Modularizing a Concurrent Artist-based UIMS for Software Environments

TR-92-80

Jennifer A. M. Anderson
Department of Computer Science
Stanford University[1]

Richard N. Taylor
Department of Information and Computer Science
University of California, Irvine[2]

Michal Young
Software Engineering Research Center
Department of Computer Sciences
Purdue University

[1]Address correspondence to the first author at: Department of Computer Science, Stanford University, Stanford, CA 94305. Email: anderson@cs.stanford.edu. Phone: (415)723-4096.

[2]This material is based upon work supported by the National Science Foundation under Award No. CCR-8704311, with cooperation from the Defense Advanced Research Projects Agency, by the National Science Foundation under Award No.s CCR-8451421 and CCR-8521398, Hughes Aircraft (PYI program), and TRW (PYI program).

Contents

List of Figures

List of Tables

Abstract

A user interface management system (UIMS) for extensible software environments must promote uniformity, and yet be both extensible and powerful. We describe the architecture of *Chiron-0*, a UIMS designed to meet the demands of a software environment. We discuss the key concepts underlying the design, and how those concepts are realized in the implementation of a prototype. Our experiences with the prototype brought to light the successes of our approach as well as its limitations. We devised ways to circumvent some of these problems; others are influencing a redesign effort that is currently under way.

1 Introduction

Chiron is a user interface management system (UIMS) for extensible software development environments. It provides basic user interface functions and is used to build interactive, graphical tools. *Chiron*'s approach separates interaction from abstract tool behavior. A program called an *artist*, which sits between *Chiron* and the tool, makes decisions as to how the abstract objects in the tool are to be displayed on the the screen. A large amount of flexibility is attained by allowing the application and the UIMS to execute concurrently. Each displayed object can have its own thread of control – for example, *Chiron* can update the display at the same time the application responds to user commands.

An initial prototype called *Chiron-0* was built on top of version 11 of the *X Window System* [SG86], as part of the *Arcadia* software environment [TBC+88]. We are now in the process of designing *Chiron-1*, which has gained extensively from our experiences with *Chiron-0*.

Throughout this paper, the name *Chiron* is used when discussing issues involving a general UIMS for a software development environments, and the names *Chiron-0* or *Chiron-1* are used to refer to those particular systems. For more general information, Young [YTT88] describes the conceptual design of *Chiron* within the software environment framework, and discusses in depth the constraints the environment architecture imposes upon the design of the UIMS. Anderson [DYT88] presents a tutorial on how to use *Chiron-0* to build graphical tools.

This paper describes the architecture and design rationale of *Chiron-0*. First, we discuss the conceptual design questions facing user interface management system designers, and give an overview of the solutions we adopted. We present the architecture of the system by describing each major component and its relationship to the other components in detail. We also discuss implementation dependent issues, such as language and platform considerations. The results of our design decisions are then examined, and compared to current work in the field. Finally, we examine the limitations of our design and implementation and present various solutions to these problems. Our experiences with *Chiron-0* should prove valuable to others interested in the development of user interface management systems.

In order to gauge the success of the design and implementation of *Chiron* the goals should be clearly identified. In particular, effort was focused on exploring new ideas regarding the interface between tools and interactive components, rather than on visual design and human factors issues. The primary design goals for *Chiron* are discussed below.

Scope and power. A software development environment supports a broad range of activities, and so must its UIMS. In no way should the UIMS limit the sort of tools available to the environment, nor should it place restrictions on any particular tool.

The UIMS should strive to take full advantage of the workstation hardware on which it is implemented. Advanced graphics and color capabilities should be used whenever appro-

priate and beneficial. In particular, support should be provided for the *direct manipulation* style of interaction that is found in the *Xerox Star* [SIK+82] and its derivatives such as the *Apple Macintosh* [AC85]. Direct manipulation allows users to manipulate the *representations* of data objects on the display, giving them the illusion of directly controlling the actual objects.

Although bells and whistles are not among our primary goals, the UIMS should provide a platform on which such features could later be built.

Uniformity. Uniformity, or consistency, is a key concern for user interface management systems. The tool builder wants a uniform scheme for adding new tools to the environment as well as consistent mechanisms for the combining of tools, communications between tools, and persistent object storage.

From the user's viewpoint, the user interface should enable consistency of the input and output conventions across all tools. The user should be able to learn a single, consistent set of commands which apply to all tools.

Attempts at end user uniformity are typically made by imposing style guidelines on applications (such as the *Apple Macintosh* user interface standards), and by providing toolkits of commonly used components [Sun86, Xtk88].

Extensibility Software environments must be able to handle modifications, deletions, and additions, and so must the corresponding UIMS. New and improved tools and capabilities will become available, and it is necessary for them to be easily incorporated into the environment. Changes to the UIMS itself must not force changes to every tool. It is important that the UIMS be designed with growth in mind, or modifications such as these will be unacceptably difficult.

2 General Design Issues

Chiron-0's design incorporates a number of recent developments in user interfaces, and has tailored them for use in software development environments. These ideas are briefly presented below. A detailed discussion of *Chiron-0*'s concrete realization of the concepts described here is presented in later sections.

2.1 Separating UIMS and Tool Functionality

One of the fundamental aspects of *Chiron-0*'s design is the decoupling of interface and tool functionality. Most interactive tools in a software development environment produce displays in which the objects on the screen actually *represent* abstract data objects in the tool. This is in contrast to applications in which the display itself is the desired end result,

such as VLSI and image processing applications. *Chiron-0* exploits this characteristic of applications to separate user interface from tool functionality, rather than intertwining graphical manipulations with the tool itself. A number of recent user interface systems, such as *Smalltalk* [GRs83] with its model-view-controller (MVC) paradigm, also have a high-level organization that separate application data objects (the *model*) from the graphical user interface objects (the *view*). In addition to the standard advantages of modularity, this scheme promotes uniformity through the reuse of both interface and tool components.

2.2 Artists and Annotation

Chiron uses artists, first introduced by Myers in the *Incense Symbolic Debugger* [Mye83] to referee the interaction between the tool and the user interface. The artist associated with each data type in the tool is responsible for making decisions as to how the data type is to be depicted on the screen and how that depiction should change in response to user input.

Chiron uses a restricted form of inheritance, called annotation [SB86, SBK86], to attach artists to tools. Accessing the abstract data objects has the side effect of invoking the artist to update the display. For details on how artists and annotations are implemented in *Chiron-0*, see section 4.1.

Since annotations (unlike unrestricted inheritance) preserve the semantics of the underlying abstract data type (ADT), an annotation cannot introduce errors into the abstract data type. This implies that multiple artists may be attached to a single type to provide multiple depictions. When the underlying abstract data type instance changes, the currently attached artist takes care of the updating the display. Furthermore, since the underlying abstract data type is preserved, annotations can be nested. It is guaranteed that intermediate annotations will not change the semantics of the data type.

2.3 The Abstract Depiction

Artists make policy decisions about how an object is to be depicted: the UIMS is concerned with the mechanism of rendering these depictions on the screen. To keep these concerns separate, *Chiron-0* distinguishes between the *abstract depiction* as composed by the artists, and the *concrete depiction* displayed on the screen. A separate, asynchronous rendering agent is responsible for mapping the abstract depiction onto the concrete depiction. Having a structured representation between the model and view facilitates input correlation and incremental updates, and makes possible the representation of displays which are not derivable from data objects alone (e.g. a display layout modified by user manipulation).

2.4 Processing Commands

In *Chiron-0* the input routines and application are concurrent processes. This differs from the more common UIMS schemes in which either the input routines are subroutines to the application (called the *prompting model*) or the application is a subroutine to the user interface's input processor (called the *dispatch model*). The concurrent model used by *Chiron-0* allows multiple tools to work on the same display. Also, the UIMS can continue to update the display while the application responds to commands. A method for developing such concurrent interfaces, called the *device model* is presented by Anson [Ans82].

A device is similar to an abstract data type. The difference lies in the fact that an instance of an abstract data type changes state only as the result of an operation, whereas devices may change state autonomously. A device may also send messages to other devices: *Chiron-0* has a simple hierarchy of communicating devices in which composite devices receive events only from their components.

In *Chiron-0*, devices are combined with artists to manage the interaction of an abstract data type. The term *gadget* is used to refer to the combination of an artist and a device.

3 Organization of Modules

The previous sections have given a high-level view of the key design decisions made in *Chiron-0*. The purpose of this section is to describe, in greater detail, each of the modules in the implementation, so that the solution mechanism – and their ensuing consequences – can be fully understood. We also discuss how the various pieces fit together to form the overall system, and present some of the tradeoffs made in the implementation. The modules are organized as shown in Figure 1.

3.1 The Abstract Depiction

The abstract depiction is *Chiron-0*'s internal representation of graphical objects as composed by the artists. The abstract depiction is represented as a tree, whose root is a window onto the entire display screen in the current implementation. A tree structure was chosen because it reflects any inherent structure in an abstract data type better than a simple linear list, without the complexity of an arbitrary graph. This facilitates maintaining the correlation between the graphical objects in the display and the data objects in the tool, as well as simplifying the determination of which parts of the display need to be redrawn for incremental updates. The tree's hierarchy can be exploited to provide inheritance of graphical attributes and other properties.

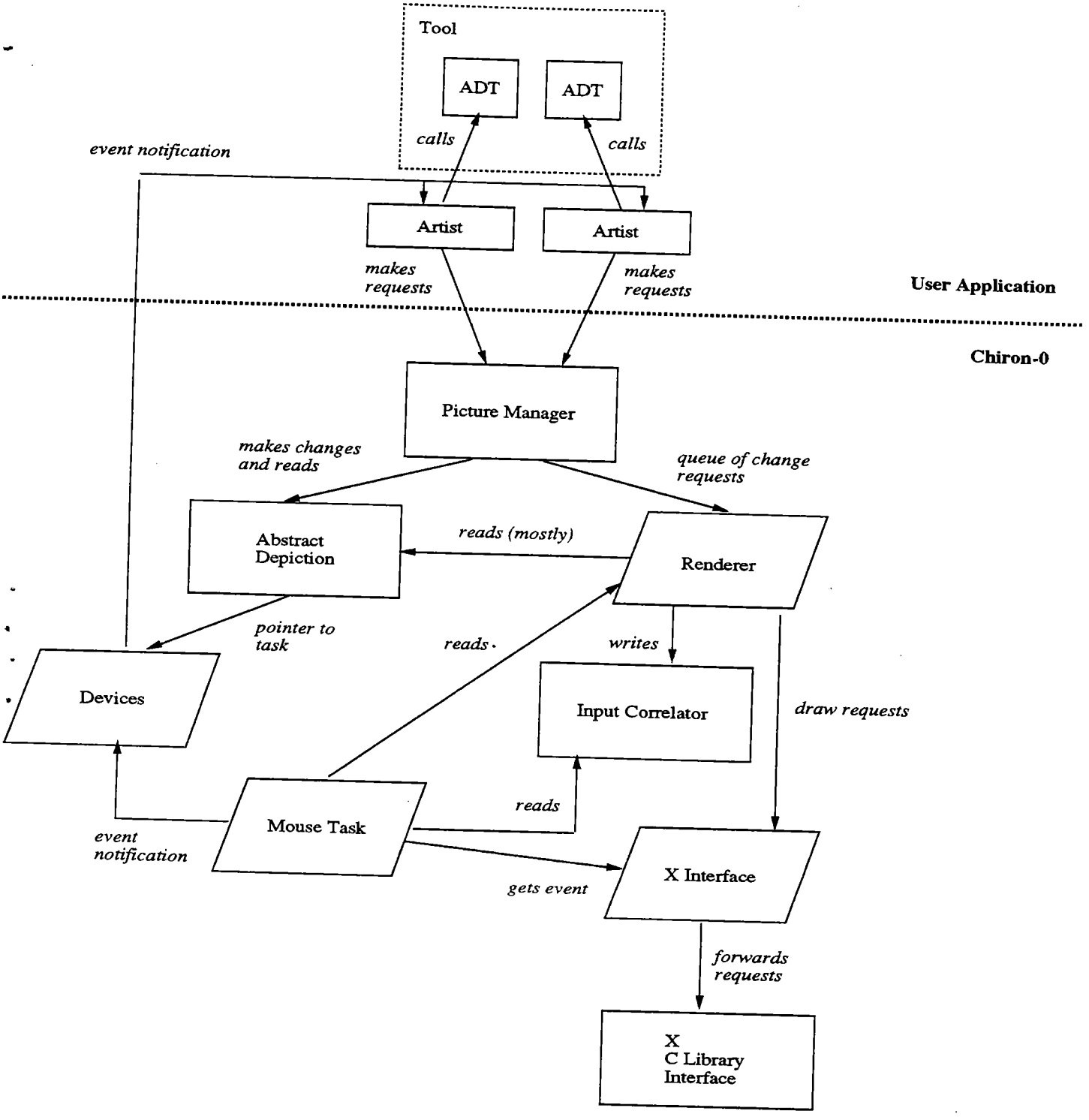

Figure 1: The overall organization of modules in *Chiron-0*.

