Supporting Separations of Concerns and Concurrency in the Chiron-1 User Interface System*

Richard N. Taylor
Kari A. Nies
Gregory Alan Bolcer
Craig A. MacFarlane
Gregory F. Johnson**
Kenneth M. Anderson

UCI Technical Report 94–12

Department of Information and Computer Science
University of California
Irvine, California 92717-3425†

**Northrop Corporation
Pico Rivera, California

{taylor, kari, gbolcer, craigm, greg, kanderso}@ics.uci.edu

March 11, 1994

*This paper is a major revision and expansion of "Separations of Concerns in the Chiron-1 User Interface Development and Management System" which appeared in the Proceedings of InterCHI'93 [TJ93].

†This material is based upon work sponsored by the Advanced Research Projects Agency under Grant Number MDA972-91-J-1010. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Abstract

The Chiron-1 user interface system demonstrates key techniques which enable a strict separation of an application from its user interface. These techniques include separating the control flow aspects of the application and user interface: they are concurrent and may contain many threads. Chiron also separates windowing and look-and-feel issues from dialog and abstract presentation decisions via mechanisms employing a client-server architecture.

To separate application code from user interface code, user interface agents called *artists* are attached to instances of application abstract data types (ADTs). Operations on ADTs within the application implicitly trigger user interface activities within the artists. Multiple artists can be attached ADTs, providing multiple views and alternative forms of access and manipulation by either a single user or by multiple users. Each artist and the application run in separate threads of control.

Artists maintain the user interface by making remote calls to an abstract depiction hierarchy in the Chiron server, insulating the UI code from the specifics of particular windowing systems and toolkits. The Chiron server and clients execute in separate processes. The client-server architecture also supports multi-lingual systems: mechanisms are demonstrated which support clients written in languages other than that of the server, while nevertheless supporting object-oriented server concepts. The system has been used in several universities and research and development projects. It is available by anonymous ftp.

Keywords: User interface systems, concurrency, separations of concerns, software architectures, GUI development, multi-lingual systems.

Contents

1 Overview

The development of user interfaces for large applications is subject to a series of well-known problems including cost, maintainability, and sensitivity to changes in the operating environment. The objective of the Chiron project is to address these software engineering concerns by creating a user interface technology with the following properties:

- architectural flexibility and extensibility
- robustness in the presence of change
- facilitation of software reuse

The Chiron project has focused its research in the area of user interface and applications architectures. We believe that current, widely-accepted user interface architectures are too constraining for many applications. They often do not exploit or even support concurrency, and they tend to view the user interface as the center of the universe. Current user interface architectures are also generally tied to a single programming language or even a single development environment. They are often not easily reconfigurable and focus little or no attention on issues of software reuse. Our objective has been to develop user interface and applications architectures that address software engineering concerns – to take the best in user interface technology and offer an environment in which it is relatively easy to develop and maintain user interfaces for new and pre-existing applications. We believe that our focus on software engineering concerns in the user interface domain is distinctive.

In this section we discuss the objectives for the Chiron project and briefly indicate key constituents of the Chiron system. The domain of discourse is user interface development and run-time management architectures and how they relate to application software architectures. Our perspective is that of researchers and developers of user interface systems; our intended audience is other researchers and developers, as well as system architects.

1.1 Architecture overview

To be clear in our brief description of the Chiron architecture, we provide a few definitions. An *application* is the software for which Chiron is being used to provide an interface to the user. A *user interface* consists of presentations (abstract graphics) plus dialog decisions (association of behavior with events or actions). For our purposes, specification of a user interface does *not* determine look-and-feel (concrete graphics). *Process* in the ensuing discussion refers to an operating system process. We also use the term *agent* to refer to software components that have their own thread of control. Multiple agents may reside in a single process.

Chiron employs a client-server architecture. The server provides clients an interface to an extensible, object-oriented class hierarchy of graphical objects called the *abstract depiction hierarchy.* The Chiron server maintains structured graphics model of the user interface managed by each client, called the abstract depiction, and oversees the rendering of the abstract depiction to a concrete depiction, using a windowing system and toolkit. In addition, the server listens for events from the underlying windowing system, interprets these events, and forwards to the appropriate client those that it cannot fully handle locally.

A Chiron client encapsulates an application and its user interface. To separate application code from user interface code, user interface agents called *artists* [1] are attached to instances of objects within the application. An artist encapsulates presentation and dialogue decisions; they are enacted by making calls to a Chiron server.

The frame of reference for communication between the artists and the application are events (operations) on objects within the application. Operations on objects implicitly trigger user interface activities within the artists. The triggering is performed by listening agents associated with the objects. A dispatching mechanism responds to the triggers, and may convey the event to one or many artists. Likewise, artists may receive events from one or many objects.

1.2 Goals

The following paragraphs elaborate, in turn, each of our software engineering design properties and briefly discuss how the Chiron architecture achieves these goals.

Architectural extensibility and flexibility. Current user interface technologies often impose a fairly rigid set of constraints on how the application must be structured. In a large application, however, there are typically many architectural desiderata, so it is undesirable for the user interface to enforce a particular architecture.

Chiron provides architectural flexibility at two levels: within application processes and across process and machine boundaries. Inside an application process, the Chiron dispatching mechanism provides an unobtrusive interface between the application and the user interface layer. The application is not built as a set of callback routines to the user interface. Across process boundaries, Chiron's client-server architecture provides flexibility in terms of application languages, windowing systems and toolkits, and process inter-connection topology.

Robustness in the presence of change. In a system that has distinct groups of concerns intermingled, it becomes difficult or impossible to manage changes in the environment, since changes to one part of such a system inevitably mandate changes in other, unrelated parts of the system. In the worst case one is required to do an amount of work proportional to the size of the entire system whenever a change is necessary.

Chiron provides separation of concerns at two levels: between the application and the user interface software, and between the user interface software and the underlying windowing system and toolkit.

Reusable software artifacts. A key to software reuse is to structure systems so that they use modular components with focused, well-defined interfaces.

The Chiron notion of artists adheres to this principle. Each artist is a stand-alone, software agent that implements a particular user interface and has a small, well-defined interface to the application and to the server. The artist notion facilitates software reuse

[1] The term originated with the Incense system [Mye83]. We use the term to apply to a concept which is similar in purpose, but more general, powerful, and complex.

by factoring out user interface code from the application code, allowing for separately maintainable and reusable entities. The reuse issue is also addressed by the design of the abstract depiction hierarchy. A programmer can make use of the standard class hierarchy of user interface objects, or can create new, tailored subclasses.

