconfigurate, a new copy of the original collection of interactors is first created, then connected to the application model, and finally connected to the loop of existing copies of the interactors, as shown in Fig. 24. The details of this final connection, that is, if it will have the form presented in Fig. 25, 26 or 27, is defined as part of the interpretation of processor AddUser. For each new interactor created, an actual screen object is requested from the users. The screen object is then associated with the interactor by adding to it the mapping defining its behavior, as explained in Section 4.3. If a token is received on control arc Rmv, the multi-user interface removes one copy of the collection of interactors from the loop and the associated screen objects.

6 The Zoom Tool: An Extended Example of a Multi-User Interface

6.1 The Zoom Tool Concept

In this section, we describe the concept and the multi-user interface of an application to display large graphical designs within the limited space of a workstation screen. The *zoom tool* is an application that gives users control over the specific regions of a *graphic world* which are displayed on a *display* window, as shown in Fig. 32.

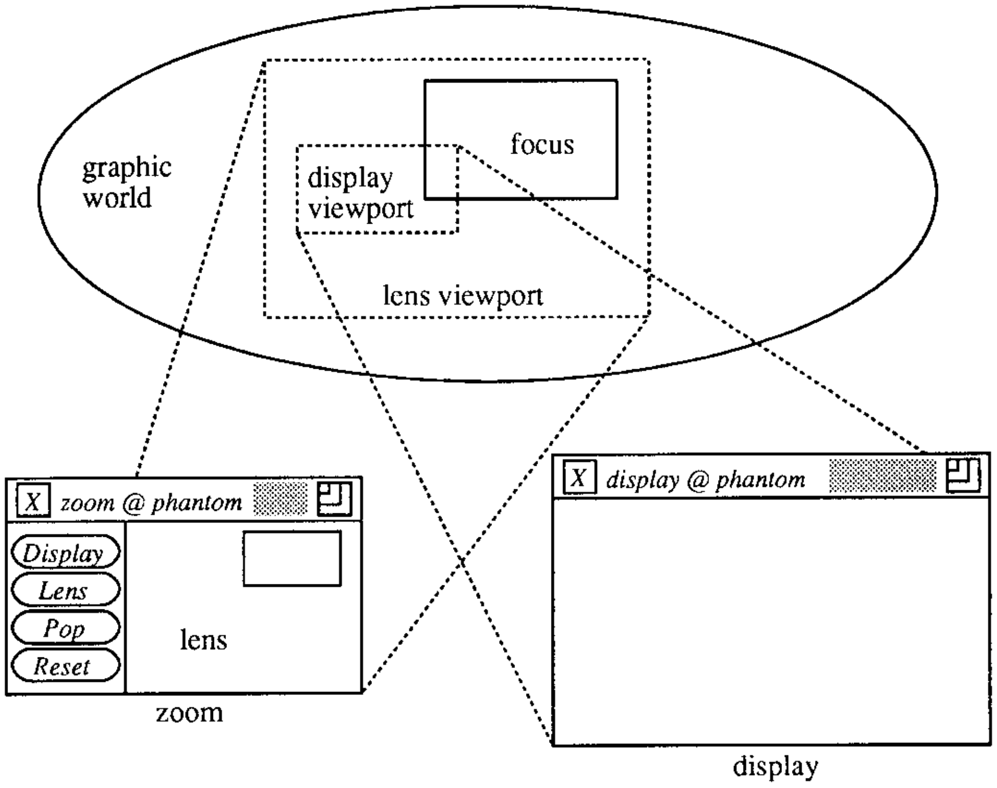


Figure 32: Concept and user interface of the zoom tool.

The *graphic world* is the collection of the graphical representations of all the design objects of interest to the users. The *zoom* window is the representation of the zoom tool application on the screen. It consists of a small window, called the *lens*, which displays the contents of a region of the graphic world, called the *lens viewport*, and a menu with four options: *Display*, *Lens*, *Pop* and *Reset*. The *display* window is a window where the users or designers can produce, display, delete and modify the design objects. It shows the contents of a second region of the graphic world, called the *display viewport*.

On the lens, designers can draw rectangles. A rectangle on the lens defines yet another region of the graphic world called the *focus*. The display viewport will point to and coincide with the focus if the option *Display* in the zoom window's menu is selected. Similarly, the zoom viewport will point to and coincide with the focus if the option *Lens* is selected in the menu. In this case, the previous zoom viewport is saved in a stack; it can be retrieved by selecting the option *Pop*. Finally, users can retrieve the default zoom viewport, and empty the stack, by selecting the option *Reset*.

During a collaborative session, two or more users may have to share the same instance of the zoom tool, while each of them uses a different display window. They share the tool so that they all look at the same display and lens viewports. They use different display windows because these are individually customized (in terms of size, colors and fonts). These users may join and leave the session at different times, and therefore the application has to be able to deal with a variable number of display and zoom windows; in particular, the application has to be able to accomodate a new display window and a new zoom window, or to remove existing display and zoom windows, at any time during the session. It is important to notice that the only type of interaction between a zoom tool application and each of the several display windows consists in the zoom tool changing the display viewport of the windows. Input to the zoom tool produced by the users is done exclusively through the zoom window, i.e., the lens and the menu, as described above.

6.2 Formal Specification of the Zoom Tool's Multi-User Interface

The structural and behavioral specification of the zoom tool's user interface is shown in Fig. 33. It is based on the standard generalized contact presented in Section 3.5 and the standard Button and Box dialogs presented in Section 3.6. The contacts are used to represent the input/output behavior of each screen object involved in the interface: the display window (contact `DisplayCnt`), the lens window (contact `LensCnt`), and each of the four buttons of the menu (contacts `''Lens'' Cnt`, `''Pop'' Cnt`, `''Reset'' Cnt` and `''Dsply'' Cnt`). Because we are interested only in the output behavior of the display window, we are not defining any specific input behavior for it, and therefore we do not include in the specification any dialogs connected to contact `DisplayCnt`. The Box dialog (`BxDlg`) is used to represent the syntax of the interaction supported by the lens. The Button dialog (`BtnDlg`) is used to represent the syntax of the interaction supported by each of the menu buttons. Because the behavioral models of these contacts and dialogs have already been described in detail in Sections 3.5 and 3.6 they are not reproduced in the figure. However, all interconnections with an end point in a dialog module have been labeled with the name of the corresponding

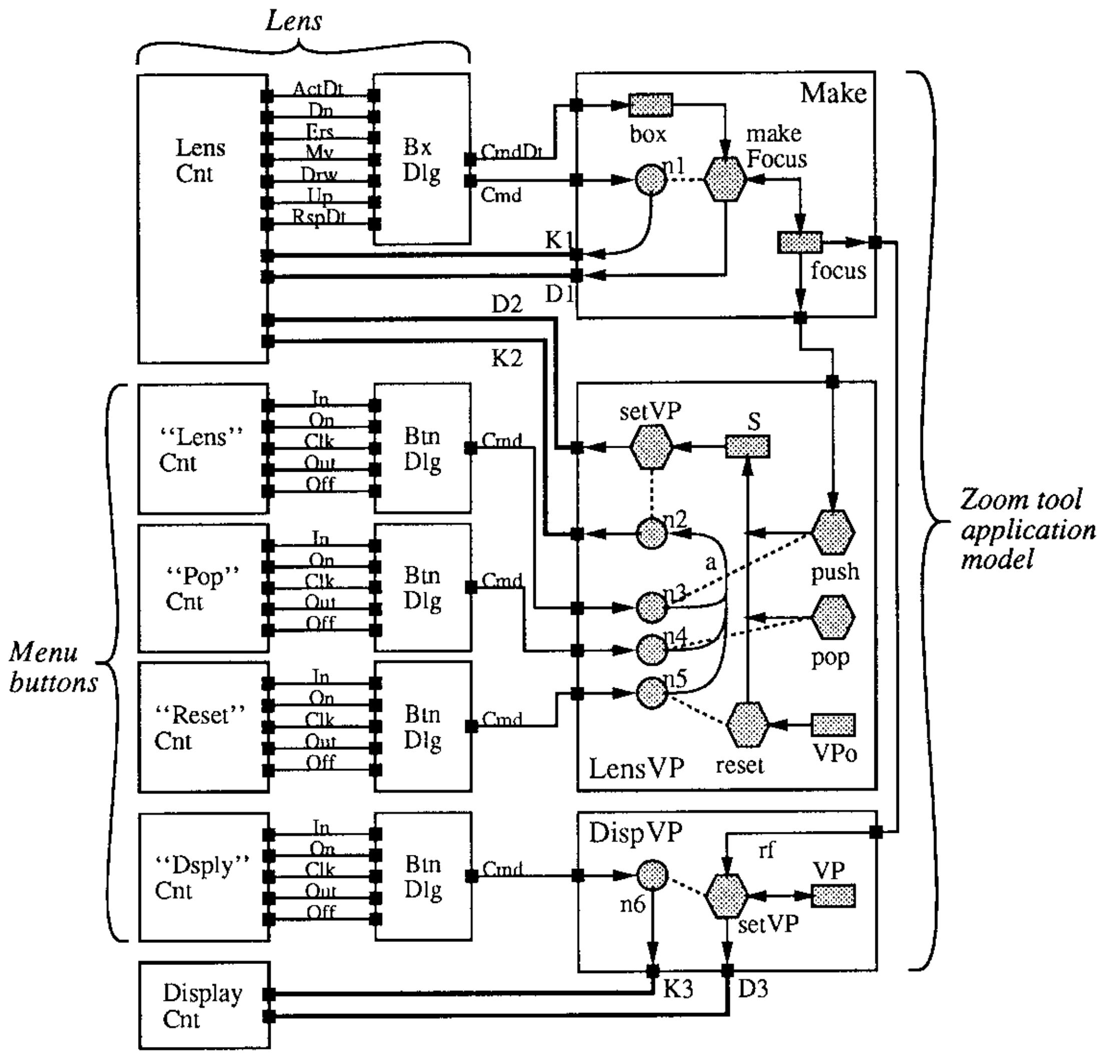


Figure 33: Zoom tool's user interface specification.

control or data arc inside the module.