3.1.1 Figures

The nodes of the abstract depiction tree are structures called *figures*. Each figure corresponds to a single graphical object. There are six different types of figures: *blocks*, *views*, *polygons*, *polylines*, *bitmaps* and *text*. This set of figures was chosen as a compromise between a general purpose set of primitives that would support a broad range of possible depictions, and specialization that would provide more power and support for a smaller class of depictions. The tree structure of the abstract depiction combined with this small number of figures is set up to work well with simple diagrams, such as are typical of the displays desired in software development environments. *Chiron-0*'s design also allows for multiple depictions to be incorporated by separate rendering agents into a single output screen, thus providing for more complex displays.

Blocks. A block is a logically infinite viewing surface. Each block has its own left-hand coordinate system (x goes left to right, y goes top to bottom).

Blocks are used for hierarchically grouping other figures, and figures are added to a single block upon creation. This implies that the block must be created before the figure it contains, and creates a strong bias toward building complex pictures from the top down. However, a top down strategy may not be appropriate for all structures. Thus, to allow for the bottom up creation of pictures, blocks (and only blocks) may be created in isolation and inserted into the abstract depiction tree at a later time. The order in which figures are inserted into a block is significant, as described in sections 3.1.2 and 3.1.3.

Blocks have the notion of *extent*. Even though blocks are considered to be infinite viewing surfaces, the figures within that block occupy a finite space. The extent of a block is the virtual rectangle that logically encompasses all the figures within it.

Blocks also have a defined *viewable region*. The viewable region of a block is the portion currently visible through a window. The difference between viewable region and extent is illustrated in Figure 2.

Views. A view is a rectangle through which a portion of a block is visible. A view is roughly equivalent to the usual idea of a *window*, and in fact views are implemented as windows in the *X Window System*. When a view is created, a block is automatically created within it to group the objects which may be visible through that view. The viewable region of a block can then be defined as the rectangular area within the block that intersects its enclosing view. The root of the abstract depiction tree is a view – in the implementation, this corresponds to the root window of X.

The relationship between blocks and views is illustrated in Figure 3.

Polygons. A polygon is an arbitrary closed shape. A polygon is defined as a series of points, any of which can be relative to other figures.

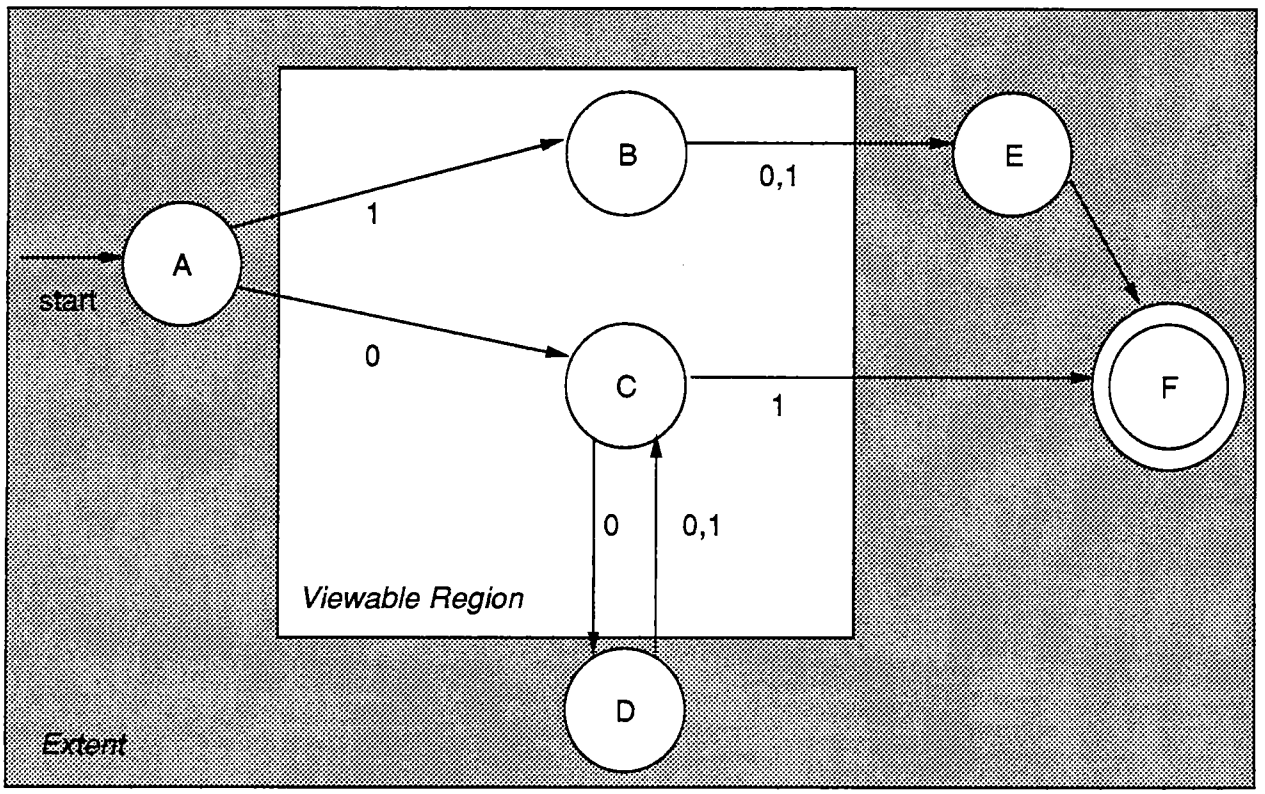

Figure 2: Distinguishing between a block's extent and viewable region. The extent is the virtual rectangle that encompasses the figures within the block. The viewable region is the portion of the block which is currently visible through a window.

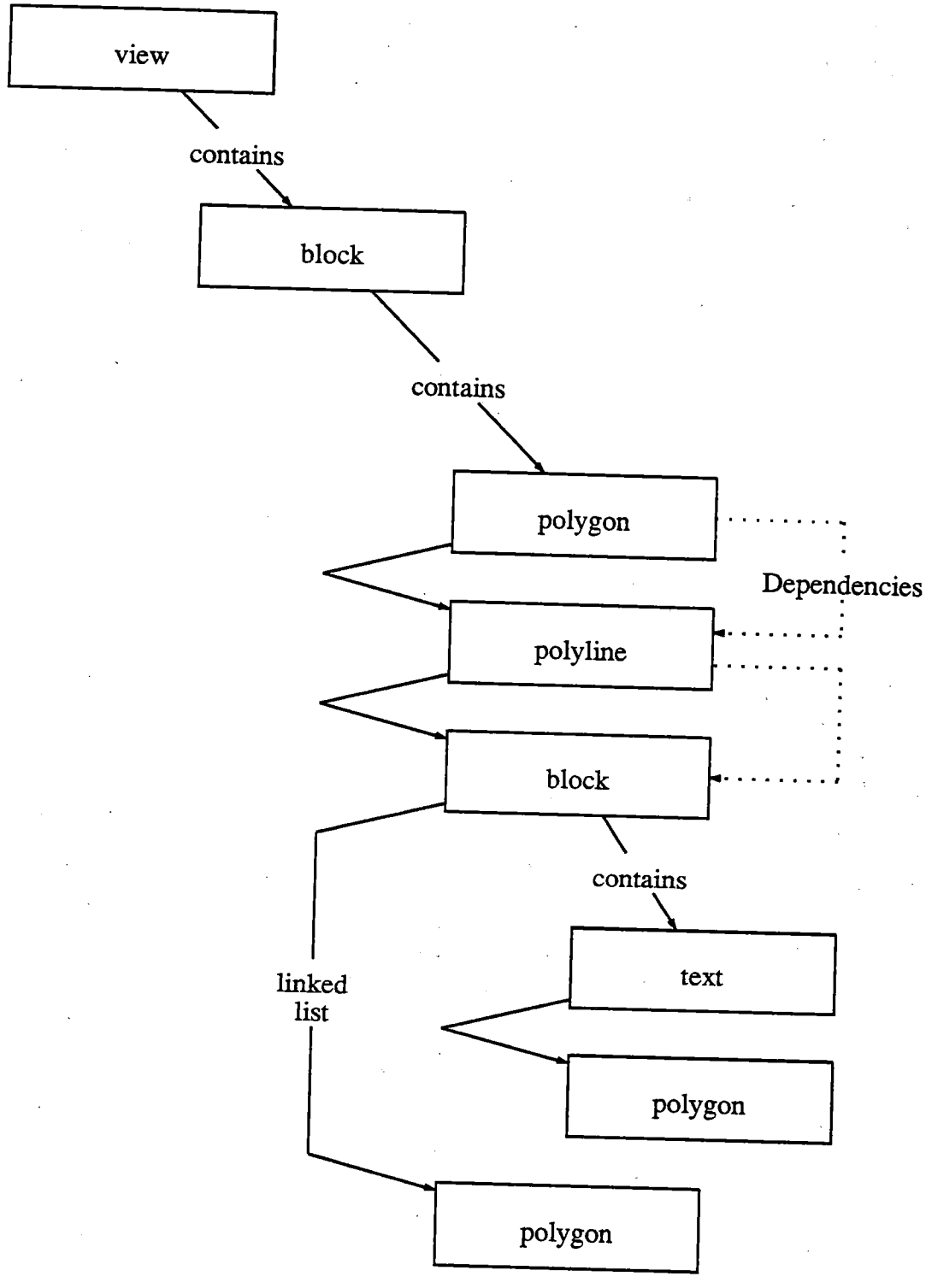

Figure 3: Blocks and views in the abstract depicition hierarchy. This figure shows a sample subtree of the abstract depiction. A view contains exactly one block. A block contains a sequence (implemented as a linked list) of polygons, polylines, text, bitmaps, views and other blocks.

Polylines. A polyline is an arbitrary open shape. Like polygons, polylines are defined by an arbitrary vector of points.