1.3 Specific objectives

Turning from the general software engineering goals above to more specific issues, Chiron was designed with the following objectives in mind:

- Support for multiple graphical and textual views, with coordinated update. Multiple coordinated views should be available to either a single user or groups of users.
- Support for concurrency, both within processes and across machine and network boundaries. The application and user interface should have their own threads of control; neither should be in absolute control.
- Performance comparable to standard user interface development systems and *ad hoc*, hand-crafted systems.
- Support for multiple look and feels without recoding, recompiling, or reloading.
- An extensible library of graphical objects.
- Support for dynamic changes to the set of active views as the application executes.
- Support for clients written in multiple languages.

The techniques used to achieve these objectives are addressed in detail throughout the remainder of this paper. Although other user interface development systems have achieved several of these objectives, our goal has been to simultaneously achieve all of them, not to excel in one area only. We believe the exploitation of concurrency, an unrestricted application architecture, and a multi-lingual design are all aspects that set Chiron apart from existing user interface development systems.

1.4 Example client

Figure 1 shows a simple flight simulator instrument panel built with Chiron. There are several points of interest in this example.

Coordinated multiple views. The display shows how multiple artists can be attached to the same object to provide alternate views. For example, the Airspeed artist and the Speed Indicator artist are both bound to the speed of the aircraft simulator, the display of one using an analog gauge and the other a digital readout. The Artificial Horizon artist is bound to both the pitch and the roll objects of the aircraft simulator, presenting a synthesis of their information. Separate artists for the pitch and roll objects are also present.

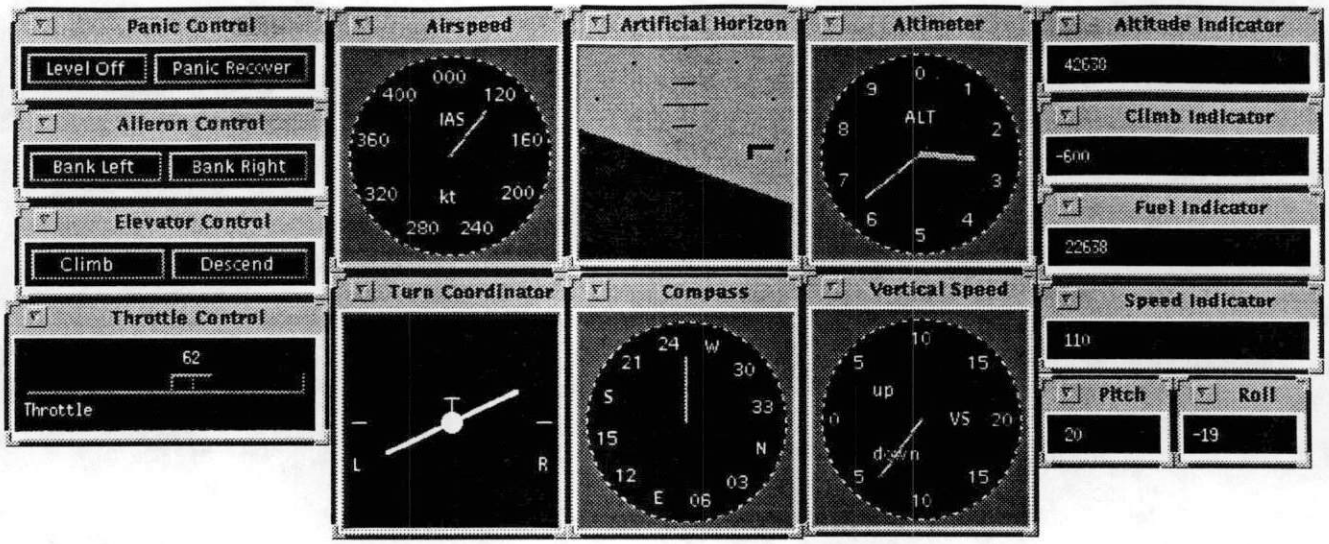

Figure 1: Flight simulator instrument panel. Each of the windows is associated with a separate artist. The application manages a simple set of flight laws, and continually updates the six degrees of freedom of the simulated aircraft. Artists giving various presentations of the vehicle's state are attached to the objects that define the six degrees of freedom of the simulated vehicle.

Concurrency. There is a symmetry between the degree of control exercised by the application and the user. The application is active, in that it continually re-computes the flight laws and updates the aircraft's state vector while the user can concurrently manipulate the controls. The application and the user interface execute concurrently, coordinating through an event-based mechanism. With conventional user interface technologies, at any given point in time either the application is dormant and a user interface listen/dispatch routine is in control, or the application is active and the system does not respond to user interactions.

Multiple Look and Feels. The same application could have a different look and feel by simply communicating with a Chiron server that uses a different implementation of the abstract depiction hierarchy. There are no graphics libraries loaded with the client process.

Extensible Graphics Hierarchy. If we found that we were duplicating graphics (i.e. a dial readout) for a particular domain, we could extend the abstract depiction hierarchy to include a new class. This would improve efficiency by eliminating calls to the server, simplifying artists, and ensuring consistency for graphics within the same domain.

Reconfigurable Views. We can easily change the set of active views of the state of the aircraft. Artists can be added or omitted and multiple artists can be invoked on the

same or on different displays, without recoding, recompiling or reloading the application. Certain aspects of the client configuration are specified in an external file that is dynamically interpreted at run-time.

1.5 Organization of the paper

The remaining sections of this paper are as follows. We begin with a detailed description of the Chiron architecture in Section 2. Section 3 describes user interface development with Chiron and Section 4 describes our limited, but promising, experience in supporting multilingual clients. Section 5 discusses the performance of the system. We then consider some relationships between Chiron and other relevant work in the user interface area in Section 6, followed by some example uses of the system in Section 7. A summary and conclusion section closes.