The zoom tool application model is represented by three modules: `Make`, `LensVP` and `DispVP`. Their interpretation is shown in Fig. 34. `Make` creates a new focus based on data stored in dataset `box` by `BxDlg` through interconnection `CmdDt`. The focus is stored in dataset `focus`, and is available to both modules `LensVP` and `DispVP`. `LensVP` uses the focus to update the lens viewport; `DispVP` uses the focus to update the display viewport. Both updates are performed under user control.

At this point we notice that the specification contains several interconnections linking the application model modules directly to some of the contact modules. We had not seen this situation in previous examples, but there is a good reason

```
   DATASETS
box:   ((x1.y1).(x2.y2))
focus: ((x1.y1).(x2.y2))
S:         Last-in-first-out list of ((x1.y1).(x2,y2))
VPo:   ((x1.y1).(x2.y2))
VP:    ((x1.y1).(x2.y2))
```

```
   PROCESSORS
makeFocus:
   (let ((old (ReadFrom “focus”))
         (new nil))
      (SendThru “D1” old)
      (setf new (make-focus (ReadFrom “box”)))
      (WriteTo “focus” new)
      (SendThru “D1” new))
setVP: {in LensVP}
   (SendThru “D2” (ReadFrom “S”))
push:
   (WriteTo “S” (ReadFrom “focus”))
pop:
   (RemoveTop “S”)
reset:
   (RemoveAll “S”)
   (WriteTo “S” (ReadFrom “VPo”))
setVP: {in DispVP}
   (let ((focus (ReadFrom “focus”)))
      (WriteTo “VP” focus)
      (SendThru “D3” focus))
```

Figure 34: Interpretation for zoom tool application model.

for it in this case. The zoom tool application allows users to define viewports over collections of graphical design objects—the graphic world—, but it does not communicate at all with such collections. The graphic world communicates directly with the lens and display windows for the purpose of displaying its contents to the users. Therefore, the zoom tool simply informs the windows, through the interconnections, about the viewports or regions of the graphic world they are supposed to display, and leaves to the windows themselves the responsibility of handling such information. The interconnections work in pairs, transmitting data (interconnections labeled $\mathtt{D}_i$ in the figure) and control signals (interconnections labeled $\mathtt{K}_i$ in the figure) from the application model to the contacts. The data are first collected and structured by the application model according to information received from the dialogs (therefore from the users) and then sent to the contacts. The control signals are generated by the application model and sent to the contacts right after the data, to activate the contacts' processors which use the data.

Before creating a new focus, module `Make` sends to contact `LensCnt` the coordinates of the current focus, through interconnection `D1`, and the corresponding control signal, through interconnection `K1`, indicating that such focus, now out of date, should be erased from the lens window. The representation of the focus on the lens window is thus replaced by the new rectangle drawn on that window by the user while defining the new focus. Producing the graphical representation of the rectangle is the reponsibility of the Box dialog connected to the window, BxDlg, and does not involve any of the modules of the application model.

Module DispVP works in a very simple way. When a user selects the *Display* option of the menu, a token travels from the ''Dsply'' Cnt contact to the corresponding BtnDlg dialog, finally enabling node n6 and activating processor setVP. While active, setVP first reads the focus coordinates from dataset focus and stores them in dataset VP; then, it sends the focus coordinates through interconnection D2 and the corresponding control signal through interconnection K2 to the display window contact Display Cnt, which is actually responsible for updating the display viewport.

Module LensVP becomes active when a user selects any one of the *Lens*, *Pop* or *Reset* options of the menu. A token then travels from the appropriate contact to the corresponding dialog, finally enabling one of the nodes n3, n4 or n5, and activating the corresponding processor push, pop or reset, respectively. Processor push reads the focus coordinates from dataset focus and pushes them into the stack represented by dataset S; processor pop simply eliminates the set of coordinates currently at the top of S; and processor reset reads the focus coordinates from dataset VPo, which define the default focus and zoom viewport, and writes them into S replacing all sets of coordinates currently stored in S. In all of these three cases, a token is then placed on the multi-source control arc a, enabling node n2 and activating processor setVP. This processor finally sends the focus coordinates at the top of the stack in S and the corresponding control signal to the lens window contact LensCnt. It uses interconnections D3 and K3 for this purpose.

6.3 Operation of the Zoom Tool Prototype

Once the specification described in the previous section has been used to produce a prototype of the zoom tool's user interface and application model, according to the development methodology introduced in Chapter 4, users and designers can execute the prototype by operating mouse-like input devices on CLUE windows. Fig. 35 shows a hardcopy, taken directly from a workstation screen during earlier work on this research, of a window displaying part of the specification of an interface (module BDE-AS) and an application model (module BDE-UI). We felt that the quality of the graphics in the figure was not good enough to be useful in explaining to the reader the process of the execution of the prototype. Therefore, instead of spending a lot of time in improving the graphics, we chose to use a sophisticated drawing application and to reproduce by hand the contents of this window and of the other windows involved in such execution, for inclusion in this document.

Figs. 36, 37, 38 and 39 show different stages during the execution of a zoom tool prototype. The prototype is running on a machine called phantom and its operation involves three windows:

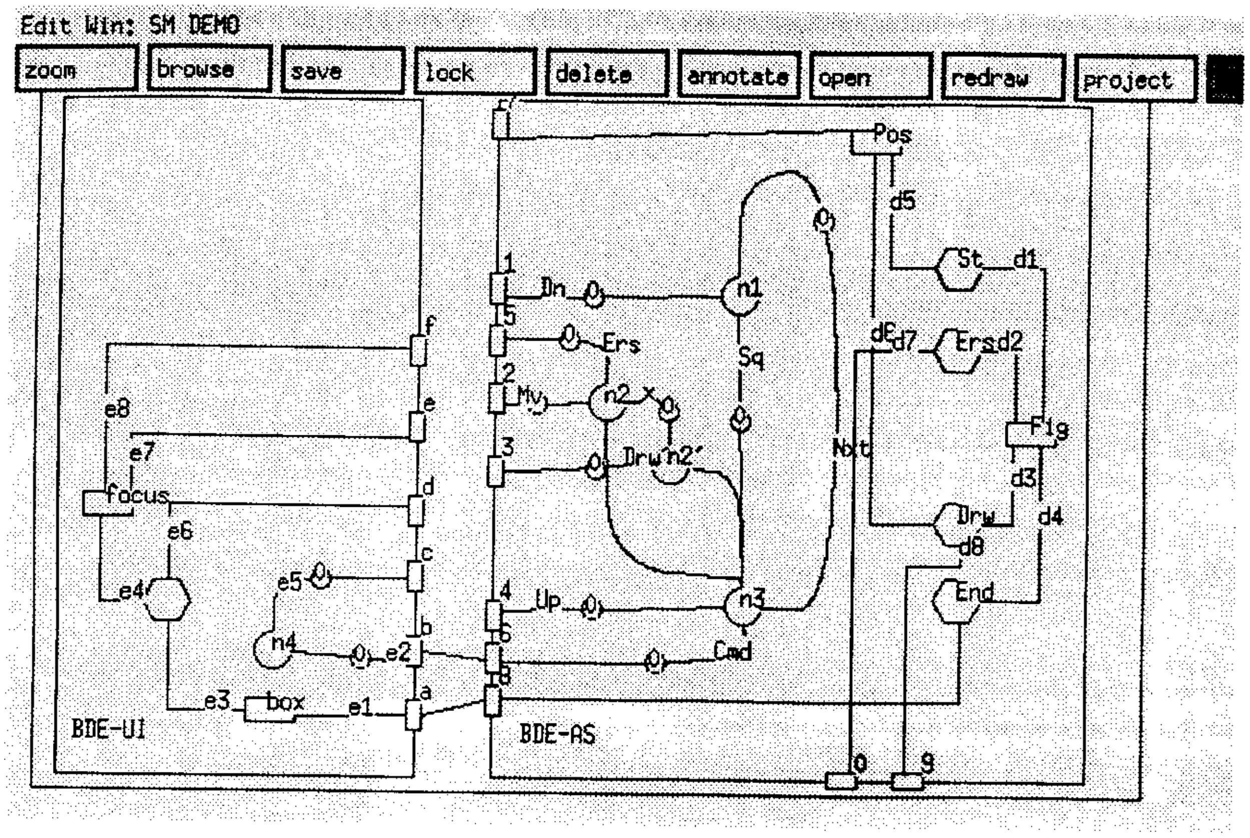


Figure 35: An actual window displaying the specification of the zoom tool's user interface.

- `zoom` implements the zoom window. Each of its four buttons has been linked to the corresponding contact model of the menu buttons in the behavioral specification presented in Fig. 33 of the previous section. Its other subwindow has been linked to the `LensCnt` contact model of the specification. A lens viewport has been previously defined for this window so that it shows a collection of figures stored in the system's graphic world.

- `display` implements the display window. It has been linked to the `DisplayCnt` contact model of the behavioral specification. Initially, no display viewport has been defined for this window and therefore it is not showing any figures.