Bitmaps. Bitmaps are rectangular images. A bitmap is represented as a two-dimensional array of pixel values.

Text. Text is represented as a virtual rectangle enclosing a string of characters. Making text a separate type of figure (as opposed to modeling it as combination of polylines, for example) makes it possible for *Chiron-0* to provide specialized text operations that do not make sense for any other objects, as well as facilitating the use of predefined fonts available via the window system. Text strings are dynamic, and all the standard functions (such as insert, append, delete, and read) are available to manipulate them. Any font available to *X* can be used for *Chiron-0* text figures.

3.1.2 Relative Positioning of Figures

Upon creation, the location of most figures is specified in terms of their relation to other figures within their block. Absolute positioning is achieved by making a figure relative to the root, and therefore is only provided for figures in the root's block. Circular relationships are impossible, since a newly created object can only be relative to figures that have already been inserted into the tree.

Since polygons and polylines are made up of an arbitrary set of points, any of which can be relative to other figures, the behavior and shape of a figure are closely coupled. For example, if the northwest corner of a polygon is dependent upon a figure that is then moved, the northwest corner of that polygon may also move and thus change the dependent polygon's shape and size. Views and text are permanently rectangular so their shapes will not change, although they may be moved or resized in response to an action on a figure to which they are relative. Finally, a block has no shape, but may move as a result of its dependencies.

Corners and Anchors. Because of the issue of relative positioning discussed above, the way the dependencies of a figure are specified will determine both its shape and behavior. For aid in regulating this, *Chiron-0* distinguishes between sets of *corners* and *anchors*. Corners are the set of points which are connected to define the shape of a figure. Anchors are the points which characterize the figure's relative dependencies on other existing figures. Corners are specified in terms of the anchors of that figure. Note that a single point can be both an anchor and a corner, and this, in fact, is quite common. For a detailed example of the use of corners and anchors, see Figure 4.

In this figure, *Z*'s anchor *A*1 is 0 units (pixels) over and 25 units down from the southeast corner of figure *X* (recall that the y coordinate increases from top to bottom).

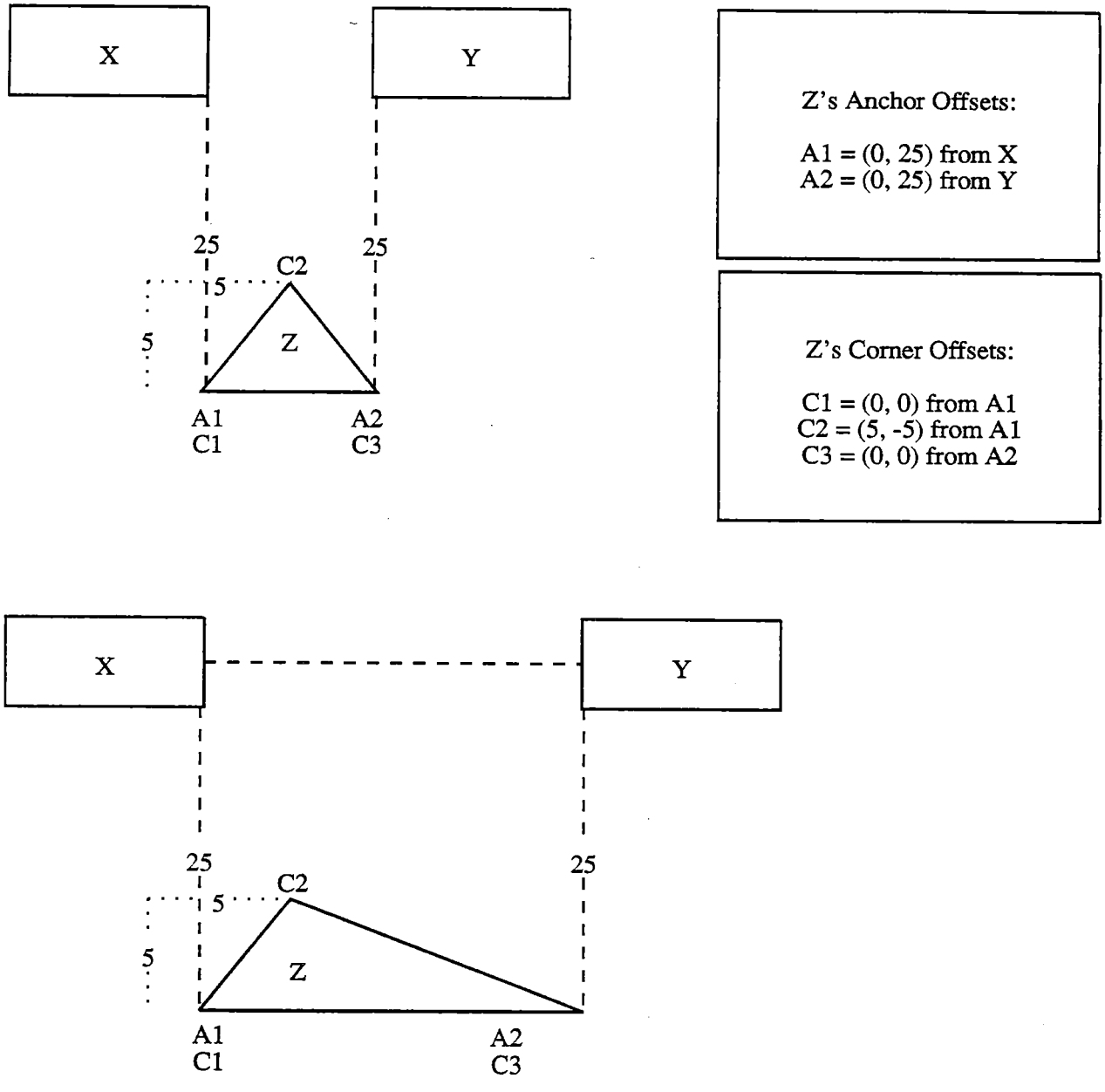

Figure 4: Using anchors and corners. Anchors determine the location of a figure, and corners determine its shape. A figure's anchors are relative to existing figures.

Similarly, $A2$ is (0, 25) away from the southwest corner of figure Y. Corners determine the shape of a figure. A figure's corners are given in terms of its anchors. In this example, $C2$ is (5, -5) away from $A1$. As the relative figure moves, the anchors which are dependent on that figure also move, possibly altering the shape of the dependent figure.

3.1.3 Occlusion Ordering of Figures

There are two different orderings of a set of siblings (i.e. two figures within the same parent block) in the abstract depiction tree: dependency order and occlusion order (see Figure 5).

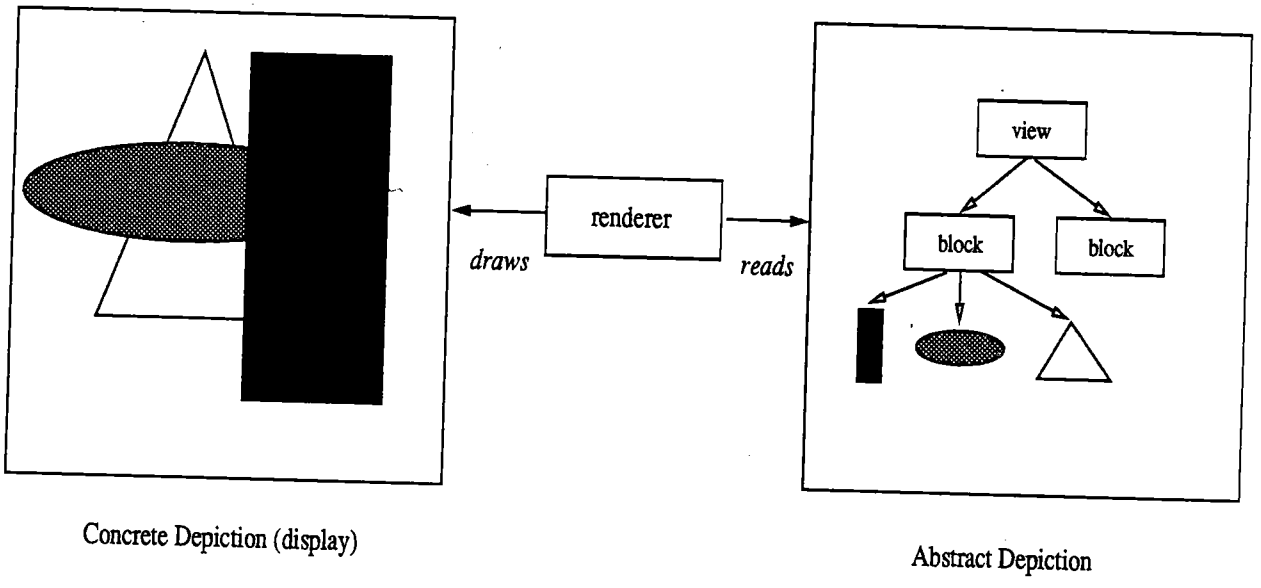

Figure 5: Occlusion vs. dependency ordering. The placement of figures in the abstract depiction tree determines their occlusion ordering, whereas the sequence in which they are inserted specifies their dependency ordering.

Dependency ordering is based on the relationships between dependent and relative figures. When the rendering agent calculates the coordinates of some figure A which depends on figure B, the position of figure B must already have been calculated. Occlusion order, on the other hand, is the order, from bottom to top, in which figures must be drawn (the last figures drawn may cover the previous figures, but never vice-versa). Note that implementing views as windows in X doesn't have exactly the correct semantics intended,

as a nested window is always drawn above everything in its parent window. *Chiron-0* stores its objects in occlusion order. When a figure is created and inserted, its position in the tree (and thus its position in the drawing order of objects) can be specified. The ordering is such that the leftmost child is drawn last. If no ordering is specified, then the new figure is placed in the leftmost position. Dependency order is implicitly specified by the order in which the figures are created.

3.1.4 Selectability

Upon creation, a figure can be tagged as *selectable*. Selectable figures are eligible to be picked by the end user (usually done by clicking one of the mouse buttons while the cursor is over that figure). By turning selectability on for a group of figures, a tool can make collections of objects appear as a single complex object. Also, turning selectability off for such objects as titles and scroll bar meters keeps end users from inadvertently playing with parts of the screen layout that should be off-limits.

3.1.5 Attributes

The graphical attributes of a figure (such as line style, color, and font) are stored with each figure in the abstract depiction. Unless otherwise specified, figures inherit attributes from their enclosing block. A suitable set of defaults are preset in the highest level block. The following attributes are available in *Chiron-0*:

- The background color of a figure (available for all figures). If an object is filled, the color becomes the figure's background color. The color *clear* is used for transparent objects that do not erase anything below them. Any other background color obscures the objects below it. On monochrome devices, colors are mapped to either black or white.
- The color of a polyline or of the outline of a polygon (i.e. foreground color).
- The style of a polyline, either dashed or solid.
- The width of each drawn segment of a dashed polyline.
- The arrow at the head of a polyline and/or an arrow at the tail of a polyline. A polyline can have arrows at both its head and its tail.
- The ability to draw line segments by inverting the background, commonly referred to as XOR drawing. This determines how the lines of a transparent figure combine with a figure below it. Lines in the upper figure will invert lines and regions of the same region in the lower figure.