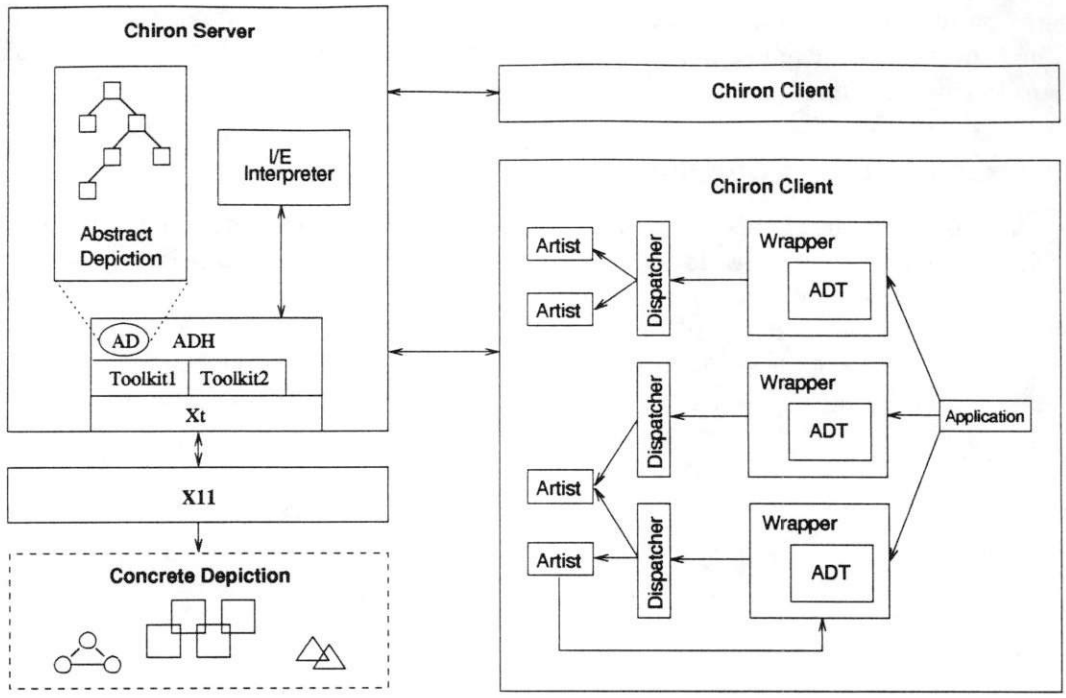

Figure 2: Chiron-1 conceptual architecture.

2 Chiron architecture

Section 1 provided a very brief description of the Chiron architecture in order to familiarize the reader with the basic mechanisms used to achieve our goals. In this section we elaborate on that description. Figure 2 shows one possible configuration of the client and server components. Before discussing the client and server architectures in detail, we attempt to familiarize the reader with the overall architecture by tracing the event flow of the system.

Any object whose state is to be depicted by an artist must be defined as an instance of an *abstract data type* (ADT). This means that any creation, query, change, or destruction of an object must occur via a call to the ADT's interface. [2] A listening agent, called a *wrapper*, is attached to each ADT that exports an interface identical to that of the ADT. The *application* makes ADT calls indirectly through the wrapper, and otherwise remains unchanged. The wrapper reifies ADT operations into events that are forwarded to a *dispatcher*. The dispatcher then forwards the event to any artists interested in the specific event.

An *artist* is an active drawing agent for an ADT. It encapsulates the presentation and dialogue decisions for a particular graphical representation of the ADT. When an artist is notified of an ADT operation, it may update its depiction to reflect the state change within the ADT. Ensuing calls by the artists to create/update depictions are transmitted over an

[2] A relaxation of this assumptions is discussed on page 16.

inter-process communication link to a Chiron server.

The server implements the inheritance hierarchy of abstract depiction classes (*ADH*). Calls from artists create and manipulate instances of these classes; the server thus maintains an 'abstract depiction' of each artist's user interface. The server also oversees rendering of the abstract depiction to a concrete depiction, using a window system and toolkit. The *instruction/event (I/E) interpreter* mediates the flow of ADH instructions from clients and window events from the underlying graphical substrates.

When the user generates a windowing event, e.g. a menu selection, it is detected by the server, converted to a Chiron event, and potentially sent back to the client where it will be routed to the artist that created the object of the event, in this case, the menu. The artist might handle the event by changing the state of an object within the application. This would be done, as in the application, through a call to an ADT's wrapper – thus any artist concerned with this state change will be notified. Note therefore that application objects are modified by the user and the application program in the exactly the same manner.

The remainder of this section elaborates on this architecture. The key run-time components of both the client and server architecture are discussed in detail in order to elucidate how the architecture satisfies some of the specific objectives of the project. Section 3 describes the process of creating a Chiron client and the tools available to aid in that activity.

2.1 Client architecture

This section describes the architectural components that make up the run-time architecture of a Chiron client. This architecture is illustrated in Figure 3.

Chiron applications. The *application* is the body of code for which the client is providing a user interface. It makes calls to ADTs indirectly through an identical interface provided by the wrapper. It must also `with` [3] [4] the client initializer package. Otherwise the application remains unchanged. An application executes in its own thread of control. It is not segregated into a collection of callback routines that are driven by the user interface, nor does it directly control the execution of the user interface code.

The Chiron dispatching architecture. Chiron's dispatching mechanism enables the separation of user interface and application. This mechanism is implemented primarily by two components: the wrapper and the dispatcher.

The *wrapper* plays the role of listener and notifier of ADT events. It listens for ADT calls and relays information about them to the client dispatcher in the form of a client event. The wrapper exports an unobtrusive interface to the ADT so that the tool and artists can make calls into it just as they would directly to the ADT. There is one wrapper generated for each depicted ADT.

[3] In Ada, a `with` clause is used to provide visibility to library units. The application does not make calls to the client initializer package; the elaboration of the client initializer, enabled by the `with` clause, will execute the required initialization code.

[4] Although it is possible to construct artists in additional languages, Chiron-1 currently supports only Ada [ALR83] clients. The underlying architecture and component generators have not yet been reimplemented to support other languages, although some preliminary work in that area is reported in Section 4.