- `model` shows the state of the behavioral specification, that is, the distribution of tokens on the control arcs, following each user input action. Modules `BxDlg` and `Make` in this figure correspond to modules `BDE-AS` and `BDE-UI`, respectively, in Fig. 35.

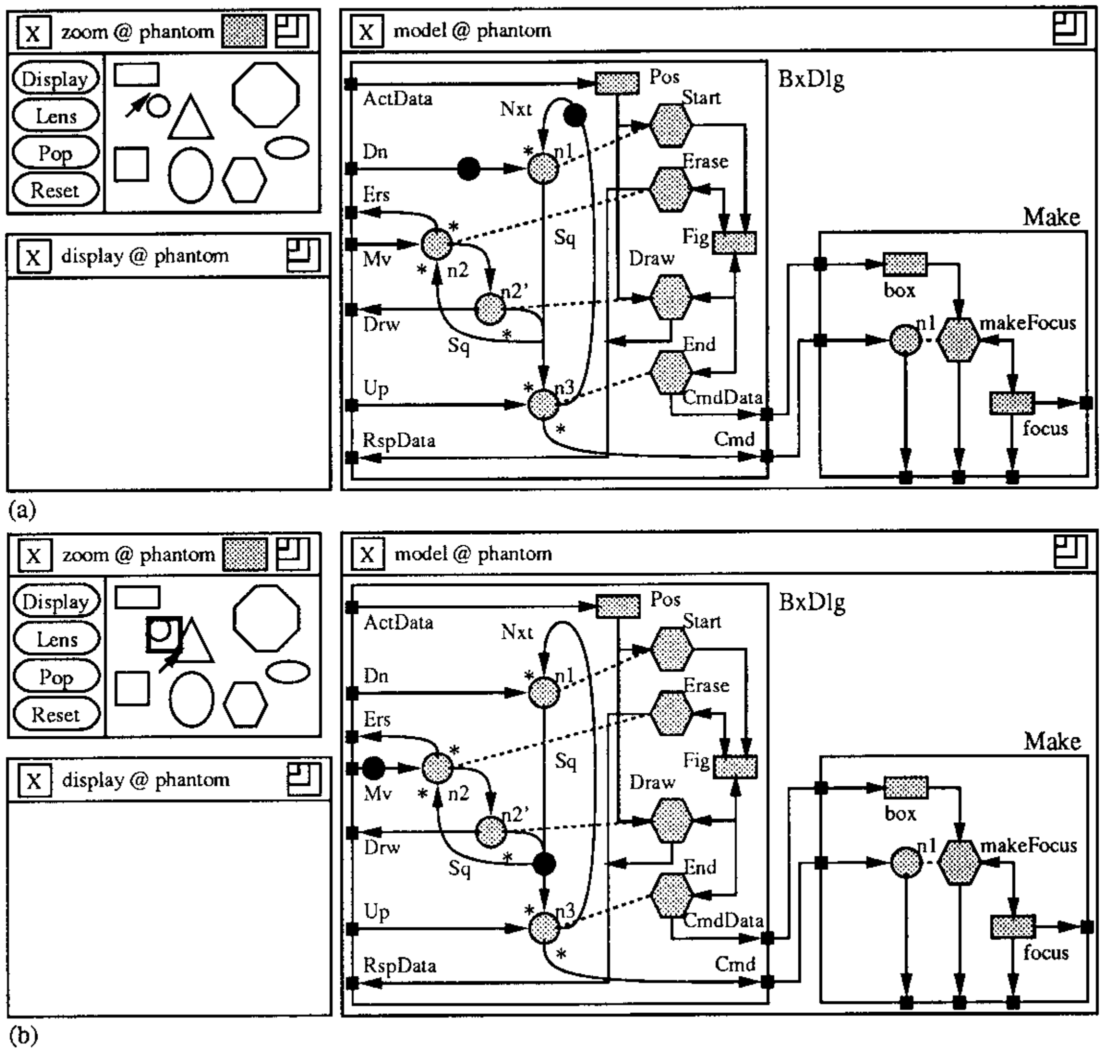


Figure 36: Executing the zoom tool prototype: defining a focus.

6.3.1 Defining a Focus

Figs. 36 and 37 show three major stages involved in the process of defining a focus by drawing a rectangle on the lens of window zoom. The black arrow on this window represents the mouse cursor. Window model is showing only that portion of the behavioral specification which is involved in the definition of a focus: modules BxDlg and Make. In Fig. 36(a) the user has just pressed down the mouse button at the position of the rectangle's top-left corner, thus beginning the drawing of the rectangle. The window detected this action and the corresponding contact model (LensCnt, not shown in the figures) placed a token on control arc Dn in BxDlg. This token will combine with the token initially placed on control arc Nxt to enable node n1 and activate processor Start. When Start finishes, a token will be placed on control arc Sq.

In Fig. 36(b) the user is dragging the mouse towards the position of the rectangle's bottom-right corner. Whenever a motion of the cursor is detected by the window, LensCnt places a token on control arc Mv in BxDlg, as shown in the figure. This token will combine with the token placed on Sq to enable in sequence nodes n2 and n2', activating processors Erase and Draw, respectively. During this sequence, tokens are sent back to LensCnt through control arcs Ers and Drw, thus producing the feedback rubber band effect on the window while the rectangle is being drawn. At the end of the sequence, a token is placed back on Sq, allowing for the repetition of the sequence or the completion of the drawing.

In Fig. 37 the user has released the mouse button at the position of the rectangle's bottom-right corner, thus finishing the drawing of the rectangle. The window detected the action, LensCnt and BxDlg processed it (when node n3 activated processor End), and finally a token was placed on control arc Cmd in module Make of the zoom tool's application model. BxDlg is again in its initial state. The token on Cmd will enable node n1 in Make and activate processor makeFocus, which will create the semantic focus and store it in dataset focus.

6.3.2 Setting the Display Viewport

Figs. 38 and 39 show three major stages involved in the process of setting the display viewport to the current focus on window zoom, by choosing the option *Display* on zoom's menu. Again, window model is showing only that portion of the behavioral specification which is involved in setting the display viewport: modules BtnDlg, DispVP and DisplayCnt. In Fig. 38(a) the user has just moved the mouse cursor inside the menu button labeled *Display*. The button detected this action and the corresponding contact model (''Dsply'' Cnt, not shown in the figures) placed a token on control arc In in BtnDlg. This token will combine with the token initially placed on control arc Nxt to enable node n1 and place tokens on control arcs Sq and On. The token on On will go back to ''Dsply'' Cnt causing the menu button to highlight. The token on Sq will allow the menu

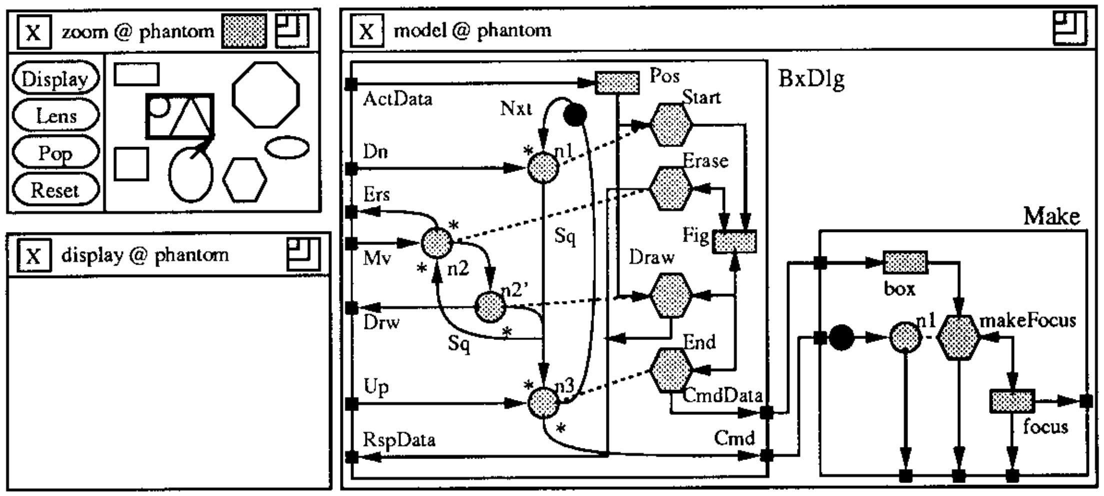


Figure 37: Executing the zoom tool prototype: a focus has been defined.

selection procedure to proceed.

In Fig. 38(b) the user has made the selection by clicking the mouse button while pointing at the menu button, which now is highlighted. The menu button detected the action, ``Dsply'' Cnt and BtnDlg processed it, and finally a token was placed on control arc Cmd in module DispVP of the zoom tool's application model. The token on Cmd will enable node n6 and activate processor setVP. While active, setVP will read the coordinates of the current focus from dataset focus in module Make (through data arc rf, as shown in Fig. 33), store them in dataset VP, and then send them to module DisplayCnt through interconnection D3. In DisplayCnt these values will be stored in dataset viewport. Finally, a token will be sent to DisplayCnt through interconnection K3, enabling node n4 and activating processor set. This processor will actually set the display viewport of window display, producing the image shown in the lower left corner of Fig. 39.