- The font used for displaying text.
- The manner in which a figure stretches. The types of stretches available are *symmetric*, *constrained-x*, *constrained-y*, and *free form*. Symmetric stretches cause the figure to stretch equally in all directions. Constrained-x and constrained-y restrict the figure to stretching uniformly, but only in the x and y directions, respectively. Lastly, the free form stretch only moves one corner of the figure. Of course, the stretchiness of a figure is also constrained by its anchors. For example, in a symmetric stretch the northwest corner of a figure will only move if the corner is not also an anchor. For examples of the results of different kinds of stretches, see Figure 6.

3.1.6 Menus

A menu of textual commands can be associated with a figure. Menus are linked to figure objects, so that each figure (not just each type of object) can have a menu. *Chiron-0* handles the menu interaction, and when the end user makes a menu selection, *Chiron-0* notifies the appropriate artist (for details on how this is implemented, see section 3.4.4).

Chiron-0 allows a hierarchical system of menus to be defined, and supports both submenus and child menus. A submenu is a menu nested within another menu. Child menus are similar to submenus, except that if a child menu has a selection with the same name as one of its ancestors, the selection is suppressed in the ancestor menu. If all of the selections in an ancestor menu are suppressed, then it is not displayed. Child menus can be used to imitate an inheritance hierarchy in which a sub-class overrides operations defined by its super-class. A "walking" style is used for presenting nested sub and child menus (similar to the default menu style in *SunView*).

Individual selections within a menu can be made inactive or active. An inactive selection will not be highlighted as the cursor passes over it, and cannot be selected – if the user clicks on an inactive selection, it is as if no selection were made (a better implementation would have inactive selections visibly distinguished, e.g. by "ghosting"). All menu selections are initially active.

Chiron-0's abstract depiction provides a rich set of operations for creating and modifying figures. Functions are included for changing any of a figure's attributes, its shape or its position on the display. Additional operations are available that query and manipulate the abstract depiction structure itself.

3.1.7 The Picture Manager

The picture manager provides artists with an interface to *Chiron-0*. It exports all the types and operations in the abstract depiction which are useful to artists, as well as the routines artists need for event notification via devices (section 3.3). The picture manager

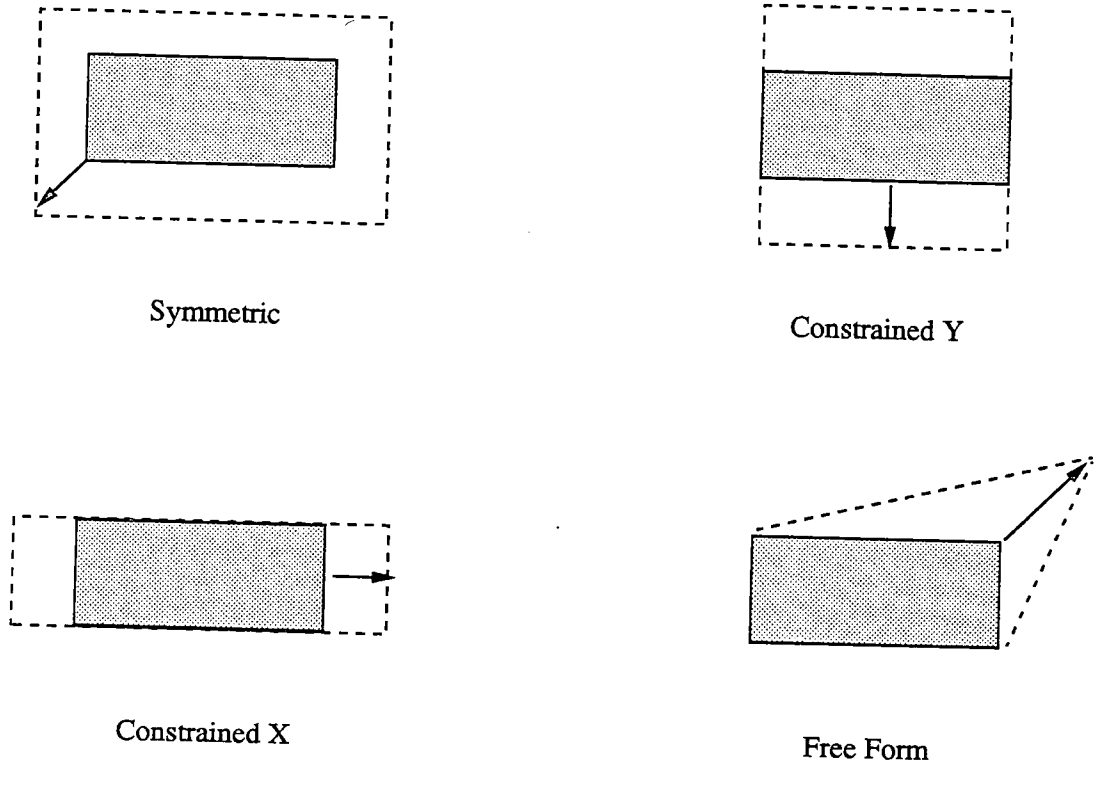

Figure 6: The results of different types of stretches. The dotted lines show the outcome after the figure is stretched the distance indicated by the vector.

also notifies the renderer when an artist calls a routine which potentially modifies the display.

3.2 Artists

There must be a clear distinction between the presentation issues decided by the artist and those decided by the user interface infrastructure. The artist should not be overburdened with making calculation-intensive decisions, but its flexibility and power should not be limited by delegating too many decisions to the UIMS. Often the artist will have more information than the user interface's rendering agent regarding which layouts are preferable. In *Chiron-0*, the artists take care of the top-level presentation decisions (such as the relative positioning and selectability of figures). Decisions that require extensive calculation or knowledge of exact coordinates are left to the renderer (the renderer is described in section 3.4.3).

The *Chiron-0* model assumes that the applications are based on abstract data types (ADTs). An artist creates an annotated type for the abstract data type it depicts (see section 2.2). Each operation on the ADT is overloaded with the annotated type and extended. The overloading routines typically call the original ADT operation in response to user input (the process by which an artist is notified of user actions is described in section 3.3) and then update the display if necessary. The artist may add local state information to keep track of an object's depiction, and may also add new functions to manage the display. Graphical figures are created and modified by calling the routines available in the picture manager interface to the abstract depiction.

The interface between the artist and ADT is determined solely by the ADT operations visible to the artist. The ADT must provide a functional interface to all references and manipulations of objects, so that these operations can be extended by the artist. As a result, the artist for an ADT in a given tool can generally be written without any modifications to the existing tool.

3.3 The Device Module

The device module makes the concurrency between *Chiron-0* and the application possible. The application and *Chiron-0*'s input routines are concurrent processes which communicate using the device abstraction described in section 2.4. Via the device mechanism, an artist receives notification of user actions that are directed towards graphical figures which depict objects of the artist's associated abstract data type. Conceptually, each figure (node in the abstract depiction tree) can have a corresponding device that defines its behavior. As all objects of the same type typically have the same behavior, all figures depicting objects of a single abstract data type have the same device in the current implementation (this was done to cut down on the number of active tasks at runtime). For example, a Petri net

application would have one device for all the Petri net places, and another device for all the transitions. This is in contrast to having a device for every individual object. There is a slight loss of flexibility to this approach, because information that could otherwise be kept in the control state of a particular object must be kept in the data state instead. However, the more general scheme of one task per object is just not practical because of tasking overhead, given the current state of *Ada* compiler technology.

The artist passes in a procedure that is called by *Chiron-0* whenever an event occurs for an object of that type. As a result of *Chiron-0*'s *concurrent model* of command processing, the application does not run only when an event procedure is called, rather the application and input processor run concurrently at all times. In contrast, the more common *dispatch model* treats a (single-threaded) application as a set of subroutines that execute only when called by the user interface.

Accordingly, a device is coupled with an artist to form a gadget that manages both the behavior and appearance of an abstract data type. The device abstraction is implemented by a group of three tasks: the mapping, relay, and channel tasks (see Figure 7).

The primary duties of the device module are to first determine the model object corresponding to the graphical figure on which the user acted, and then to notify the artist. To accomplish the first of these duties, the mapping task assigns serial numbers to the data objects and keeps them in a table. The serial number is also stored with the graphical figure in the abstract depiction. The artist is responsible for informing the mapping task (via the picture manager) of the data objects for which it wants to receive events, and for canceling the service when it is no longer needed. Thus, the mapping task is essentially a *monitor* which accepts commands from the artist and provides mutual exclusion for the table which registers data objects.

When the input listener receives an event from the underlying window system, it calculates the graphical figure for which the event was intended, and sends the event on to the channel task (see section 3.4.2 for details on *Chiron-0*'s input handler). The channel task acts like mailbox which accepts messages from the input listener and holds them until they are picked up by the relay task.

Upon receiving an event, the relay task uses the figure's serial number to obtain the appropriate data object from the mapping task. Once it has this data object, the relay task can notify the artist of the event.

At first thought, it would seem as if the mapping and relay tasks could be combined into a single task; this is largely an artifact of *Ada*'s asymmetric rendevous. The mapping task must be able to accept commands from the artist even when notification from the relay task to the artist is blocked. Also, the mapping task is essentially a server which sits in a wait loop, and therefore is not able to make calls on the channel to receive events.

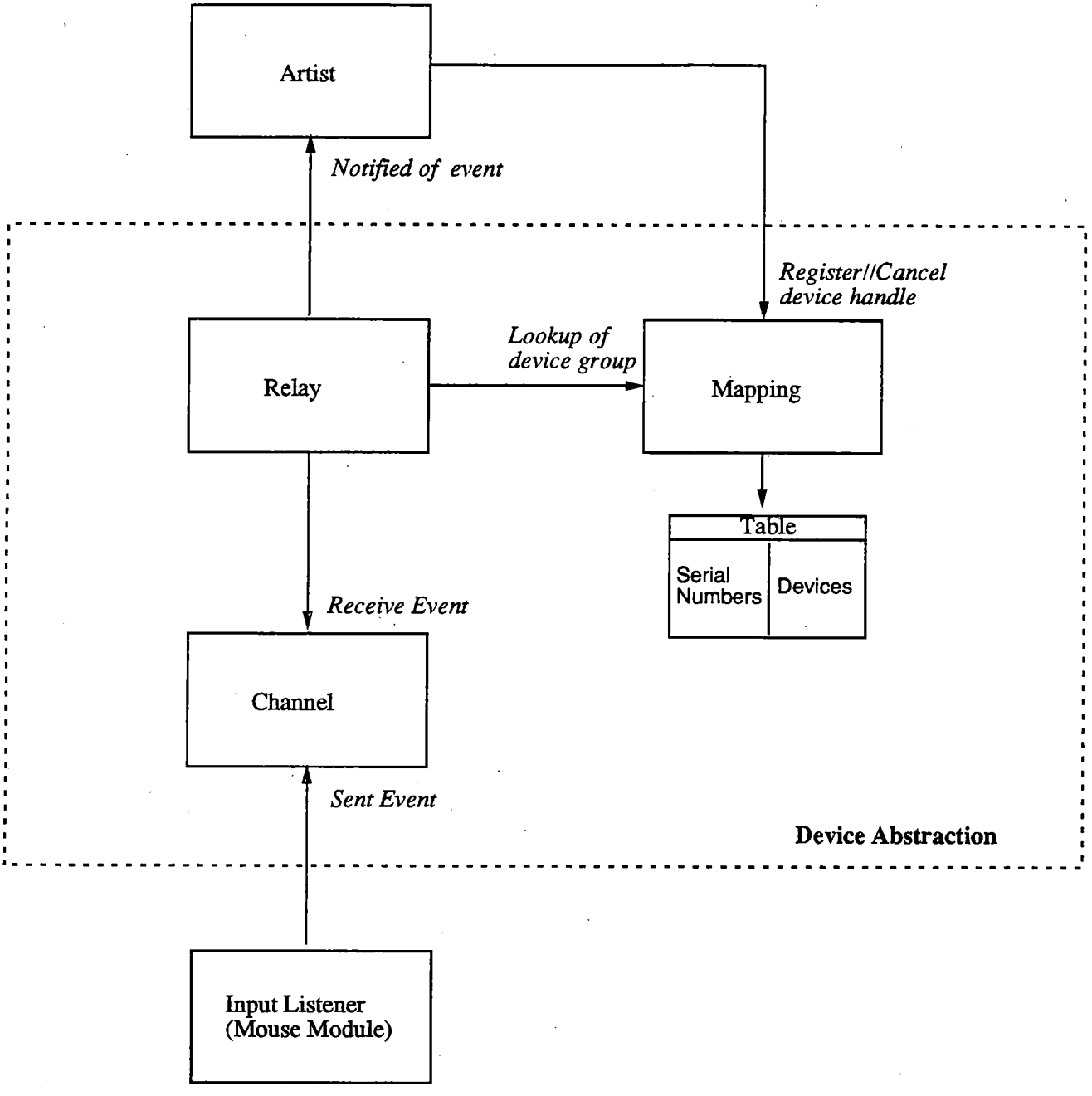

Figure 7: Implementation of the device abstraction. A group of three tasks are used to communicate events from the input listener in *Chiron-0* to the artist.