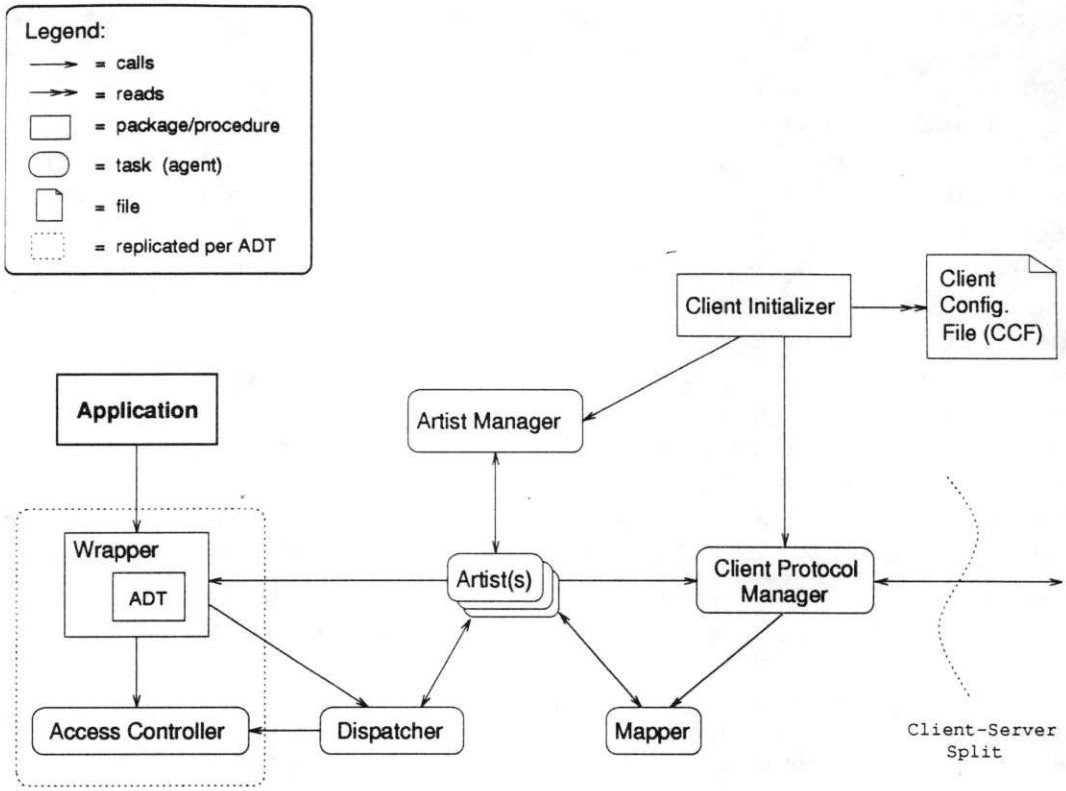

Figure 3: The client run-time architecture

The client *dispatcher* routes client events from wrappers to the appropriate artists. Artists register directly with a dispatcher for notification of specific events and a dispatcher only passes events to artists that have registered for that event. Dynamic registration and de-registration is supported. There may be a single centralized dispatcher or dedicated dispatchers per ADT. This mechanism is conceptually similar to broadcast message systems such as Field [Rei90] and SoftBench [Cag90], though the actual mechanisms employed are quite different. The differences will become apparent in the following discussion.

It is also this mechanism that supports coordinated, multiple views of objects. Multiple artists may register for the same ADT events, allowing them to simultaneously reflect the current state of the ADT. If an artist modifies an ADT using the wrapper, all other listening artists will automatically be notified of the client event. Figure 4 shows the dispatching architecture of the flight simulator application depicted in Figure 1. The top four artists are pilot control artists and correspond to the leftmost artists in the flight simulator depiction. Note that these artists modify the state of ADTs in response to interactions with the user. This is done through calls to the corresponding ADT wrapper. This architecture uses one dedicated dispatcher per ADT. Note that several artists may be registered for events with the same ADT. Four separate artists, for example, monitor the various state aspects of the Altitude Module ADT. The Artificial Horizon Artist registers for events on both the pitch