6.4 Reconfiguring the Multi-User Interface Model

Section 5.2 presented the fundamental aspects related to the dynamic reconfiguration of a multi-user interface. In particular, section 5.2.2 deals with the reconfiguration of a collection of interacting dialogs as a consequence of the participation of a new user in an ongoing collaborative session. We said then that in order to keep an application independent of the details of its multi-user interface, the corresponding application model modules are not replicated as are the dialog modules. In this way, an application always communicates with only one instance of each application model module. However, each application model module that communicates with a dialog or a contact, has to change its control and data graphs as the dialog or contact is replicated, to be able to communicate with each copy

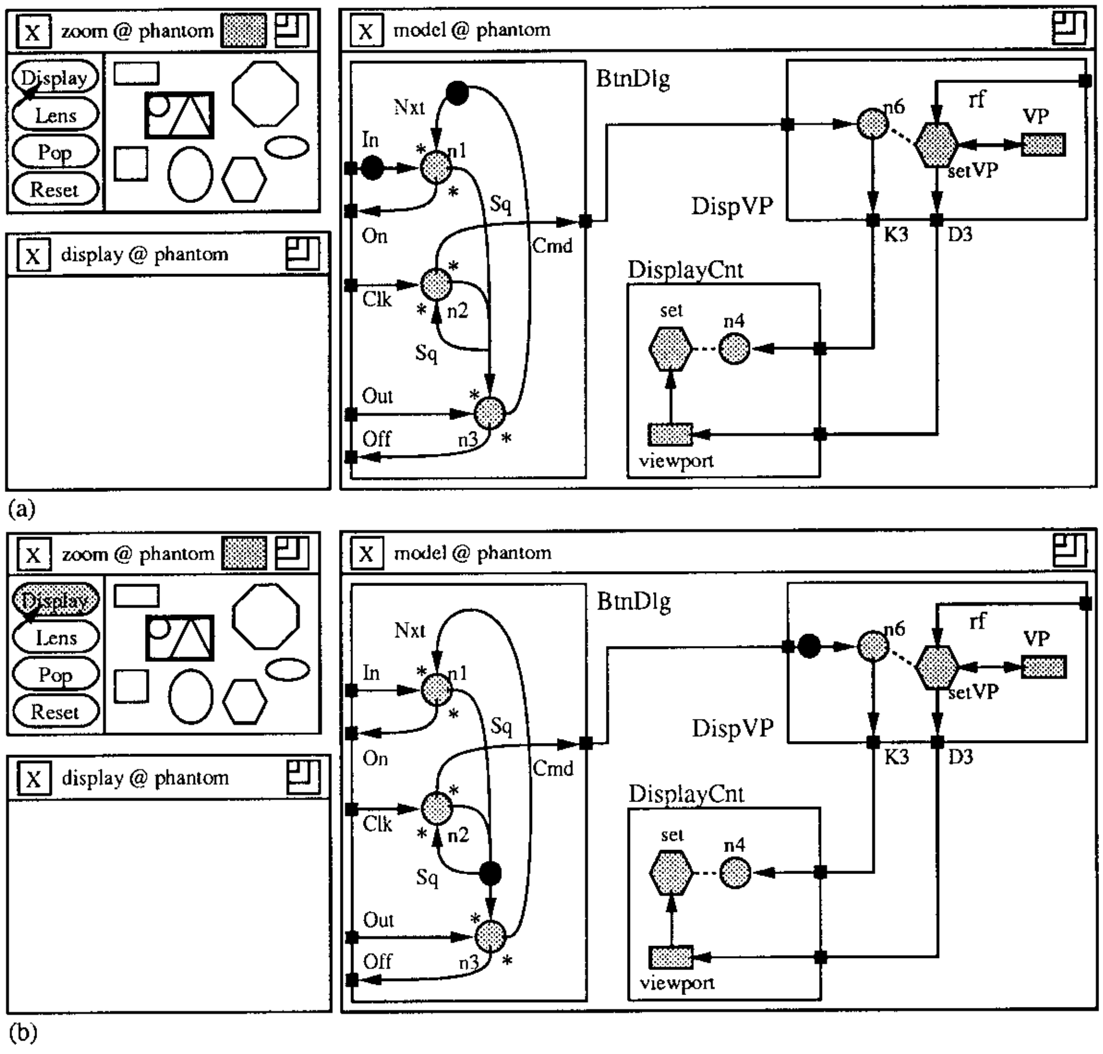


Figure 38: Executing the zoom tool prototype: setting the display viewport.

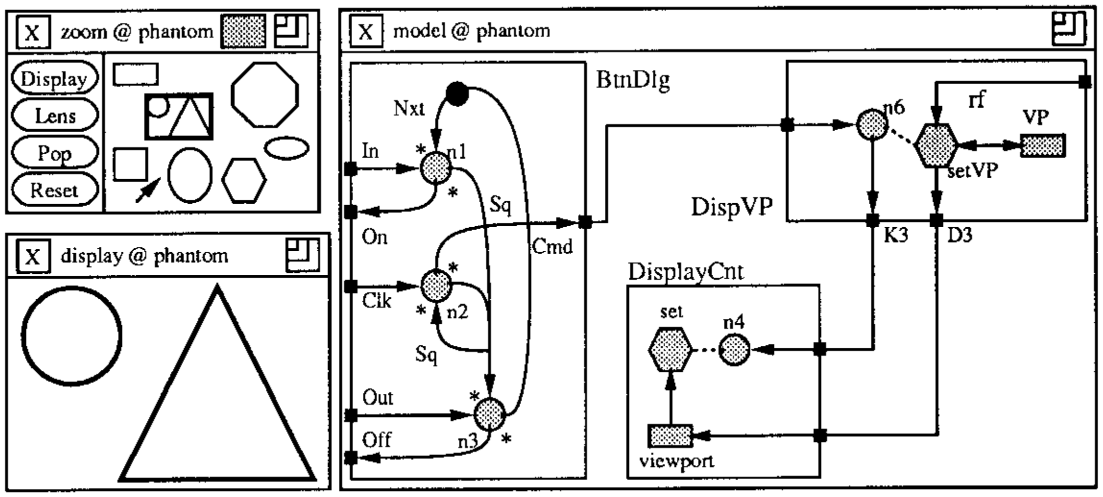


Figure 39: Executing the zoom tool prototype: the display viewport has been set.

of the dialog or contact. In the case of the zoom tool application model, each of the three modules `Make`, `LensVP` and `DispVP` communicates directly with both a contact and a dialog, as shown in Fig. 33. When a new user joins an ongoing session the following steps take place:

1. A new copy of each original interactor (that is, each original contact and the dialogs connected to it) is produced. Each new copy is connected to the already existing copies of the same interactor according to one of the possible connection patterns described in Section 5.1, with the purpose of guaranteeing the mutually exclusive or strictly sequential operation of the dialogs. The resulting configuration has a general structure similar to that shown in Fig. 24. In particular, the multiple copies of each original dialog are interconnected in patterns similar to those shown in Figs. 25, 26 or 27.

2. For each new copy of an interactor an actual screen object is requested. When the screen object is provided, it is mapped to the interactor, following the process described in Section 4.3, to make it sensitive to the user-generated input actions and the application-generated responses specified in the interactor's contact.

3. Finally, for each new copy of an interactor, the application model modules connected to the original interactor are updated to be able to communicate with the new interactor's contact and/or dialogs. This step is explained in more detail below.

Fig. 40 shows the configuration of part of the zoom tool's multi-user interface after a second user has joined a session originally involving only one user. The

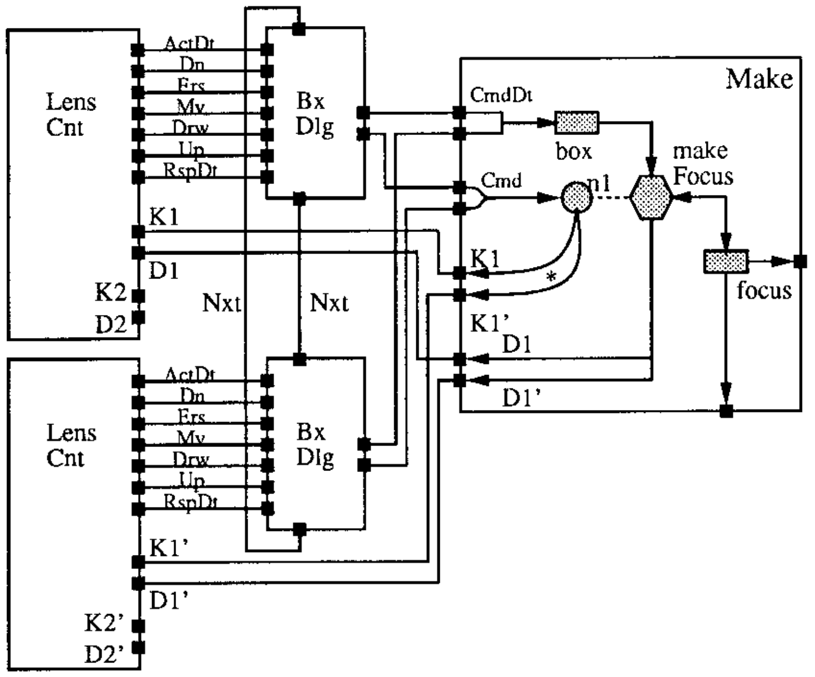


Figure 40: Reconfiguring the model to let a second user join the session.

figure shows the updated module Make in the application model, and the two copies of the interactor representing the zoom window lens, each containing a contact (LensCnt) and a Box dialog (BxDlg). The two dialogs are interconnected through extensions of their Nxt control arcs, for mutually exclusive operation.

The updated module Make has to be able to receive input control signals and data defining new focuses from both interactors. It also has to be able to send output control signals and data indicating rectangles to be drawn or erased to both interactors. To achieve this behavior, Make augments or replicates its input and output control and data arcs. In this particular case, input to Make comes from the interactors' dialogs, and output from Make goes to the interactors' contacts.