3.4 The Concrete Depiction Module

The concrete depiction is responsible for mapping the abstract depiction onto the screen, and routing user actions (events sent by the underlying window system) from the keyboard and mouse to the channel task. It is split into three modules: the input correlator, the renderer, and the mouse task.

3.4.1 The Input Correlator

The input correlator determines the abstract depiction view in which an event occurred. The events *Chiron-0* receives from the *X Window System* indicate the window to which the event was directed and the position of the mouse at the time. Accordingly, the correlator's primary duty is to maintain a mapping between the *X* windows and the corresponding views in the abstract depiction. For this purpose, the correlator keeps a table of *X* window identifiers and a pointer to the associated view (recall that *Chiron-0* views are implemented as *X* windows). The renderer is responsible for depositing this information into the correlator's table.

3.4.2 The Mouse Task

The mouse task receives all the events sent by the *X Window System* ('mouse task' is somewhat of a misnomer, since this module handles all input from both the keyboard and mouse), and translates them into *Chiron-0* events. *Chiron-0* events convey the same information as raw *X* events, but in terms of figures which is more suitable for use by artists (for more details on events, see section 3.4.4 below).

The mouse task must determine the figure for which the event was intended. For this reason, the bounding box (virtual rectangle that completely encompasses a graphical object) of each figure is stored along with it in the abstract depiction tree. This information is calculated and deposited into the abstract depiction by the renderer. The mouse task first looks into the correlator's table to find the view in the abstract depiction tree representing the window in which the event occurred. It then traverses the subtree of the abstract depiction rooted at that view, comparing the event's coordinates with the bounding box of each figure until a match is found.

Now that the mouse task has found the figure for the event, it must send the event on to the proper device – more specifically the mapping task within a device – which will then forward it to the artist. Recall that a figure *may* have an associated device; only those figures for which an artist wishes to receive events will have one. The mouse first checks if the figure for the event has an associated device – if so, the event is sent. Otherwise, the mouse task searches upwards in the abstract depiction tree (following parent pointers) for a figure with a device. If none is found, the event is discarded.

3.4.3 The Renderer

The renderer produces the concrete depiction which is presented to the end user by making calls to the underlying window system. *Chiron-0*'s basic drawable figure types (view, polygon, polyline, bitmap and text) are very similar to to *X*'s primitive shapes, so little extra calculation is needed. Whenever an artist makes a call to the picture manager that will cause the display to change, the picture manager sends a message to a mailbox task maintained by the renderer. The renderer then asynchronously processes the requests in the order they were received, getting the information it needs by reading the abstract depiction tree. In the current implementation, each display update message causes the entire screen to be redrawn; the renderer can also be told to batch such updates and incorporate them into a single redraw. A smarter update algorithm which only redraws the modified areas of the display is certainly possible within the current design. An initial attempt at smarter refresh was made using *X11* clipping regions to redraw only a small region surrounding updated figures, but the *X11R3* implementation of clipping regions was so slow that the smarter refresh policy actually slowed *Chiron-0* down. When the renderer draws a figure for the first time, it calculates that figure's bounding box, and stores those coordinates in the abstract depiction for later use by the mouse task. Upon drawing a view – which involves creating a new window – the renderer deposits the window identifier returned by *X* in the correlator's table.

3.4.4 Events

The mouse task, as mentioned above, converts raw *X* events into a form more useful to artists. *X* events communicate the details of user actions with window identifiers and coordinate values, which is too low level for artists that are working with *Chiron-0* views and other figures. Specifically, *Chiron-0* events contain the following information:

- The kind of event received (such as expose, move or select) and possibly a numeric detail (for example, representing which selection was made in a menu).
- The figure acted upon in the event and the figure which actually received the event (these may be different – see section 3.4.2). Also for binary operations, the object figure is included.
- The devices of both the figure that received the event and of the object figure.
- The initial and final positions of the mouse during the event (used primarily for move events).
- A timestamp of when the event occurred.

4 Implementation Considerations

The preceding sections have emphasized the implementation independent aspects of the *Chiron-0* design, noting a few implementation limitations where relevant. This section focuses on issues specific to the implementation, and examines the results of the decisions that were made.

4.1 Ada as the Implementation Language

Chiron-0's design is language independent, allowing a clear distinction to be made between the UIMS architectural issues and language dependent concerns. Discussions and functionality considerations led us to decide on *Ada* [ALR83] as the primary implementation language. A number of advantages and disadvantages to this decision were found in the course of building and evaluating the *Chiron-0* prototype. In particular, *Ada* package construct for building ADTs and its built-in tasking for concurrency fit very well with the *Chiron-0* model. However, the lack of inheritance and procedure variables in the language proved to be handicaps in mapping the conceptual design into the implementation.

Advantages of Ada. Aside from the general positive aspects of using *Ada* as an implementation language (such as explicit provision for modules through the package construct), there are a number of particular advantages to using it to implement *Chiron-0*.

Abstract data types are easily implemented using *Ada* packages. This is a significant benefit, as the *Chiron-0* model assumes that the tools for which artists are written are based on abstract data types.

Another considerable advantage of *Ada* is its tasking primitives. The *Chiron-0* model requires concurrency between the application and the UIMS, and the device abstraction described in section 3.3 maps naturally into tasks. In the implementation of *Chiron-0* it was assumed that tasks were fairly inexpensive but not free. Thus, we opted for one three-task device for all of the graphical objects representing the data objects of an abstract data type, rather than associating a different task for each figure. For example, a Petri net application would have one device for all the Petri net places, and another device for all the transitions. This is in contrast to having a device for every individual object. Also, there are no polling loops (the tasks are lazy tasks) in the implementation of devices.

Disadvantages of Ada. Ideally, an artist should be able to pass *Chiron-0* an arbitrary procedure to call when an event is received for a particular object. However, since *Ada* does not support procedure variables, they had to be simulated using task types and generics as described in [LH83, Rou85].

Ada's type system caused the worst mismatch between the high level design and the implementation. *Ada* is strongly typed and does not support inheritance. This made it

difficult to implement annotation, which had to be simulated with packages and overloading. Given an abstract data type package as the original tool, the artist can be written as another package which exports the types and operations of the original. The artist then performs its own processing to receive events and update the display before or after calling the operations in the original tool.

Furthermore, since *Ada* objects are static, it is impossible to dynamically attach and detach artists to individual objects at runtime. Moreover, artists must be bound statically to an entire class of objects (of course objects can be created and drawn on the display dynamically). Also, since the *Ada* assignment operator cannot be overloaded, procedure calls are necessary to make assignments to annotated objects. The problems encountered in the implementation of *Chiron-0* are discussed in more detail in section 5.2.

4.2 Deciding on a Platform

Our primary interest in building *Chiron-0* has been in the user interface issues – we are not interested in graphics *per se*, and so wanted to use existing graphics software as much as possible. The choice of a graphics platform is an important one as it determines, to a large extent, the course of the implementation.

Conventional graphics packages, such as *Core* [MvD78] and *GKS* [BEHtH82], were evaluated. The primary shortcoming of these traditional standards is that they only provide support for producing graphical output. They do little to assist in relating the figures on the display with the data objects those figures represent. These packages are intended to create intricate graphics, but typical software environments rarely take advantage of such functionality. Furthermore, these packages are not yet available for all the modern bit-mapped workstations for which *Chiron-0* is intended[1] For these reasons, very few software development tools have used these conventional packages.

Window systems such as *X*, *NeWS* [NeW87] and *SunView* [Sun86] are more suited to the platform *Chiron-0* needs. They provide support for graphics and windowing, as well as higher level components (i.e., scrollbars, menus). Also, if necessary, a tool that uses a conventional graphics package could always run in its own window under these systems. Although most of the functionality of window systems is still geared towards producing graphical output, they do provide a control structure for processing user inputs.

SunView and NeWS. Both *SunView* and *NeWS* were evaluated, and rejected for several reasons. Neither of these systems are portable as they will run only on Sun workstations. *NeWS* has the desirable feature of being network transparent (*SunView* is not network transparent), and and is very flexible since its client and server communicate using *Postscript* [Ado85] programs. However, for *Chiron-0* the support for the handling of user

[1]Sun does have SunCORE and SunGKS packages for use with its SunView window system.

inputs is one of the significant advantages of window systems in general, and *Postscript* is designed primarily for display, not input correlation. Also, *NeWS* has failed to attract a large following among window system users, and does not have a broad support base. Another important consideration is that the style of *SunView* applications assumes a single thread of control per large process. A client program (window systems treat both the UIMS and the application as a single client) is expected to sleep until it is notified by the window system. Notification is accomplished via a call to a client's procedure, thus informing the client that an event (user action) has occurred. This protocol is in direct conflict with the concurrency between the application and UIMS required by the *Chiron-0* model. The *SunView* notifier does have an "explicit dispatching" style in which a client passes control to the window system at regular intervals. Notification is not required in this case, but the window system still assumes a single thread of control.

The X Window System. The *X Window System* was best suited for *Chiron-0* for the following reasons. *X* has become the standard window system, and is portable to a wide range of hardware systems. *X* is network transparent, meaning that a distributed environment can be supported. Also, *X*'s client/server model reduces the size of client programs. Finally, *X*'s control structure is compatible with the concurrency in *Chiron-0*. All events are represented as data objects in a queue. When the client wants an event, it can perform either blocking (client waits until an event arrives before returning) or non-blocking reads from the queue. No notification is required, thus both the application and UIMS are continually active.

5 Experience Using the Prototype

The current version of *Chiron-0* is built on top of Version 11, Revision 3 of the *X Window System*. It has been tested on *Sun 3/50* and *Sun 3/260* workstations, and *DEC VaxStations*.

The prototype is in a stable working state. All the basic functionality described in this document has been implemented, including all types of figures (polygons, polylines, text, bitmaps, blocks and views), attributes and menus. The event structure is in place, and events can be received. Also, a small library of reusable components, including buttons, scrollbars, and dialogue boxes is available.