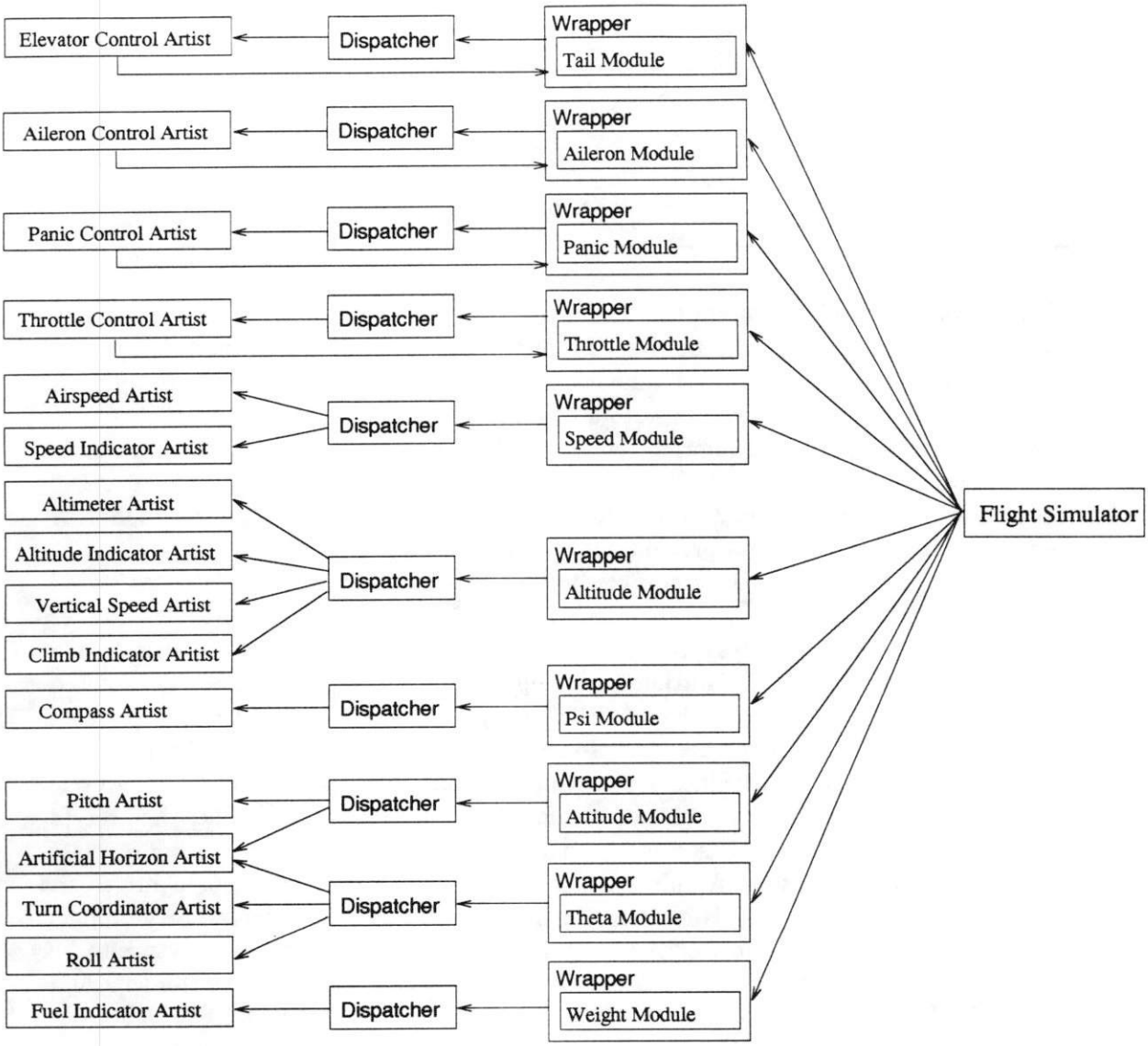

Figure 4: Flight simulator dispatching architecture

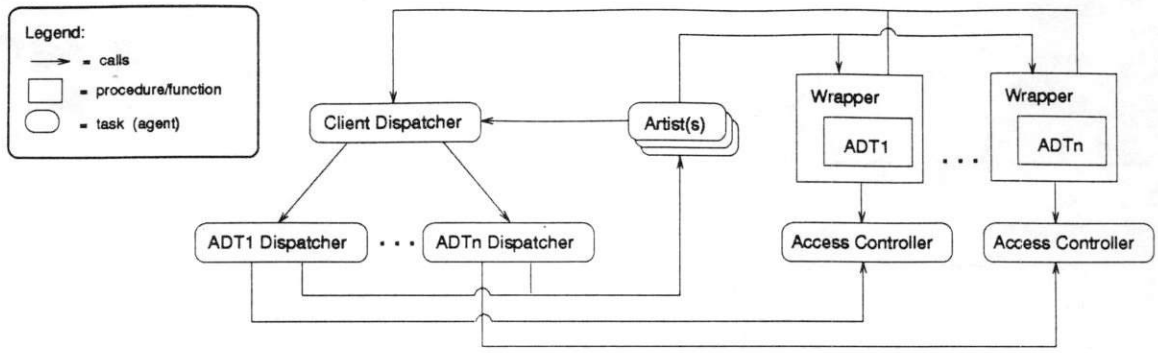

Figure 5: Chiron 1.4 dispatching architecture

and the roll of the simulated aircraft.

Because multiple artists may be invoking operations on the ADT in addition to the application, it is necessary to employ a means of access control on the ADT. Also, when an artist updates its depiction in response to a change to ADT state, it will sometimes need to query the ADT state if the information encoded in the client event is not sufficient to enable the artist to update the depiction. For this reason, it is necessary that any ADT write (or state modifying) operation be blocked until all artists have completed updating their depictions in response to a previous state change.

ADT locking is handled by the *access controller* component. The access controller is used by the wrapper to ensure concurrent-read, exclusive-write access to an ADT. The wrapper must obtain a lock from the access controller before accessing the ADT. In addition, when a write operation is performed all additional write operations are blocked until each interested artist has completed updating its depiction. Update completion is detected and signaled by the dispatcher. There is one access controller generated for each depicted ADT.

The default architecture provides one centralized *client dispatcher* per client. However, it is possible to generate dedicated dispatchers for ADTs. The client dispatcher then serves as a router that directs ADT-based events and registration requests to the appropriate ADT dispatcher for processing. An illustration of the resulting run-time dispatching architecture is given in Figure 5. This ensures a consistent dispatching interface for wrappers (for notification) and artists (for registration), allowing the dispatching architecture to be modified without requiring any changes to wrappers or artists. The client dispatcher and dedicated ADT dispatchers share the same interface, so it is possible to optimize a client by circumventing the client dispatcher completely (as seen in Figure 4). This requires a trivial hand-editing of the wrapper and artist code that can be done once an optimal dispatching architecture is determined. Informal experience indicates that ADTs that have a high event bandwidth should be given a dedicated dispatcher. For ADTs with low event bandwidth, it may be optimal to use the centralized client dispatcher rather than to incur the extra tasking overhead of a dedicated dispatcher.

Artists. *Artists* encapsulate the dialogue and presentation aspects of the user interface.

An artist may depict the state of zero or more ADTs, or it may be purely interactive and not bound to any ADT. An artist has three main responsibilities: to create an initial graphical depiction, to respond to ADT (client) events, and to respond to user-generated (server) events. Unlike model-based user interfaces such as ITS [WBB+90], UIDE [SFG93], and HUMANOID [SLN93], Chiron artists use a programmatic as opposed to a declarative approach in defining the dialog and presentation of the user interface. This choice simply reflects the research emphasis of the project, and is not an inherent characteristic of the architecture.

An artist is defined as a task type, allowing multiple instances of an artist to be dynamically invoked and terminated. It also means that artists have their own threads of control, executing in parallel with each other and with the application. Since concurrency typically presents many new opportunities for creating buggy programs, we applied a concurrency analysis system, CATS, to Chiron. CATS performs a type of reachability analysis and checks (exhaustively) for deadlocks. The analysis found two race conditions and a deadlock, which we subsequently fixed.

An example of an artist specification for the throttle artist (depicted in Figure 1) is given in Figure 6.

The interface to all artists consists of four task entries: one to start up the artist, one to shut down the artist, one to accept notification of client events and another to accept notification of server events. For each artist within a client, an artist specification is generated along with a template for the artist body by the Chiron tool set. Further details are provided in section 3.

Artists create and manipulate their graphical depictions by making calls to the server's abstract depiction hierarchy. Artists respond to client and server events by registering handling routines within the artist that are automatically invoked when the artist is notified of an event. The definition of client events and server events differ, as does the registration mechanism.

Server events. Server events are user-generated events that are detected within the server such as a button press, a menu selection, or entering a value in a text field. More precisely, the server listens for X events, and reifies them to server events. Some X events are handled entirely within the server. Some are aggregated into a server event at a higher level of abstraction than the original X event. The object of an event may also be at a higher degree of abstraction. For example, if a new dial class were added to the hierarchy, the artist could receive select, move, and menu events on the composite dial object. The declaration of a Chiron server event is given in Figure 7. The Chiron server event type contains information about a server event that an artist might need in order to update its depiction or determine how the ADT state should be changed.