The output behavior can only be achieved by replicating the output control arc K1 and the output data arc D1. In the figure, the new arcs are named K1' and D1', respectively. Data arcs D1 and D1' have a common tail connecting them to processor makeFocus, representing the fact that when the processor writes data through one of them it is also writing the same data through the other. Similarly, control arcs K1 and K1' are connected by an *AND* (*) output logic expression, meaning that when makeFocus finishes its execution, node n1 places one output token on each arc. Upon reception of both the data and the control signals, each LensCnt contact proceeds to update the image on the corresponding screen object, erasing the current rectangle, which represents the old focus, and drawing a new rectangle that represents the new focus.

The input behavior, on the other hand, is achieved by augmenting the input control arc Cmd and the input data arc CmdDt, each with one more source attached to a new socket, which in turn is connected to the new dialog. Data, control signal pairs arrive from the dialogs. The data is stored in dataset box, which may already contain unprocessed data from other dialogs. It is important to remember that only one dialog may attempt to write into box at any given time, because of their mutual exclusion operation. The control signal is placed on arc Cmd in the form of a token; Cmd may also contain other tokens corresponding to the unprocessed data in box. When the token machine examines node n1, it will remove one of the tokens and activate processor makeFocus. This processor will read one set of data from box and create the corresponding semantic focus. Defining box to operate as a first-in-first-out queue, one of the possible dataset behaviors, guarantees that the focuses will be created in the order in which the corresponding data were received by box.

A similar updating process will take place in each of the other application model modules, LensVP and DispVP, when a new user joins an ongoing collaborative session. In general, each original, single-user version output control arc and output data arc, carrying information from the application model to the interactors, will be replicated; and during the execution of the corresponding nodes and processors, identical control signals and sets of data will be sent through each replica. On the other hand, each input control arc and input data arc, carrying information from the interactors to the application model, will be augmented with one more source connected to a new socket. Data received through different sources of the same data arc will be stored in the corresponding dataset queue to be processed later.

Fig. 41 shows the workstation screens of machines phantom and ghost after the user at ghost has been incorporated into the session that initially included only the user at phantom. We present the drawing application-generated reproductions of the contents of the screens, instead of actual screen hardcopies, for reasons similar to those mentioned at the beginning of Section 6.3. It is important to notice that while the contents of the different windows is the same in both machines, the distribution of the windows on the screen and the geometry of the windows can be completely different. Furthermore, as the figure shows by the position of the arrow representing the mouse cursor, the two users can be working on different windows at any given time: while the user of phantom is working on window zoom, the user of ghost is working on window model. To deal with such possible ambiguities, the UCLA Collaborative Design Environment, coSARA [Mujica, 1991], requires that every user be able to see what other users are working on.

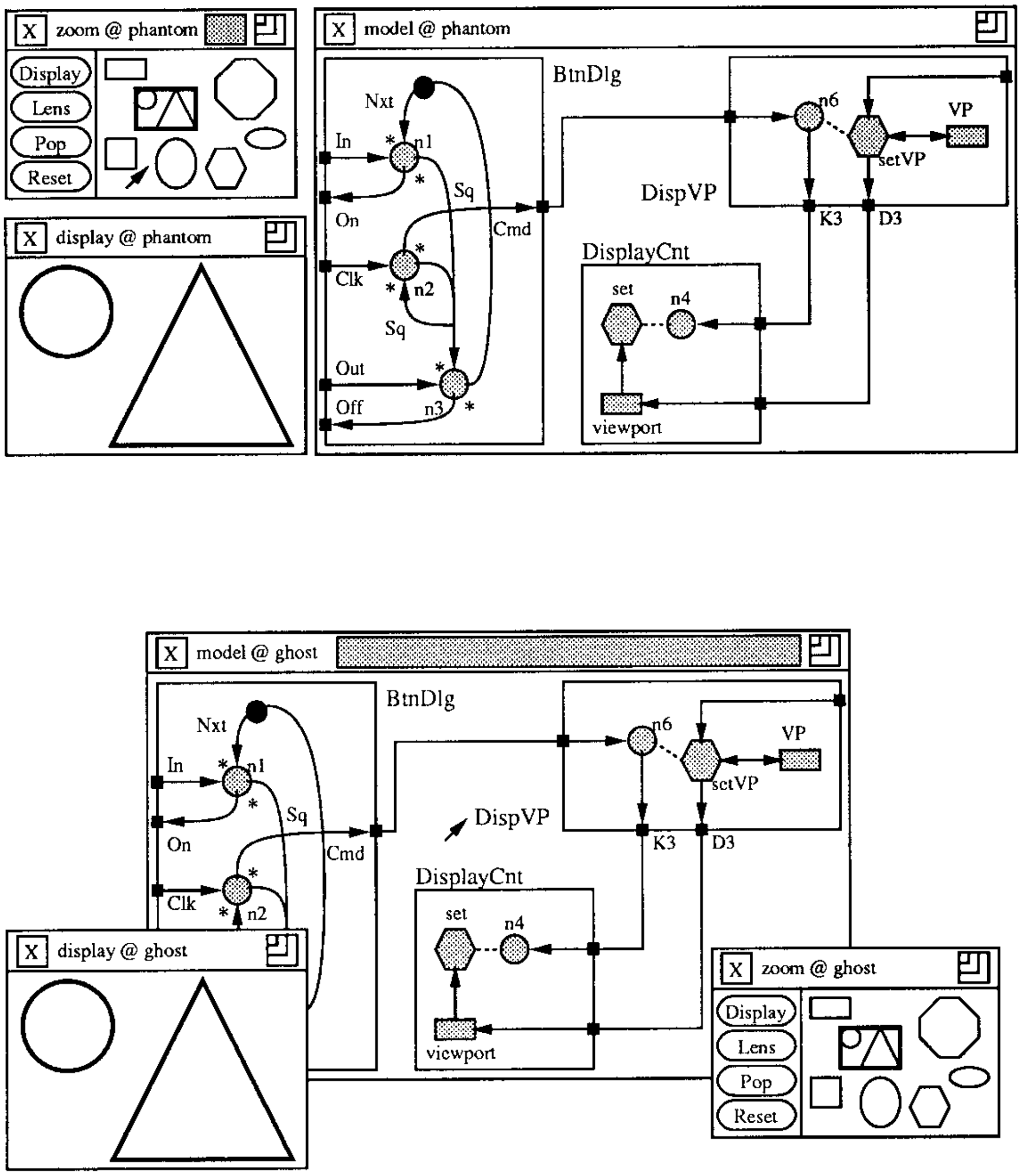


Figure 41: Two users concurrently sharing the zoom tool.

7 Conclusions and Future Work

In this work, we have presented solutions to three important problems in the field of multi-application multi-user interfaces: modeling, specification, and development of executable prototypes. We have characterized these interfaces as graphical, interactive, collaborative, based on the direct-manipulation style of interaction, and dynamically reconfigurable. Our solutions to the problems consist of:

1. An operational model of the interfaces;
2. An object-oriented specification and development methodology to build executable prototypes;
3. Application-independent mechanisms for the specification of coordination among collaborating users concurrently sharing an application; and
4. Application-independent mechanisms for the specification of interfaces whose structure and behavior can change in a structured, controlled way during their operation.

We presented a formal operational model of multi-application multi-user interfaces, which combines, extends and refines the Direct-Manipulation model and the Reference model. Our model includes detailed structural and behavioral specification of:

1. The screen objects with which the users interact directly. These objects are represented in more detail than in any other existing model. Still, the representations are completely independent of any particular implementation. A screen object model, called an interactor, includes separate, communicating models for the input and output aspects of the interface, and for the expression of the syntax of user-application dialogs.
2. The application model supporting the shared data model of interfaces based on the direct-manipulation style of interaction. This component facilitates the sharing of data between the interface and the application. It is defined as an abstract, object-oriented representation of the functionality of the application as seen from the interface's point of view.
3. The communication of control and data between a user-specific collection of screen objects and an application model, and the communication of control and data among equivalent screen objects supporting the interaction of different users with the same application.

4. A structured, hierarchical test environment for interface specifications. The environment allows designers to systematically test the specification of individual syntactic components (dialogs) in a screen object, of the collection of dialogs of a screen object, of the collection of screen objects in a user-specific interface, and of the complete multi-user interface.

We presented an object-oriented specification and development methodology to build executable prototypes of multi-user interfaces. The prototypes can be executed by designers and end users by operating real input devices on real screen objects. The specification method uses the OREL modeling language to describe the organization of the object classes composing an interface (class diagram). Then, the SM and GMB modeling languages are used to describe the behavior of these classes; in particular, the structure and behavior of the screen objects are represented in the same terms as in the operational model. The structure and behavior of the application model component of the interface is also described in terms of SM and GMB, following an object-oriented approach in which modules represent classes and submodules represent operations on classes.

The development method uses an extensible library which contains: (1) software modules to assign interpretation to the behavioral models; (2) class and method definitions produced by the OREL compiler from user interface class diagrams, to create and manipulate class instances of the interface and its components (the application model and the interactors); and (3) CLUE contacts to create real screen objects. Instances of the application model and of the interactors can be linked to the corresponding behavioral models. Then, the screen objects can be linked to the corresponding interactor instances, thus defining the behavior of the screen objects and allowing designers and users to execute the specifications through the screen objects.