Several versions of *Chiron-0* have been in use at UC Irvine as well as other *Arcadia* sites for about two years now. As a result, several experimental applications were developed:

Petri Net Editor: A simple graphical editor for Petri nets, which can simulate transition firings.

IRIS Graph Editor: A graphical editor for *IRIS* graphs. *IRIS* is a graph-oriented internal representation of *Ada* programs [BFS88].

DFA Editor: An editor for creating deterministic finite state automata.

Semantic Ada Viewer: A viewer for displaying semantic information on statements in an *Ada* program.

The Petri net editor, written by one of the developers of *Chiron-0*, was the first application written and was a good test of the *Chiron-0* model (see Figure 8 for a screen shot of the Petri net editor). First, a package was written that implemented the Petri net abstract data type. Its data structures were designed for efficient use of a Petri net, and are quite different than the data structures appropriate for graphical representation. For instance, the number of tokens in a Petri net place is represented as a single integer, although graphically it is represented as a collection of individual graphical objects.

After the Petri net was complete, the artist responsible for maintaining the display of the Petri net and handling user inputs was written. The other applications listed above were also written in this two phase manner (i.e. first write the tool encapsulating the abstract data, and then write the artist).

The successful building of these applications shows that *Chiron-0*'s approach to UIMS design is feasible. Artists were written for the abstract data types within these various tools without modifying their original implementations. Furthermore, *Chiron-0* was able to manage the display while the tool concurrently processed the user actions. Yet despite this, our experiences in writing these applications brought to light several limitations of the design and resulting implementation. Ways were devised to avoid some of them; others are influencing a redesign effort currently under way.

5.1 Design Limitations

5.1.1 Explicit Representation

Within the annotation scheme used to bind artists to model objects (see section 2.2), access to the data type of the model is gained by going through the artist (recall that the artist overloads the operations available on the abstract data type). As the model data type is not directly accessible, the ADT must export all significant operations so that they can be overloaded and called by the artist. For example, the artist is informed if the user chooses to delete an object and updates the display accordingly. Yet, if the tool does not implement an explicit destruction routine, the artist has no way of effecting the deletion of the corresponding model object.

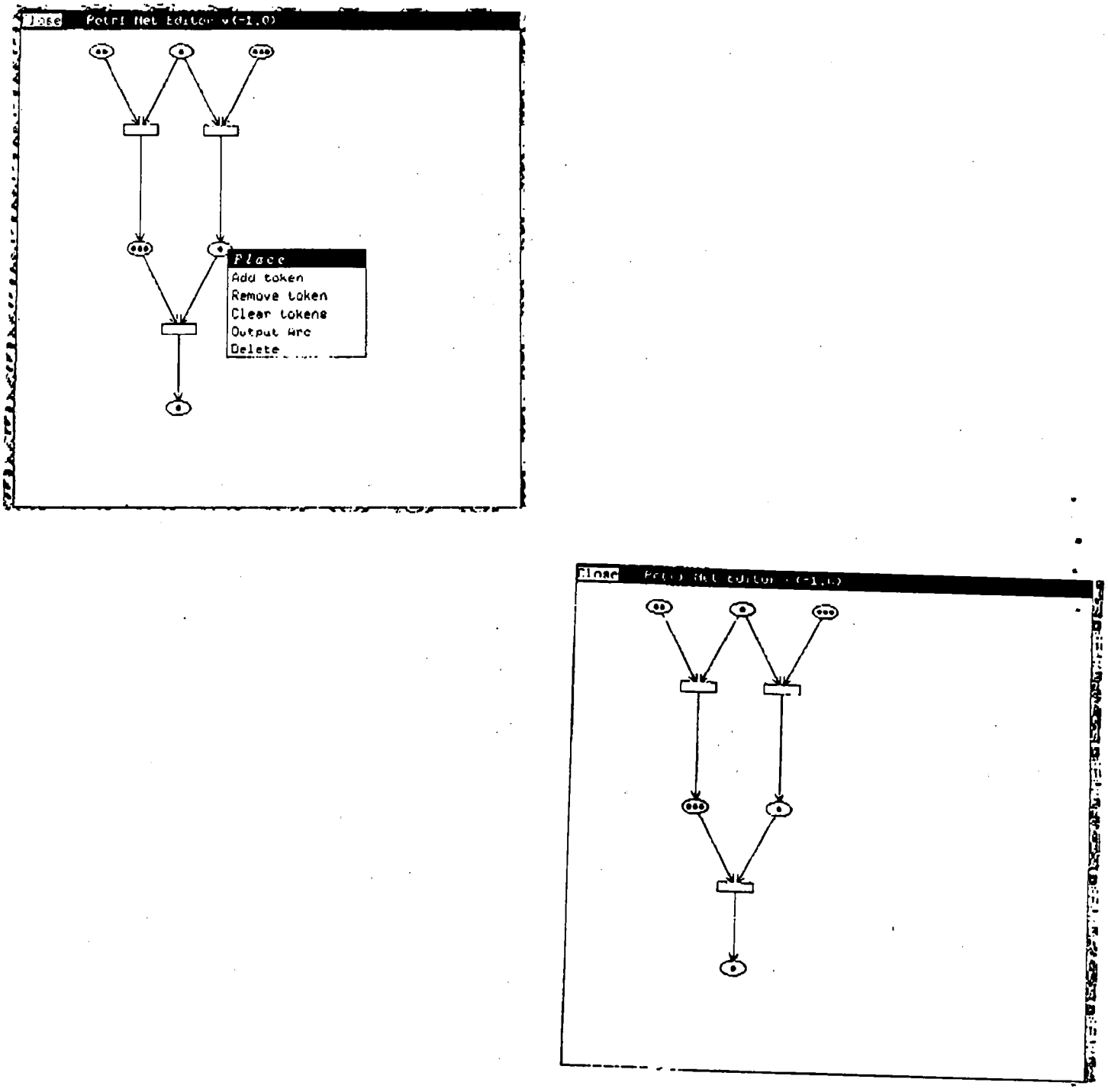

Figure 8: Sample screens from the Petri net editor.

5.1.2 Controlling the Semantics of Applications

Annotation is guaranteed to preserve the correctness of the underlying data type: new operations provided by the artists must not alter the state of data objects. For this reason, artists cannot provide any operations that depend on controlling the semantics of the ADT. For example, a general "undo" function would be very difficult to implement. Upon receiving an undo command from the user, the artist can easily update the display, but it has no way of relaying the change to the underlying model objects. As a result, unless the tool explicitly exports undo operations, the artist can only provide an undo option for those functions that do not alter model objects (for instance, graphically moving a place or transition in the Petri net editor without altering the underlying Petri net).

5.2 Implementation Problems

5.2.1 Semantics of the Assignment Operator

Assignment is not a first-class operator in many languages, including *Ada*, as it cannot be redefined or extended. The semantics of the assignment operator depend on the representation, rather than abstract semantics, of an object. For instance, assignment of a binary tree represented by a nodes-and-links structure causes the new copy to share a structure with the old, while assignment of a binary tree represented as an array does not cause structure-sharing. The artist must be aware of the model object's representation in cases such as this, in order to maintain a correct display. Thus complete transparency of an artist is possible only for *Ada* limited private types (a type where assignment is not allowed directly). In contrast, *C++* allows the assignment operator to be redefined in order to prevent this problem [Str86].

5.2.2 Simulating Annotation

Implementing annotation by simulating inheritance in a strongly-typed language such as *Ada* led to several problems. Experience writing the Petri net editor and other applications revealed that the artist in some cases must maintain much of the same state information as the abstract data type in order to keep the display current. This results in the artist duplicating a significant amount of code from the ADT. The heart of this problem lies in the fact that there is no link from the model object to the annotated object in the artist. Artists for *Chiron-0* were designed not to force any modifications to the underlying abstract data type, including the installation of pointers back to annotated objects in the artist. The artist manages the display based only on calls made into the abstract data type – it has no way of knowing when a model object changes state, and thus must simulate many of the tool's actions. For example, the Petri net abstract data type keeps track of all the arcs connecting places and transitions, and removes them when the place or transition

is deleted. At the same time, the Petri net artist also maintains the connectivity of the net's components so that it can correctly erase the arcs from the display in response to a delete place or transition command (see Figure 9).

In a language that supports true inheritance, the association between annotated and model objects is trivially maintained as they are actually the same object. The artist is informed of all calls made on abstract data type routines, including calls made from within the abstract data type to other internal subprograms. The annotated type would be implemented as a subclass of the abstract data type's class. This subclass overloads its superclass' routines (including object creation functions), from which the display is updated and the original routine called. As a result of true inheritance, whenever a function in the superclass is called, the overloading function in the subclass is executed, resulting in the proper display and correct model objects.

A better simulation of inheritance can be achieved with an automatically generated *dispatcher*, a software layer built around each abstract data type in a tool that dispatches incoming calls from both the tool and the artist. The dispatcher overloads all the subprograms in the abstract data type, and presents an interface that is identical to the type's interface. The tool must have all calls to the original abstract data type replaced with calls to the dispatcher (in *Ada* this modification means changing the `with Abstract_Data_Type` clause to `with Dispatcher`). The bodies of the overloading routines contain calls to the original abstract data type, followed by a call to the artist notifying it of the operation. With this scheme, the artist is informed of each subprogram call made to the abstract data type, with only a minor modification to the tool's code.

Note, however that the dispatcher does not solve the entire duplication of code problem. It is not a perfect substitute for a link between the abstract data type and the annotated object in the artist. The dispatcher can only notify artists of subprogram calls made to the abstract data type, not of every state change to an object. For example, within a graph ADT's subprogram to delete a vertex, it may also remove all the edges connected to that vertex as a side effect. The dispatcher will notify the artist that a vertex has been deleted when this subprogram is called, but it has no way of knowing which edges were also deleted. Thus, the addition of a dispatcher will result in less duplication of abstract data type code in the artist, but will not eliminate the problem completely.

Another manifestation of *Chiron-0*'s imperfect simulation of annotation is that the series of declarations used to annotate an abstract data type does not work correctly with recursive data structures such as trees and lists. Links within the recursive data type point to instances of the base type, not the annotation. As a result, the artist cannot access the next annotated object by following the existing links, and is forced to maintain the entire pointer structure itself! Again, in languages with inheritance, the annotated and model objects are one and the same, so this problem does not occur.