Within the client, the *mapper* routes server events to the appropriate artist, namely the artist that created the object on which the event occured. Artists register for server events in the following manner. Artists define behaviors (event handling procedures) for specific objects, or classes of objects, for particular server events. When the mapper receives a server event on an object, if the artist that created that object has registered a behavior for that object for the received event, then the mapper will notify the artist of the event, passing

```
with Client_Events;  use Client_Events;
with Chiron_Standard_Library;
with System;  use System;

package Throttle_Artist is

   package CSL renames Chiron_Standard_Library;


   task type Throttle_Artist is

      entry Start_Artist (
            ID   : CSL.Artist_ID_Type;
            Self_Aptr : address;
            Display_Name : CSL.Str);

      entry Notify_Client_Event (
            Client_Event    : Client_Events.Client_Event_Ptr;
            Handler_Routine : address);

      entry Notify_Server_Event (
            Object          : CSL.Object_Type;
            Server_Event    : CSL.Chiron_Event_Ptr;
            Handler_Routine : address);

      entry Terminate_Artist;

   end Throttle_Artist;


   type Throttle_Artist_Ptr is access Throttle_Artist;

end Throttle_Artist;
```

Figure 6: Throttle artist specification

it the object of the event, the server event, and the appropriate handling routine. The handling routine within the artist is then invoked, with the server event and the object as parameters. Behaviors for specific objects override behaviors for classes of objects. Artists may change behaviors dynamically by re-registering with the mapper.

Client events. A client event definition, unlike the server event definition, differs for each client. An initial client event definition is generated from the set of all operations on all depicted ADTs within a client. For example, given the ADT for the throttle module (Figure 8) of the flight simulator example, the definition is mapped onto a partial definition of a client event type in Figure 9. One client event kind is defined for each ADT operation. The event kind is a concatenation of the ADT name and operation name. For overloaded operations, a unique number is appended. The client event type defines a list of fields for each ADT operation. Each field corresponds to a parameter or return value for that operation. The client event mode is defined to specify whether an operation is a read or

```
type Chiron_Event_Kind is (Menu_Event,
                           Select_Event,
                           Adjust_Event,
                           Key_Event,
                           Move_Event,
                           Resize_Event);

type Chiron_Event_Type(Kind : Chiron_Event_Kind) is
  record
    Mouse_X  : integer;
    Mouse_Y  : integer;
    Time     : integer;
    Num_Val  : integer;
    Text_Val : Str;
    case Kind is
      when Menu_Event =>
        null;
      when Select_Event =>
        null;
      when Adjust_Event =>
        null;
      when Key_Event =>
        Key_Code : character;
      when Move_Event — Resize_Event =>
        Dest_X : integer;
        Dest_Y : integer;
    end case;
  end record;

type Chiron_Event_Ptr is access Chiron_Event_Type;
```

Figure 7: Server event definition

write operation on the ADT. This is used by the wrapper in order to ensure concurrent-read, exclusive-write access to the ADT. This information must be hand-specified by the client builder as it cannot be determined accurately without a good deal of semantic analysis which is outside of the domain of this project.

The assumption that all necessary client events can be captured by operations on the ADT can in some cases be too constraining. The ADT interface can usually be modified to compensate for any short comings, but sometimes this is undesirable. Also, some applications are simply not naturally object-based. Therefore, it is possible for client designers to define their own non-ADT events. This is done by simply editing the generated client events definition. For example, an application without well defined ADTs could be "seeded" with hand specified events. This is similar to what is done in the Zeus [Bro92] system for algorithm animation. Although it is still possible to support coordinated multiple views using this approach, as also demonstrated by Zeus, it is not possible to modify the application without consciously generating events for the user interface. Thus the separation between the application and the user interface has been compromised.

An artist handles a client event by first registering interest in the event with the client dispatcher. Registration includes specifying the handling routine for the event. When the

```
package Throttle_Module is

    Maximum_Throttle: constant := 1.000000;
    Minimum_Throttle: constant := 0.063000;

    procedure Adjust_Throttle (Delta_Value: in FLOAT);
    procedure Set_Throttle (New_Value: in FLOAT);

    function Get_Throttle return FLOAT;

end Throttle_Module;
```

Figure 8: Throttle_Module ADT specification

dispatcher is notified of a client event, it notifies all artists pre-registered for that event, passing them the client event, and a pointer to the handling routine. The handling routine is then invoked within the artist, passing it the client event. Artists may register and de-register for client events dynamically by making calls to the dispatcher.

Communication with the Server. The *client protocol manager* (CPM), handles communication to and from the Chiron server. This includes encoding/decoding messages into/from a lower-level communication protocol. Chiron uses Q [MOS90], a multi-lingual interprocess communications system built on top of XDR/RPC for client-server communication. ADH calls from an artist are translated to lower level calls to the CPM where they are encoded to Q protocol form and shipped to the Chiron server. The CPM also receives events from the server, decodes them, and forwards them to the mapper where they can be directed to the appropriate artist.

Initialization and dynamic reconfiguration. The *client initializer* is responsible for bringing up the initial client configuration. This consists of starting up the CPM task and invoking artist instances on specified machine displays. There is one client initializer generated per client.

Chiron supports the dynamic re-configuration of the set of active views (artists). The *artist manager* provides an interface through which new instances of pre-compiled artists can be invoked and shutdown at run-time. It is used by the client initializer for dynamic configuration and may also be used by an artist to invoke other artists. There is one artist manager generated per client.

The *client configuration file* (CCF) is used to specify the intended architecture of the client and is used by the client component generators which generate code according to the specified architecture. Additionally, the initial run-time configuration is specified in this file including which artists should be invoked at startup, how many instances of those artists, and on which machine displays. The client initializer will read the client configuration file at client initialization, allowing configurations to be modified without recompilation.

```
with Throttle_Module;  use Throttle_Module;
                              .
                              .
                              .
package Client_Events is

   type Client_Event_Kind is (
      Throttle_Module_Adjust_Throttle,
      Throttle_Module_Set_Throttle,
      Throttle_Module_Get_Throttle,
                              .
                              .
                              .
      Null_Event
   );

   type Client_Event_Type (Event_Kind : Client_Event_Kind := Null_Event) is
   record
      case Event_Kind is
         when Throttle_Module_Adjust_Throttle =>
            Throttle_Module_Adjust_Throttle_Delta_Value: FLOAT;
         when Throttle_Module_Set_Throttle =>
            Throttle_Module_Set_Throttle_New_Value: FLOAT;
         when Throttle_Module_Get_Throttle =>
            Throttle_Module_Get_Throttle_Result: FLOAT;
                              .
                              .
                              .
      Null_Event
   );
                              .
                              .
                              .
   type Client_Event_Ptr is access Client_Event_Type;
   Client_Event_Mode : constant array (Client_Event_Kind)
                              of Client_Event_Modes := (
      Throttle_Module_Adjust_Throttle => Write,
      Throttle_Module_Set_Throttle => Write,
      Throttle_Module_Get_Throttle => Read,
                              .
                              .
                              .
      Null_Event => None
   );

   procedure Free_Client_Event (Client_Event : in out Client_Event_Ptr);

end Client_Events;
```