We presented a set of application-independent mechanisms to specify coordination among the activities of multiple users concurrently sharing an application. The mechanisms are based on mutually exclusive or strictly sequential operation of equivalent screen objects associated with different users. They have been easily encapsulated in the models of the screen objects, freeing designers from having to explicitly specify them. And they support several different types of coordination, giving designers some freedom in chosing the most appropriate strategy for each particular interface.

Finally, we extended the SM and GMB modeling languages to support the specification of interfaces which can reconfigure themselves during the operation of a collaborative session, as a consequence of new users joining the session and new applications becoming available. The extensions are based on the existence of regular patterns of control behavior, both inside individual screen objects and across equivalent screen objects communicating different users with the same application.

Two important topics for future research in the field of multi-application multi-user interfaces have appeared as a consequence of this work. First, the specification methodology described in this document involves several different specification and modeling languages which in some cases have not been designed to work with one another. Although it has been possible to combine them fruitfully, our methodology would benefit enormously from a more unified approach. We propose to accomplish this by extending the OREL language in order to support true modularization through information hiding, specially with respect to composite classes, and by extending the SM and GMB languages in order to support features of object-oriented specifications. The extensions should move the languages closer to each other. Second, the environment in which the multi-user interface prototypes produced with our development methodology have been tested consists of four Sun workstations connected by a local area network. In this environment, prototypes are executed by a centralized token machine interpreter which distributes the results of the interpretations to all the workstations. Supporting remote collaboration, however, through a distributed token machine interpreter, would greatly improve the usefulness of our system.

A CLOS code for user interface OREL model

```
;;; -*- Mode: Lisp; Package: OREL -*-

(in-package 'OREL :use '(pcl orel object-world lisp))

;;  Uncomment if OREL-BASIC is not already loaded
;; (load "/x/CDE/orel/orel-basic")

(pcl:defclass BLOCK (SIMPLE-OBJECT)
  ((BtmRight :initform nil :accessor BLOCK-BtmRight)
   (TopLeft :initform nil :accessor BLOCK-TopLeft)))

(defun MAKE-BLOCK (&optional (name "") )
  (make-instance 'BLOCK :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass CALL (SIMPLE-OBJECT)
  ((CALLS :initform nil :accessor CALL-CALLS)))

(defun MAKE-CALL (&optional (name "") )
  (make-instance 'CALL :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass COMMAND (SIMPLE-OBJECT)
  ((COMMANDS :initform nil :accessor COMMAND-COMMANDS)))

(defun MAKE-COMMAND (&optional (name "") )
  (make-instance 'COMMAND :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass DIALOG (SIMPLE-OBJECT)
  ((ACTIONS :initform nil :accessor DIALOG-ACTIONS)
   (RESPONSES :initform nil :accessor DIALOG-RESPONSES)))

(defun MAKE-DIALOG (&optional (name "") )
  (make-instance 'DIALOG :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass ACTION (SIMPLE-OBJECT)
  ((ACTIONS :initform nil :accessor ACTION-ACTIONS)))
```

```
(defun MAKE-ACTION (&optional (name "") )
  (make-instance 'ACTION :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass RESPONSE (SIMPLE-OBJECT)
  ((RESPONSES :initform nil :accessor RESPONSE-RESPONSES)))

(defun MAKE-RESPONSE (&optional (name "") )
  (make-instance 'RESPONSE :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass CONTACT (SIMPLE-OBJECT)
  ((ACTIONS :initform nil :accessor CONTACT-ACTIONS)
   (RESPONSES :initform nil :accessor CONTACT-RESPONSES)))

(defun MAKE-CONTACT (&optional (name "") )
  (make-instance 'CONTACT :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass TOOL (COMPOSITE-OBJECT)
  ((CALLS :initform nil :accessor TOOL-CALLS)
   (COMMANDS :initform nil :accessor TOOL-COMMANDS)
   (BLOCK :initform nil :accessor comp-BLOCK)))

(defun MAKE-TOOL (&optional (name "") )
  (make-instance 'TOOL :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass INTERACTOR (COMPOSITE-OBJECT)
  ((CALLS :initform nil :accessor INTERACTOR-CALLS)
   (COMMANDS :initform nil :accessor INTERACTOR-COMMANDS)
   (RESPONSES :initform nil :accessor comp-RESPONSES)
   (ACTIONS :initform nil :accessor comp-ACTIONS)
   (DIALOG :initform nil :accessor comp-DIALOG)
   (ACTION :initform nil :accessor comp-ACTION)
   (RESPONSE :initform nil :accessor comp-RESPONSE)
   (CONTACT :initform nil :accessor comp-CONTACT)))

(defun MAKE-INTERACTOR (&optional (name "") )
  (make-instance 'INTERACTOR :storable-name name
                 :storable-id (object-world::unique-sym)))
```

```
(pcl:defclass UI (COMPOSITE-OBJECT)
  ((TOOL :initform nil :accessor comp-TOOL)
   (CALLS :initform nil :accessor comp-CALLS)
   (COMMANDS :initform nil :accessor comp-COMMANDS)
   (CALL :initform nil :accessor comp-CALL)
   (COMMAND :initform nil :accessor comp-COMMAND)
   (INTERACTOR :initform nil :accessor comp-INTERACTOR)))

(defun MAKE-UI (&optional (name "") )
  (make-instance 'UI :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass CALLS (RELATION-OBJECT)
  ((TOOL-many-p :reader TOOL-many-p :initform T)
   (TOOL-ordered-p :reader TOOL-ordered-p :initform NIL)
   (TOOL-directed-p :reader TOOL-directed-p :initform NIL)
   (TOOL-map-type :reader TOOL-map-type :initform :PARTIAL)
   (CALL-many-p :reader CALL-many-p :initform T)
   (CALL-ordered-p :reader CALL-ordered-p :initform NIL)
   (CALL-directed-p :reader CALL-directed-p :initform NIL)
   (CALL-map-type :reader CALL-map-type :initform :PARTIAL)
   (INTERACTOR-many-p :reader INTERACTOR-many-p :initform T)
   (INTERACTOR-ordered-p :reader INTERACTOR-ordered-p :initform NIL)
   (INTERACTOR-directed-p :reader INTERACTOR-directed-p
:initform NIL)
   (INTERACTOR-map-type :reader INTERACTOR-map-type
:initform :PARTIAL)
   (orel::tuple-list :initform nil :accessor orel::tuple-list)))


(defun MAKE-CALLS (&optional (name "") )
  (make-instance 'CALLS :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass CALLS-TUPLE (TUPLE-OBJECT)
  ((TOOL :accessor TOOL :initarg :TOOL)
   (CALL :accessor CALL :initarg :CALL)
   (INTERACTOR :accessor INTERACTOR :initarg :INTERACTOR)))


(defun MAKE-CALLS-TUPLE (&optional (name "")
&key INTERACTOR CALL TOOL )
```

```
  (make-instance 'CALLS-TUPLE :storable-name name
:storable-id (object-world::unique-sym)
:INTERACTOR INTERACTOR :CALL CALL :TOOL TOOL))

(pcl:defclass COMMANDS (RELATION-OBJECT)
  ((TOOL-many-p :reader TOOL-many-p :initform T)
   (TOOL-ordered-p :reader TOOL-ordered-p :initform NIL)
   (TOOL-directed-p :reader TOOL-directed-p :initform NIL)
   (TOOL-map-type :reader TOOL-map-type :initform :PARTIAL)
   (COMMAND-many-p :reader COMMAND-many-p :initform T)
   (COMMAND-ordered-p :reader COMMAND-ordered-p :initform NIL)
   (COMMAND-directed-p :reader COMMAND-directed-p :initform NIL)
   (COMMAND-map-type :reader COMMAND-map-type :initform :PARTIAL)
   (INTERACTOR-many-p :reader INTERACTOR-many-p :initform T)
   (INTERACTOR-ordered-p :reader INTERACTOR-ordered-p :initform NIL)
   (INTERACTOR-directed-p :reader INTERACTOR-directed-p
:initform NIL)
   (INTERACTOR-map-type :reader INTERACTOR-map-type
:initform :PARTIAL)
   (orel::tuple-list :initform nil :accessor orel::tuple-list)))


(defun MAKE-COMMANDS (&optional (name "") )
  (make-instance 'COMMANDS :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass COMMANDS-TUPLE (TUPLE-OBJECT)
  ((TOOL :accessor TOOL :initarg :TOOL)
   (COMMAND :accessor COMMAND :initarg :COMMAND)
   (INTERACTOR :accessor INTERACTOR :initarg :INTERACTOR)))


(defun MAKE-COMMANDS-TUPLE
(&optional (name "") &key INTERACTOR COMMAND TOOL )
  (make-instance 'COMMANDS-TUPLE :storable-name name
                 :storable-id (object-world::unique-sym)
:INTERACTOR INTERACTOR :COMMAND COMMAND :TOOL TOOL))

(pcl:defclass RESPONSES (RELATION-OBJECT)
  ((CONTACT-many-p :reader CONTACT-many-p :initform T)
   (CONTACT-ordered-p :reader CONTACT-ordered-p :initform NIL)
   (CONTACT-directed-p :reader CONTACT-directed-p :initform NIL)
   (CONTACT-map-type :reader CONTACT-map-type :initform :PARTIAL)
   (DIALOG-many-p :reader DIALOG-many-p :initform T)
   (DIALOG-ordered-p :reader DIALOG-ordered-p :initform NIL)
   (DIALOG-directed-p :reader DIALOG-directed-p :initform NIL)
   (DIALOG-map-type :reader DIALOG-map-type :initform :PARTIAL)
   (RESPONSE-many-p :reader RESPONSE-many-p :initform T)
   (RESPONSE-ordered-p :reader RESPONSE-ordered-p :initform NIL)
   (RESPONSE-directed-p :reader RESPONSE-directed-p :initform NIL)
   (RESPONSE-map-type :reader RESPONSE-map-type :initform :PARTIAL)
   (orel::tuple-list :initform nil :accessor orel::tuple-list)))

(defun MAKE-RESPONSES (&optional (name "") )
  (make-instance 'RESPONSES :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass RESPONSES-TUPLE (TUPLE-OBJECT)
  ((CONTACT :accessor CONTACT :initarg :CONTACT)
   (DIALOG :accessor DIALOG :initarg :DIALOG)
   (RESPONSE :accessor RESPONSE :initarg :RESPONSE)))

(defun MAKE-RESPONSES-TUPLE (&optional (name "")
&key RESPONSE DIALOG CONTACT )
  (make-instance 'RESPONSES-TUPLE :storable-name name
                 :storable-id (object-world::unique-sym)
:RESPONSE RESPONSE :DIALOG DIALOG :CONTACT CONTACT))

(pcl:defclass ACTIONS (RELATION-OBJECT)
  ((DIALOG-many-p :reader DIALOG-many-p :initform T)
   (DIALOG-ordered-p :reader DIALOG-ordered-p :initform NIL)
   (DIALOG-directed-p :reader DIALOG-directed-p :initform NIL)
   (DIALOG-map-type :reader DIALOG-map-type :initform :PARTIAL)
   (ACTION-many-p :reader ACTION-many-p :initform T)
   (ACTION-ordered-p :reader ACTION-ordered-p :initform NIL)
   (ACTION-directed-p :reader ACTION-directed-p :initform NIL)
   (ACTION-map-type :reader ACTION-map-type :initform :PARTIAL)
   (CONTACT-many-p :reader CONTACT-many-p :initform T)
   (CONTACT-ordered-p :reader CONTACT-ordered-p :initform NIL)
   (CONTACT-directed-p :reader CONTACT-directed-p :initform NIL)
   (CONTACT-map-type :reader CONTACT-map-type :initform :PARTIAL)
   (orel::tuple-list :initform nil :accessor orel::tuple-list)))
```

```
(defun MAKE-ACTIONS (&optional (name "") )
  (make-instance 'ACTIONS :storable-name name
                 :storable-id (object-world::unique-sym)))

(pcl:defclass ACTIONS-TUPLE (TUPLE-OBJECT)
  ((DIALOG :accessor DIALOG :initarg :DIALOG)
   (ACTION :accessor ACTION :initarg :ACTION)
   (CONTACT :accessor CONTACT :initarg :CONTACT)))

(defun MAKE-ACTIONS-TUPLE (&optional (name "")
&key CONTACT ACTION DIALOG )
  (make-instance 'ACTIONS-TUPLE :storable-name name
                 :storable-id (object-world::unique-sym)
:CONTACT CONTACT :ACTION ACTION :DIALOG DIALOG))

(defmethod ADD-COMPONENT ((c TOOL) (o BLOCK) &optional role-name)
  (push o (comp-BLOCK c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c TOOL) (o BLOCK))
  (setf (comp-BLOCK c) (delete o (comp-BLOCK c) :test #'equal)))

(defmethod COMPONENTS-OF ((o TOOL))
  (list (comp-BLOCK o)))

(defmethod ADD-COMPONENT ((c INTERACTOR) (o RESPONSES)
&optional role-name)
  (push o (comp-RESPONSES c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c INTERACTOR) (o RESPONSES))
  (setf (comp-RESPONSES c) (delete o (comp-RESPONSES c)
:test #'equal)))

(defmethod ADD-COMPONENT ((c INTERACTOR) (o ACTIONS)
&optional role-name)
  (push o (comp-ACTIONS c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c INTERACTOR) (o ACTIONS))
  (setf (comp-ACTIONS c) (delete o (comp-ACTIONS c) :test #'equal)))
```

```
(defmethod ADD-COMPONENT ((c INTERACTOR) (o DIALOG)
&optional role-name)
  (push o (comp-DIALOG c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c INTERACTOR) (o DIALOG))
  (setf (comp-DIALOG c) (delete o (comp-DIALOG c) :test #'equal)))

(defmethod ADD-COMPONENT ((c INTERACTOR) (o ACTION)
&optional role-name)
  (push o (comp-ACTION c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c INTERACTOR) (o ACTION))
  (setf (comp-ACTION c) (delete o (comp-ACTION c) :test #'equal)))

(defmethod ADD-COMPONENT ((c INTERACTOR) (o RESPONSE)
&optional role-name)
  (push o (comp-RESPONSE c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c INTERACTOR) (o RESPONSE))
  (setf (comp-RESPONSE c) (delete o (comp-RESPONSE c)
:test #'equal)))

(defmethod ADD-COMPONENT ((c INTERACTOR) (o CONTACT)
&optional role-name)
  (push o (comp-CONTACT c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c INTERACTOR) (o CONTACT))
  (setf (comp-CONTACT c) (delete o (comp-CONTACT c) :test #'equal)))

(defmethod COMPONENTS-OF ((o INTERACTOR))
  (list (comp-RESPONSES o)
        (comp-ACTIONS o)
        (comp-DIALOG o)
        (comp-ACTION o)
        (comp-RESPONSE o)
        (comp-CONTACT o)))

(defmethod ADD-COMPONENT ((c UI) (o TOOL) &optional role-name)
  (push o (comp-TOOL c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c UI) (o TOOL))
  (setf (comp-TOOL c) (delete o (comp-TOOL c) :test #'equal)))

(defmethod ADD-COMPONENT ((c UI) (o CALLS) &optional role-name)
  (push o (comp-CALLS c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c UI) (o CALLS))
  (setf (comp-CALLS c) (delete o (comp-CALLS c) :test #'equal)))

(defmethod ADD-COMPONENT ((c UI) (o COMMANDS) &optional role-name)
  (push o (comp-COMMANDS c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c UI) (o COMMANDS))
  (setf (comp-COMMANDS c) (delete o (comp-COMMANDS c)
:test #'equal)))

(defmethod ADD-COMPONENT ((c UI) (o CALL) &optional role-name)
  (push o (comp-CALL c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c UI) (o CALL))
  (setf (comp-CALL c) (delete o (comp-CALL c) :test #'equal)))

(defmethod ADD-COMPONENT ((c UI) (o COMMAND) &optional role-name)
  (push o (comp-COMMAND c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c UI) (o COMMAND))
  (setf (comp-COMMAND c) (delete o (comp-COMMAND c) :test #'equal)))

(defmethod ADD-COMPONENT ((c UI) (o INTERACTOR) &optional role-name)
  (push o (comp-INTERACTOR c))
  (setf (comp-parent o) c))

(defmethod DELETE-COMPONENT ((c UI) (o INTERACTOR))
  (setf (comp-INTERACTOR c) (delete o (comp-INTERACTOR c)
:test #'equal)))
```

```
(defmethod COMPONENTS-OF ((o UI))
  (list (comp-TOOL o)
          (comp-CALLS o)
          (comp-COMMANDS o)
          (comp-CALL o)
          (comp-COMMAND o)
          (comp-INTERACTOR o)))
```