A final issue is that since the assignment operator cannot be overloaded, procedure calls are needed for assignment to annotated objects. This requires the abstract data type

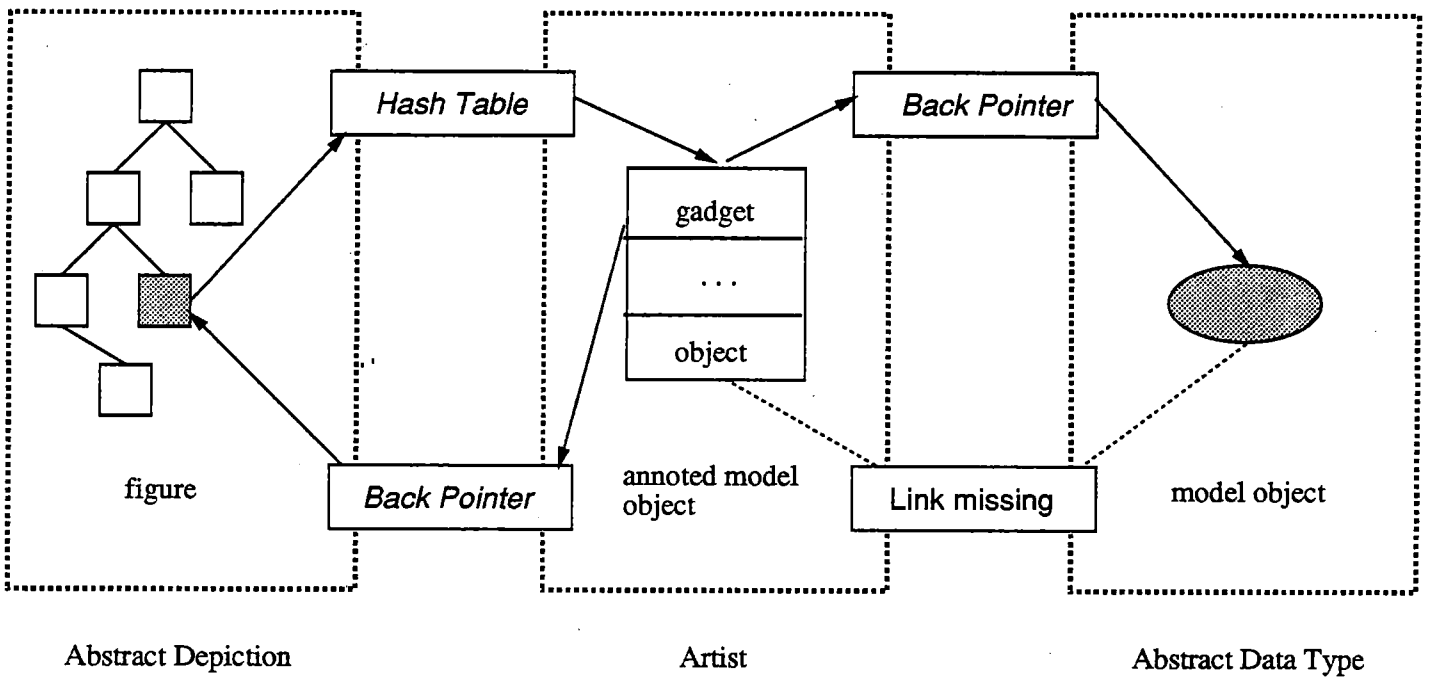

Figure 9: Duplication of abstract data type code in the artist. Pointers exist from the abstract depiction in *Chiron-0* to artist and back, as well as from the artist to the abstract data type. Yet, lack of a link from the abstract data type to the artist can force some of the data type's code to be duplicated in the artist. The artist must keep track of much of the same information as the abstract data type because the artist has no way of knowing when the state of a model object changes.

to export an explicit creation routine, which can then be overloaded by the artist.

5.2.3 Performance Problems

One of the most visible problems with *Chiron-0*'s implementation is its performance. Although the performance is acceptable for a simple editor (such as the Petri net and DFA editors), *Chiron-0* is still too slow to be used as a UIMS for real systems. For instance, a figure dragged across the screen cannot keep up with a quickly moving mouse. Also, there is a noticeable delay between the selection of a menu item and the corresponding update on the screen. Even though speed was not considered one of the primary goals in developing *Chiron-0*, it is an important consideration in the current redesign effort. There are a large number of factors, including quality of the generated code, tasking overhead, and choice of algorithms, that could account for *Chiron-0*'s slow performance, and unfortunately, few profiling tools for *Ada* programs are available.

Since *Chiron-0* is built on top of the *X Window System*, the performance of *X* must also be taken into account. *Chiron-0* interfaces to *X*'s library (written in *C*) via a set of *Ada* bindings[2]. A number of simple applications were written in both *C* and *Ada* to measure the effect of the binding's extra level of indirection on several common *X* library calls. The results are shown in Table 1. The numbers listed are elapsed times (obtained via the system clock), taken on a *Sun 3/50* with only the *X* server running (and one xterm window, of course).

These tests show that the *Ada* interface dominates the function call time for all but the most compute intensive *X* system routines (`XOpenDisplay`, `XLoadQueryFont`, and `XSetStandardProperties`). On the other hand, the absolute times are small – on the order of a few milliseconds, and empirically, the *Ada* programs were only slightly slower to the end user.

As there were no profiling tools available for the *Ada* compiler available to us, we wrote a simple profiler ourselves, and used it to time each of the routines in *Chiron-0*. The profiler records the elapsed and system time on entry and exit to each subprogram, and uses a call stack to keep track of nested routines. As there is only one stack, it can only support a single thread of control at a time. *Chiron-0*'s high degree of concurrency meant that the the tests had to be repeated for each subset of subprograms within a single thread of execution. Furthermore, a routine with a high time is not necessarily inefficient (tasks that are executing concurrently with a routine that is being timed will be included in the routine's final time). Despite this, the profiler does give some indication of where to start looking for inefficiencies. Table 2 shows a listing of the most time-consuming routines in *Chiron-0* (all routines which that had a cpu time of greater than 0.5 seconds are included). Again, these tests were performed on a *Sun 3/50* workstation with only the *X* server

[2]The *Ada* bindings were written by SAIC and are publically available

Overhead of using an *Ada* interface to *X*				
Name of Routine	*Ada* time (msec)	*C* time (msec)	*Ada* - *C* time (msec)	Percentage
XOpenDisplay	144.8	144	0.8	.6%
XDefaultScreen	2.6	0.0	2.6	100.0
XWhitePixel	3.6	0.0	3.6	100.0
XBlackPixel	3.0	0.0	3.0	100.0
XDefaultRootWindow	3.3	0.0	3.3	100.0
XLoadQueryFont	88.6	61.2	27.4	31.0
XSetStandardProperties	14.6	2.0	12.6	86.3
XCreateSimpleWindow	6.6	0.0	6.6	100.0
XSetWMHints	5.9	0.0	5.9	100.0
XCreateGC	4.3	0.0	4.3	100.0
XSelectInput	4.6	0.0	4.6	100.0
XMapWindow	3.9	0.0	3.9	100.0
XClearWindow	3.6	0.0	3.6	100.0
XDrawRectangle	3.1	0.0	3.1	100.0
XDrawLine	2.3	0.0	2.3	100.0
XDrawString	2.9	0.0	2.9	100.0

$$Percentage = (Adatime - Ctime)/Adatime$$

Table 1: Averages were taken over 6 runs of both the *Ada* and *C* code. Times were taken for 10 calls to each of the above routines (results here are thus divided by 10) with the exception of `XOpenDisplay`.

running. The times reported are for a sample run of the Petri net editor, during which the following actions were performed:

- Create a place.
- Add a token to the place.
- Create a transition.
- Add an input arc between from the place to the transition.
- Move the transition.
- Fire the transition.
- Delete the transition.
- Delete the place.
- Click on the close button to end.

As a result of these profiles, we discovered that a large amount of time (about 25% of the total elapsed time) was spent in the menus package – most likely a result of its multiple polling loops checking for events. The multiple loops were necessary because of a bug in an early version of the *X Window System* which caused certain mouse events to be received out of order. We are now trying to find a more efficient implementation of this package which combines these multiple polling loops into one. The polling loop in the Mouse module also has a high time, but this is to be expected since this includes the time spent waiting for user events. Most of the other routines in the table have a number of calls to X in them, which easily accounts for their high times (e.g. `Create_Window`, `Draw_A_Polygon`, `Create_Batch_Windows`). Other routines were called so often that the times added up (e.g., `Find`, `Search`, `Parent_Block`, `Calculate_Coords`).

In addition, these profiles revealed a potential deficiency in locking mechanism for the abstract depiction. Recall that since the UIMS and the application execute concurrently, some form of access control is needed for the abstract depiction structure. For instance, the renderer could be reading the abstract depiction tree to update the display, while the artist is writing into the structure to create a new figure. Currently, *Chiron-0* employs a very coarse-grained scheme that locks the entire tree when it is being read (multiple readers are allowed) or written. The profiles showed that during initialization, the renderer takes out a lock to write in the abstract depiction. Meanwhile, the initialization routine is trying to create a new figure (which requires writing a new node into the abstract depiction tree). The initialization routine is forced to wait until the renderer completes its write, even though it needs to access a completely different part of the abstract depiction tree. This artificial serialization does not cause a slowdown on a uniprocessor, but for multiprocessor machines it may have a noticeable impact. A new version of the access controller is being considered that will lock only subtrees, rather than the entire abstract depiction tree, in order to allow more concurrent accesses to the structure.

Petri Net Editor Profile				
Module	Subprogram	Elapsed time (sec)	CPU time (sec)	Calls
Abstract Depiction:	Recalc_Placement	1.120	1.020	12
	Parent_Block	1.060	0.600	116
	Calculate_Coords	0.880	0.640	44
	Figure_Extent	0.800	0.560	17
Access Controller:	Seize_Write	7.680	2.960	29
Hash Table:	Find	3.920	2.840	245
	Search	2.500	2.040	279
Menus:	Show_Menu	21.160	13.000	7
	Create_Batch_Windows	1.520	0.740	3
Mouse:	poll loop	57.480	11.760	74
	Sub_Window	1.500	1.160	195
	Find_Figure	1.420	1.360	100
Picture Manager:	Initialize	5.020	0.580	1
	Delete_Figure	1.520	0.620	5
	Change_Polygon	1.540	0.740	2
	Corner_Index	1.220	0.720	22
Renderer:	Add_View	5.480	1.980	1
	Change_Block	1.640	1.160	21
	Deposit	5.380	1.340	23
	Determine_Clip	0.620	0.580	8
	Draw_Polygon	3.540	1.560	29
	Draw_Text	2.240	1.000	12
	Enter_Block	0.740	0.580	12
	Update_Screen	0.960	0.500	17
	Add_Figure	2.220	0.920	9
	Bye_Figure	1.380	0.500	5
	Create_Window	3.640	1.380	1
	Draw_A_Polygon	1.340	0.780	17
	Total	90.960	24.060	482

Table 2: All subprograms which had a cpu time of greater than 0.5 seconds (over all calls) are included. The figures for the total is representative of the time it took to run a complete test – it is not the sum of the numbers listed here.

In addition to the efforts described above, an improved refresh scheme is being considered. There is a one-to-one mapping between each figure in the abstract depiction and the bitmap which depicts it on the display. In current implementation, none of these bitmaps are stored – an object is completely redrawn each time the screen is updated. In order to speed up these refreshes a bitmap cache can be implemented by storing bitmaps with their corresponding figures in the abstract depiction. When a figure is created or modified, a new bitmap is created and cached before being mapped to the display. Subsequent screen updates that do not alter the appearance of the figure (such as a move or refresh) can use a cached bitmap and should be much faster as a result. As bitmaps consume a large amount of storage space, having an infinite cache size (i.e., cache every figure's bitmap) is not realistic. We need to investigate the tradeoffs among various bitmap cache sizes and cache replacement strategies. For example, only complex and commonly used bitmaps may be made eligible for caching. If a least-recently-used replacement strategy is adopted the most commonly used bitmaps will tend to remain in the cache. The size of the cache and other parameters can be varied dynamically depending on such factors as system load and available memory.

Finally, we are also examining different *Ada* compilers. One of the significant differences between the compilers available when *Chiron-0* was first written and the current generation is in tasking optimization. Some of *Chiron-0*'s many tasks can be compiled into simple monitors by smarter compilers [HN80]. This may have a large impact on the tasking overhead, and will certainly affect the design of the tasking structure in *Chiron-1*. Also, more profiling tools are being made available with the newer compilers, making early detection of bottlenecks much easier.