Figure 9: Partial client event definition for flight simulator example

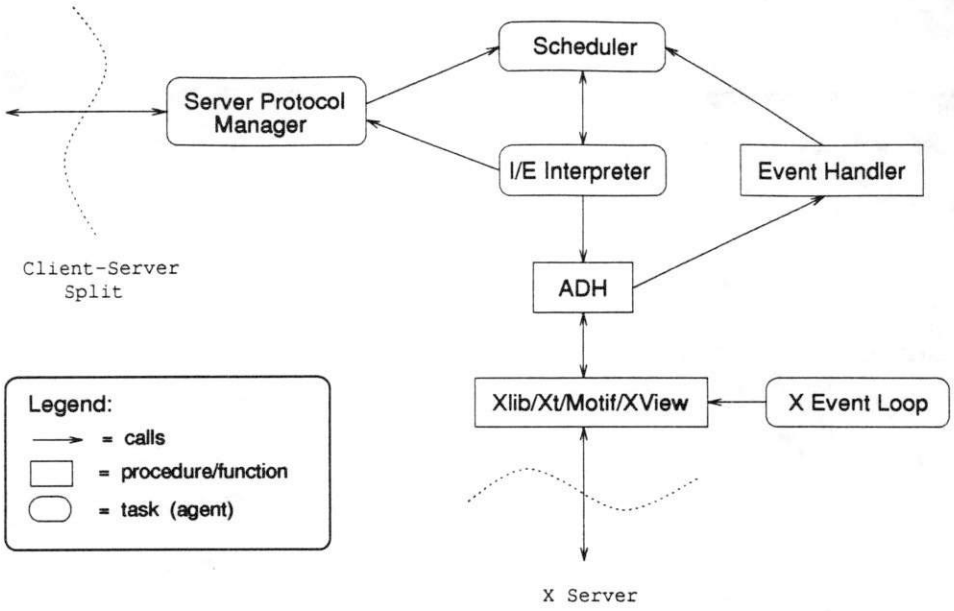

Figure 10: Chiron-1 Runtime Server Architecture

2.2 Server architecture

The server maintains an structured graphics model, or abstract depiction, of the graphical contents of each window managed by Chiron. This abstract depiction simplifies and abstracts the details of low level user interface libraries while adding concurrency, distributed processing, reuse, and insulation from changes in substrate technology. Concurrency allows the server to receive, schedule, execute instructions, and respond to events generated by the user.

Figure 10 shows the architecture of the Chiron server. The Server Protocol Manager (SPM) handles all of the interprocess communication. It receives and decodes Q messages from client processes, and encodes and sends Q messages to client processes. The scheduler acts as a concurrent buffer, queuing both instructions and events until the interpreter has time to process them. The interpreter pulls messages from the scheduler in a FIFO manner. If the message is an instruction, it decodes the parameter list and calls the specified ADH member function. If the message is an event destined for a client, it encodes the event and sends it to the SPM. Events, which are generated by the X server in response to user interactions, are sent from the ADH to the event handler, where they are converted to a form appropriate to be inserted into the scheduler and either processed in the server or sent back to the client. The SPM, scheduler and interpreter are all implemented in Ada, while the ADH and a small portion of the interpreter are implemented in C++. The event handling loop is inside an Ada task. This makes it possible to process X events and messages from clients concurrently.

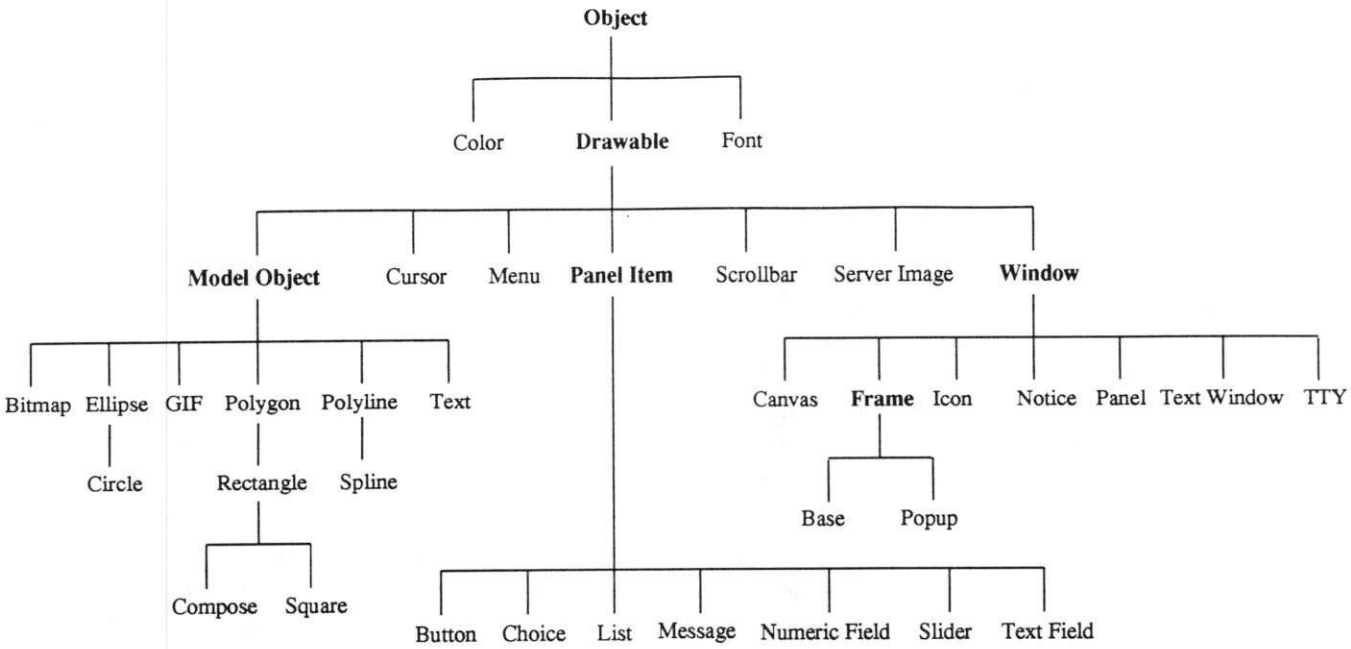

Figure 11: Chiron-1 Abstract Depiction Hierarchy

2.2.1 The abstract depiction hierarchy

The Abstract Depiction Hierarchy (ADH) is a set of classes whose purpose is to provide a simple and high level abstraction for the construction of user interfaces. The hierarchy, historically based on XView[5], is a superset of the intersection of XView and Motif class hierarchies. It provides a consistent interface to both toolkits. The choice of look and feel is a run-time choice; no recompilation of artists is necessary to change the look and feel of the user interface.