B CLOS code for process Interactor Translator

```
(in-package 'SARA :use '(lisp pcl orel))

(defun produce-installation (gmb-model)

  (let* ((contact-names nil)
 (tool-names nil)
 (model (storable-name gmb-model))
        (file (open (format nil "/x/CDE/SARA/~A-install.lisp" model)
     :direction :output
     :element-type :default
     :if-exists :supersede)))

    (format file
"~%(defmethod install ((tool ~A) &key form simulator)~%" model)
    (format file
"~%  (setf (tool-callbacks tool) (make-callbacks))~%")
    (setq contact-names (get-contacts gmb-model)
  tool-names (get-tools gmb-model))

    (dolist (name contact-names)
      (format file
       "~%  (setf (cn::button-label-text (car (cn::form-buttons
form)))")
      (format file "~%       \"CONTACT ~A ID:\")~%" name)
      (format file "~%  (if (cn::collect-info form)")
      (format file "~%      (let ((id (read-from-string")
      (format file
      "~%                (cn::intext-value (first
       ~%                (cn::form-intexts
form)))))")
      (format file "~%              (contact nil))")
      (format file "~%        (setq contact (ow::read-object
:id id))")

      (produce-callbacks name file)

      (format file "~%        (setf (orel::tuple-list
(tool-callbacks tool))")
      (format file "~%              (cons (make-callbacks-tuple")
      (format file "~%                     :contact contact
:tool tool :message-object nil)")
      (format file "~%                   (orel::tuple-list
(tool-callbacks tool)))))~%"))

    (format file "~%   (setf (tool-uses tool) (make-uses))~%")
    (format file  ")~%")
    (close file))

(defun produce-callbacks (module file)

  (let ((data-link nil)
(proc nil)
(control-arc nil))

    (dolist (socket (comp-socket module))
      (setq data-link (get-data-arc socket))
      (if data-link
  (if (or (eq (data-arc-type data-link) 'w)
  (eq (data-arc-type data-link) 'rw))
      (progn
(setq proc (data-processor
    (first
     (find-objects
      (first (comp-data-rel (comp-parent data-link)))
      #'(lambda (x) (eq (data-arc x) data-link))))))
(return)))))

    (dolist (socket (comp-socket module))
      (setq control-arc (get-control-arc socket))
      (if control-arc
  (if (eq socket (second (first (control-arc-head-set control-arc))))
      (progn
(format file
"~%       (add-callback (cn::canvas-view contact)")
(format file
"~%                     ~A" (callback control-arc))
(format file
"~%                     'tm-run")
(format file
"~%                     simulator")
(format file
"~%                     ~A" control-arc)
```

```
(format file
"~%                         ~A" data-link)
(format file
"~%                         ~A)" proc)))))))

(defun callback (control-arc)
  (let ((name (storable-name control-arc))
(call nil))
    (if (or (string-equal name "C1") (string-equal name "LDN"))
(setq call ":press-left"))
    (if (or (string-equal name "C2") (string-equal name "LUP"))
(setq call":release-left"))
    (if (or (string-equal name "C3") (string-equal name "LCK"))
(setq call ":single-click-left"))
    (if (or (string-equal name "C4") (string-equal name "L2CK"))
(setq call ":double-click-left"))
    (if (or (string-equal name "C5") (string-equal name "LMV"))
(setq call ":move-left"))
    call))

(defun tm-run (x y u v simulator control-arc data-arc processor)
  ($write-data processor (storable-name data-arc) (cons x y))
  (update-tokens control-arc 1)
  (with-slots (*enabled*) simulator
    (dolist (node (dynamic-headset control-arc))
      (if (typep node 'control-node)
  (if (evaluate (control-node-in-logic node) node)
      (setq *enabled* (add-element *enabled* node))))))
  (tm-start simulator))

(defun get-tools (module)
  (let ((type (module-type module))
(tools nil))
    (if (eql type 'tool)
(setq tools (list module))
(if (not (eql type 'contact))
    (dolist (submodule (comp-module module))
      (setq tools (append tools (get-tools submodule))))))
    tools))

(defun get-contacts (module)
  (let ((type (module-type module))
(contacts nil))
    (if (and (eql type 'contact) (comp-gmb module))
(setq contacts (list module))
(if (not (eql type 'tool))
    (dolist (submodule (comp-module module))
      (setq contacts (append contacts (get-contacts submodule))))))
    contacts))
```