5.3 Improvements to the Design

The previous two sections described the problems found in the design and implementation of *Chiron-0*. The solutions presented in this discussion are being applied in the current redesign effort into *Chiron-1*. Our experience with the prototype also pointed out several improvements and additions that could be made to the current design, which are also planned for *Chiron-1*.

5.3.1 Limited Event Handling

Chiron-0's event handling mechanism is very simple. For each data type, the artist passes an event-handling procedure to the UIMS that is executed whenever any event for an object of that type is received. It would be desirable for an artist to be able to specify classes of events in which an annotated type is interested (i.e., button clicks, mouse motion) and have the UIMS separate out the rest. Without this facility, unnecessary calls are made to the artist's event-handler, and the artist is forced to perform extra processing to filter

out the unwanted events itself. Furthermore, there is no mechanism for an artist to put an event back in the queue, or to pass it on to another artist. This capability is useful when an operation requires the cooperation of several artists (for example, a group delete operation that deletes different types of objects).

A simple solution to these problems, which is being incorporated into the current redesign, is for the artist to pass a list of interesting events along with its event-handling procedure. The mapping task in the devices module (section 3.3) can easily store this information in a table. The relay task will check each event received against the table and only notify the artist when necessary.

5.3.2 Multiple Artists

Currently only one artist can be attached to a single abstract data type at a time (this is a result of the current implementation – multiple artists are conceptually possible in the framework of the design). Several artists can be attached to a single abstract data type, by further extending the mapping task's duties. This makes it possible to display multiple, coordinated views of a single abstract data type at the same time. Each artist that wants notification of events for objects of the abstract data type can pass the mapping task an event-handling procedure, which it stores in a table. The relay task can then query the mapping task for the list of artist procedures to call. Multiple artists are also consistent with the dispatcher mechanism proposed as a partial solution to the imperfect simulation of annotation in *Ada*. Instead of informing a single artist of each abstract data type operation, the dispatcher can notify all interested artists.

5.3.3 Artists are Difficult to Implement

A common complaint of users of *Chiron-0* is that artists are difficult to write. The programmer needs some understanding of *Chiron-0*'s device model in order to use event notification mechanism, and must be familiar with interface to *Chiron-0* provided by the picture manager. Even though all the routines the artist writer needs are isolated in this one package, the interface is still quite lengthy and complex. Furthermore, the necessary duplication of abstract data type state in the artist (discussed above in section 5.2.2) forces the artist writer to reimplement much of the original data structures. As a result, writing an artist is a significant undertaking for a single programmer, and yet does not lend itself well to sub-division so that the project can be worked on by a team.

A possible remedy for this problem is to provide artist builders with an *artist generator* – a graphical system that automatically generates artist code. An artist generator for *Chiron-0* should be largely dialogue and menu-driven. It must provide the artist builder with the capability to draw the graphical representation of an abstract data type on the screen, and then generate the proper artist code. In addition, the artist generator can

maintain a library of reusable graphical components (e.g. buttons, titlebars) which would further simplify user interface construction, as well as foster reuse and uniformity.

A number of current user interface management systems include editors for specifying the presentation and functional behavior of their user interfaces. Existing systems include *Prototyper* for the Macintosh, and *Interface Builder* for the NeXT Computer. Several recent projects, such as *Serpent* [Ins89], *ET++* [WGM88], and *InterViews* [LVC89], are currently developing graphical specification tools and generators.

5.4 Client-Server Split

In the current implementation the tool and artist are linked together into a single executable process. This results in very large run-time processes (partly because of the use of *Ada*), and the entire system must be reloaded (and modules possibly recompiled) whenever any code is modified. Furthermore, tools written in languages other than *Ada* cannot be accommodated without some kind of language-dependent interface.

One possible solution to these drawbacks calls for a fundamental change to *Chiron-0*'s architecture: split the UIMS into a client and a server. The server implements the internals of the user interface management system – essentially *Chiron-0*'s abstract depiction, concrete depiction, and devices tasks. The tools and artists are on the client side, which provides a data-shipping interface to communicate with the server. The advantages of such a client/server architecture are numerous:

- Tools can be built in different (non-*Ada*) languages. As long as the tools use the correct communication protocol they can make make requests of the *Chiron-0* server, and are treated as first-class applications.

- Separation of the server from the client code leads to smaller processes, as well as the ability to modify one without having to relink and possibly recompile the other. Only the client side is loaded with a tool, thus reducing the overhead of experimenting with various applications.

- The *Chiron-0* server and the display can run on a separate machine from the client process. The system configuration can be tuned based on such factors as machine load and network traffic.

- Multi-display or multi-client configurations are supported. A single client can display results on several different machines, or a single display could handle input from multiple clients.

- The heavyweight process model of *Unix* provides protection boundaries between processes, whereas the lightweight model of *Ada* tasks does not. By making the client and server separate *Unix* processes, *Chiron-0* will have a useful boundary between

client and server processes (e.g., clients can't destroy UIMS data structures), without paying the overhead of heavyweight protection boundaries between all tasks. The heavyweight process management may also be helpful in maintaining housekeeping details, such as noticing that a client has terminated.

6 Conclusions

The goal of the *Chiron* project is to develop organizing principles for UIMS that promote uniformity without compromising power or extensibility. The design of *Chiron* presented in this paper is a viable approach for meeting these goals, and is applicable to software environments in general.

Chiron's decoupling of interaction and tool functionality provides the benefits of modularity, and allows independent development of interface and tool components. Both interface and tool components can be reused, encouraging uniformity and reducing the time to build new applications. This separation of concerns is accomplished in a powerful, flexible manner by binding artists to abstract data types with the annotation mechanism.

Chiron does not limit the types of tools available by imposing a dispatch model of control. The tools run concurrently with the interface facilities: the UIMS can continue to process inputs and update the display while the tool responds to user commands. Within *Chiron-0*, the renderer runs concurrently with the abstract depiction, allowing requests on the abstract depiction (e.g. create figure, move figure) to proceed at the same time as display updates.

Chiron's abstract depiction provides a structured representation between the model objects and view objects – no particular representation scheme (such as parse trees) is forced upon the applications. The abstract depiction's hierarchical structure provides the means to build complex representations from a small set of primitive figures. Also, such functionality as input correlation and incremental update is facilitated.

The lessons we learned from building *Chiron-0* are shaping the design and implementation of *Chiron-1*. We discovered several limitations resulting from our implementation – most notably the imperfect simulation of annotation and disappointing performance. There also number improvements and additions to this design that are planned for *Chiron-1*. These include an extended event handling mechanism, artist generators, and the client/server split. In addition, we are investigating a scheme for the persistent storage of views using *P-Graphite* [WWFT88]. Lastly, we are also developing an object-oriented *Abstract Depiction Language* to simplify the definition of graphical objects and their relationships (this information is stored in the abstract depiction).

Acknowledgements We are grateful to the many people within the Arcadia consortium that have contributed to the *Chiron* project. At UC Irvine, Dennis Troup was involved

in the design from early on, contributed to the *Chiron-0* implementation, and is now working on *Chiron-1*. Cheryl Kelly and Craig Snider also contributed to the design and implementation when it was still in its early stages. Greg Bolcer, Mary Cameron, Greg James, and Reudi Keller are currently working on the *Chiron-1* redesign. We also wish to thank the following Arcadia members that have contributed to *Chiron*: Ray Klefstad, Stephen Sykes, Rick Selby, Izhar Shy, Kari Forester, and Adam Porter.

References

[AC85] Inc. Apple Computer. *Inside Macintosh*. Addison Wesley, Reading, Mass., 1985.

[Ado85] Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley, Reading, Massachussetts, 1985.

[ALR83] American National Standards Institute. *Military Standard Ada Programming Language (ANSI/MIL-STD-1815A-1983)*. January 1983.

[Ans82] Ed Anson. The device model of interaction. *Computer Graphics*, 16(3):107–114, July 1982. (Proceedings of SIGGRAPH 82).

[BEHtH82] Peter R. Bono, José L. Encarnação, F. Robert A. Hopgood, and Paul J. W. ten Hagen. GKS—the first graphics standard. *IEEE Computer Graphics and Applications*, 2(5):9–23, July 1982.

[BFS88] Deborah A. Baker, David A. Fisher, and Jonathan C. Shultis. The gardens of iris. Technical Report Arcadia-IncSys-88-03, Incremental Systems Corporation, August 1988. Draft.

[DYT88] Jennifer-Ann M. Durand, Michal Young, and Dennis B. Troup. A tool builder's guide to Chiron. Arcadia Technical Report UCI-88-18, University of California, Irvine, May 1988.

[GRs83] Adele Goldberg and David Robson. *Smalltalk–80: The Language And Its Implementation*. Addison-Wesley, 1983.

[HN80] A.N. Habermann and I.R. Nassi. Efficient implementation of Ada tasks. Technical Report CMU-CS-80-103, Carnegie Mellon University, 1980.

[Ins89] CMU Software Engineering Institute. Serpent overview. Technical Report CMU/SEI-89-UG-2, Carnegie Mellon University, April 1989.

[LH83] David A. Lamb and Paul N. Hilfinger. Simulation of procedure variables using Ada tasks. *IEEE Transactions on Software Engineering*. SE-9(1):13–15. January 1983.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with interViews. *IEEE Computer*, 22(2):8–22, February 1989.

[MvD78] James C. Michener and Andries von Dam. A functional overview of the Core system with glossary. *ACM Computing Surveys*, 10(4):381–387. December 1978.

[Mye83] Brad A. Myers. Incense: A system for displaying data structures. *Computer Graphics*. 17(3):115–125. July 1983.

[NeW87] Sun Microsystems. Inc.. Mountain View, California. *NeWS Technical Overview*. 1987.

[Rou85] O. Roubine. Programming large and flexible systems in Ada. In John G. P. Barnes and Gerald A. Fisher, Jr.. editors. *Ada in Use: Proceedings of the Ada International Conference*, volume 5, pages 197–209. Paris, May 1985. Association for Computing Machinery and Ada-Europe, Cambridge University Press.

[SB86] Mark Stefik and Daniel Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*. 6(4):40–62, Winter 1986.

[SBK86] Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3(1):10–18. January 1986.

[SG86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, April 1986. Actually appeared June 1987.

[SIK+82] David Canfield Smith, Charles Irby. Ralph Kimball. Bill Verplank, and Eric Harslem. Designing the Star user interface. *BYTE Magazine*, 7(4):242–282, April 1982.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Menlo Park. California, 1986.

[Sun86] Sun Microsystems, Inc.. Mountain View, California. *SunView Programmer's Guide*, February 1986.

[TBC+88] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia environment architecture. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 1–13, Boston, November 1988. Appeared as *Sigplan Notices 24*(2) and *Software Engineering Notes 13*(5).

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++ - an object-oriented application framework in C++. In *Object Oriented Programming Systems, Languages and Applications '88 Conference Proceedings*, pages 46–57, September 1988.

[WWFT88] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, Boston, November 1988.

[Xtk88] Massachusetts Institute of Technology and Digital Equipment Corp., Cambridge and Maynard, Massachusetts. *X Window System, X Version 11, Release 3*, October 1988.

[YTT88] Michal Young, Richard N. Taylor, and Dennis B. Troup. Software environment architectures and user interface facilities. *IEEE Transactions on Software Engineering*, 14(6):697–708, June 1988.