References

[1] Barth [1986]. An Object-Oriented Approach to Graphical Interfaces. *ACM Trans. on Graphics* 5 2, April 1986, 142–172.

[2] G. Booch [1991]. *Object-Oriented Design with Applications.* Benjamin Cummings, 1991.

[3] CLX [1989]. *CLX Common LISP X Interface*, Texas Instrument Incorporated, 1989.

[4] D. Coleman, F. Hayes and S. Bear [1992]. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Trans. on Software Engineering* SE-18 1, Jan. 1992, 9–18.

[5] J. DeSoi, W. Lively and S. Sheppard [1989]. Graphical Specification of User Interfaces with Behavior Abstraction. *Proc. ACM SIGCHI89*, May 1989, 139–144.

[6] P. Dewan and R. Choudhary [1991]. Flexible User Interface Coupling in a Collaborative System. *ACM CHI'91 Conference Proc.*, April 1991, 41–48.

[7] P. Dewan and R. Choudhary [1991]. Primitives for Programming Multi-User Interfaces. *ACM UIST Symposium Proc.*, Nov. 1991, 69–78.

[8] G. Estrin, R. Fenchel, R. Razouk and M. Vernon [1986]. SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems. *IEEE Trans. on Software Engineering* SE-12 2, Feb. 1986, 293–311.

[9] R. Fenchel and G. Estrin [1982]. Self-Describing Systems Using Integral Help. *IEEE Trans. on Systems, Man, and Cybernetics* SMC-12 2, March/April 1982, 162–167.

[10] G. Fisher [1987]. An Object-oriented Construction and Tool Kit for Human-Computer Communication. *Computer Graphics* 21 2, April 1987, 105–109.

[11] J. Foley [1987]. Transformations on a Formal Specification of User-Computer Interfaces. *Computer Graphics* 21 2, April 1987, 109–113.

[12] J. Foley, W. Kim, S. Kovacevic and K. Murray [1989]. Defining Interfaces at a High Level of Abstraction. *IEEE Software*, Jan. 1989, 25–32.

[13] J. Goguen and M. Moriconi [1987]. Formalization in Programming Environments. *Computer*, Nov. 1987, 55–64.

[14] M. Green [1985]. The University of Alberta User Interface Management System. *ACM SIGGRAPH'85 Conference Proc.*, July 1985, 205–213.

[15] M. Green [1986]. A Survey of Three Dialogue Models. *ACM Trans. on Graphics* 5 3, July 1986, 244–275.

[16] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring and M. Trakhtenbrot [1990]. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Trans. on Software Engineering* SE-16 4, April 1990, 403–414.

[17] D. Harel [1987]. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 1987, 231–274.

[18] A. Harbert, W. Lively and S. Sheppard [1990]. A Graphical Specification System for User-Interface Design. *IEEE Software*, July 1990, 12–20.

[19] R. Hartson [1989]. User-Interface Management Control and Communication. *IEEE Software*, Jan. 1989, 62–70.

[20] R. Hill [1986]. Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction—The Sassafras UIMS. *ACM Trans. on Graphics* 5 3, July 1986, 179–210.

[21] D. Hix and R. Schulman [1991]. Human-Computer Interface Development Tools: A Methodology for Their Evaluation. *Comm. of the ACM*, 34 3, March 1991, 74–87.

[22] S. Hudson [1987]. UIMS Support for Direct Manipulation Interfaces. *Computer Graphics* 21 2, April 1987, 120–124.

[23] W. Hurley and J. Sibert [1989]. Modeling User Interface-Application Interactions. *IEEE Software*, Jan. 1989, 71–77.

[24] R. Jacob [1986]. A Specification Language for Direct-Manipulation User Interfaces. *ACM Trans. on Graphics* 5 4, Oct. 1986, 283–317.

[25] E. Kantorowitz and O. Sudarsky [1989]. The Adaptable User Interface. *Comm. of the ACM*, 32 11, Nov. 1989, 1352–1358.

[26] S. Keene [1989]. *Object-Oriented Programming in Common Lisp—A Programmer's Guide to CLOS* Addison-Wesley, 1989.

[27] K. Kimbrough [1989]. *A Quick and Dirty Guide to CLUE.* Texas Instruments Inc., Version 6.0, July 1989.

[28] K. Kimbrough and L. Oren [1990]. *Common Lisp User Interface Environment.* Texas Instruments Inc., Version 7.20, July 1990.

[29] J. Konstan and L. Rowe [1991]. Developing a GUIDE Using Object-Oriented Programming. *ACM OOPSLA'91 Conference Proc.*, Oct. 1991, 75-88.

[30] J. Kramer and J. Magee [1990]. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. on Software Engineering* SE-16 11, Nov. 1990, 1293–1306.

[31] K. Lantz [1987]. Multi-process Structuring of User Interface Software. *Computer Graphics* 21 2, April 1987, 124–130.

[32] K. Lantz, P. Tanner, C. binding, K. Huang and A. Dwelly [1987]. Reference Models, Window Systems, and Concurrency. *Computer Graphics* 21 2, April 1987, 87–97.

[33] T. Lewis, F. Handloser, S. Bose and S. Yang [1989]. Prototypes from Standard User Interface Management Systems. *Computer*, May 1989, 51–60.

[34] M. Linton, J. Vlissides and P. Calder [1989]. Composing User Interfaces with Interviews. *Computer*, Feb. 1989, 8–22.

[35] K. Lor and D. Berry [1991]. Automatic Synthesis of SARA Design Models from System Requirements. *IEEE Trans. on Software Engineering* SE-17 12, Dec. 1991, 1229-1240.

[36] S. Mujica [1991]. *A Computer-based Environment for Collaborative Design.* Ph.D. diss., Computer Science Dept., U. of California, Los Angeles, 1991.

[37] R. Mulligan, M. Altom and D. Simkin [1991]. User Interface Design in the Trenches: Some Tips on Shooting from the Hip. *ACM CHI'91 Conference Proc.*, April 1991, 232–236.

[38] B. Myers [1989a]. Encapsulating Interactive Behaviors. *Proc. ACM SIGCHI89*, May 1989, 319–324.

[39] B. Myers [1989b]. User-Interface Tools: Introduction and Survey. *IEEE Software*, Jan. 1989, 15–23.

[40] B. Myers, D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish and P. Marchal [1990]. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *Computer*, Nov. 1990, 71–85.

[41] R. Razouk, M. Vernon and G. Estrin [1979]. Evaluation Methods in SARA—The Graph Model Simulator. *Conference on Simulation, Measurement and Modeling of Computer Systems*, 1979, 189–206.

[42] L. Rowe, J. Konstan, B. Smith, S. Seitz and C. Liu [1991]. The PICASSO Application Framework. *ACM UIST Symposium Proc.*, Nov. 1991, 95–105.

[43] G. Singh and M. Green [1991]. Automating the Lexical and Syntactic Design of Graphical User Interfaces: The UofA* UIMS. *ACM Trans. on Graphics* 10 3, July 1991, 213–254.

[44] K. Tatsukawa [1991]. Graphical Toolkit Approach to User Interaction Description. *Proc. CHI'91*, April 1991, 323–328.

[45] M. Vernon [1983]. *Performance-Oriented Design of Distributed Systems.* Ph.D. diss., Computer Science Dept., U. of California, Los Angeles, 1983.

[46] P. Wellner [1989]. Statemaster: A UIMS Based on Statecharts for Prototyping and Target Implementation. *Proc. SIGCHI89*, May 1989, 177–182.

[47] C. Wiecha, W. Bennett, S. Boies and J. Gould [1989]. Generating Highly Interactive User Interfaces. *Proc. SIGCHI89*, May 1989, 277–282.