The ADH classes provide a mechanism for localization of data. In many instances it is unnecessary to query the X server for a particular object's attribute. The ADH classes store the most commonly used attributes as data members of the individual classes. The ADH supports the common graphical objects found in most toolkits, plus two-dimensional drawing objects such as polygons, circles, splines, and images such as GIFs[6] and bitmaps.

The ADH also provides encapsulation of functionality via subclasses and composition. If the artist writer has an operation that will be used frequently it may be incorporated into the ADH as a new class or an extension to an existing class. The burden of maintaining a depiction thereby shifts from the artist to the server. This functionality then becomes available for use by all other artists. Adding new classes to the hierarchy does not result

[5]This is most evident in the naming of the classes and organization of the class tree which looks remarkably similar to XView's

[6]The Graphics Interchange Format(c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated.

in recompilation of pre-existing artists. Neither does adding a new member function to an existing class. A change to the parameter list of an existing member function may require a recompile of an artist if the artist uses that member function. If a recompilation is required, it can usually be handled without hand editing, by simply running the artist through the preprocessor and recompiling.

The ADH provides a general mechanism for event handling. Events are reported in (ADH object, event) pairs; the Chiron server sends an event message to the client, where it gets routed to the appropriate artist. Each ADH object is capable of receiving a certain set of events based on its class type. For example, buttons may only receive select events, while all objects subclassed from ADL_application may receive menu, select, key, adjust, and move events.

Like most recent research and commercial user interface toolkits, the ADH suffers from the 'least common denominator' problem as applied to the intersection of the toolkits. For the most part, the objects that are supported by the ADH are limited to those that are found in both toolkits. (We extended the hierarchy to support two-dimensional drawing objects and simplified event handling.) The tradeoff of this limitation is that we provide an easier to understand and easier to use interface. Experience with student users has shown that a novice programmer is able to construct fairly sophisticated interfaces based on this hierarchy in a matter of a few weeks.

We found that the logical extensibility of the abstract depiction hierarchy was limited by its historical dependence on XView; in some cases subclassing is difficult. Contrasted with systems such as Interviews [LCV87]. XView's architecture was not designed with extensibility in mind. Object-oriented programming is characterized by inheritance and dynamic binding. XView, because it was implemented in C, uses static subclassing. While XView embodies the first characterization in class hierarchies, it does not provide dynamic binding. The ADH consists of C++ classes that encapsulate each of the XView classes. While this provides the artist writer with polymorphism, it does not provide an ideal design from which to subclass and extend the hierarchy.

Several commercial toolkits, such as Galaxy [BBW92] and Open Interface [Neu91], are now available which provide a similar kind of toolkit independence as Chiron's ADH. The API provided by both systems can support standard look-and-feels for Motif, XView, Windows, and Macintosh across multiple platforms. Both systems use the lowest rendering agent on each platform giving them high performance, more layout control, and look-and-feels on non-native platforms (i.e. XView on a PC). Unlike Chiron, both systems use a standard single-threaded callback architecture.

2.2.2 Server drawing models

In addition to hiding the details of the implementation toolkits, the ADH provides a high level of abstraction allowing Chiron's drawing objects to be supported using several different graphical languages. While Chiron's multi-lingual support has been predominantly focused on the client side (see discussion in section 4), some experimentation with graphical and image modeling languages within the server have been performed. Similar to how the ADH encapsulates details of the underlying toolkit, the drawing model implementation of

the Chiron server can also be hidden from the client. As a means of demonstration, an example Chiron client was built to run with several Chiron server prototypes. Each server prototype used either Phigs [Gas92], PostScript [WG92], XGL[7], or Xlib [O'R92] calls as the drawing model in addition to the standard ADH classes. The example client allowed the user, through menu selection, to choose the color of a triangle drawn on a graphical canvas. Through abstraction, the choice of the imaging language, like the toolkit, is a decision that is delayed until run-time or even the execution of the graphical call through the use of dynamic library loading. This allows Chiron to capitalize on multi-lingualism at the imaging level as well as at the programming and development level of the client.

[7]XGL-2.0 Reference Manual

3 User interface development under Chiron

In order to construct a Chiron client, it is necessary to specify the desired client configuration, build an initial client architecture, and define each artist. (These steps are described in detail below.) Then, it is necessary to translate each artist from LoCal to native Ada. Finally, the client is compiled, linked, and executed.

Although the focus of this work has been primarily on user interface and application architectures, issues of user interface development cannot be ignored. Realistically a system architecture in the user interface domain, no matter how well engineered, is not viable unless the process of UI development can be at least partially automated. With this in mind, a collection of tools has been implemented to assist in performing the above tasks.

3.1 Specifying a client configuration

The client developer must first determine what artists are needed and which ADTs they will monitor. He or she must also choose a dispatching architecture, and which artists should appear at startup and on which machine displays.

Once the configuration is determined, it must be specified in the form of a *client configuration file* (CCF). The information in this file is used by various generator tools to create appropriate client components. It is also read during client initialization in order to determine the run-time configuration of a client. The run-time configuration defines which artists and how many instances of these artist should be invoked initially as well as which display they should use. An example of a client configuration file for the configuration of a flight simulator is given in Figure 12. A client configuration file contains the following information:

ADT specification list: A list of the filenames of all ADT specifications from which client events should be generated. Each ADT specification filename can optionally be followed by the word "dispatcher". If present, a dedicated dispatcher will be generated for that ADT.

Artist descriptions: This portion of the configuration file describes each artist, listing its name followed by any ADTs for which it will want to receive events.

Runtime configuration: Each line contains an artist name followed by the number of instances of that artist that should be invoked at client initialization, optionally followed by which machine display should be used. If no display is specified, unix:0 is the default.

3.2 Generating an initial client architecture

Several tools exist that generate the client run-time components described in section 2.1. In addition, a single tool, called the *client_builder*, generates a full client architecture based on the contents of a given client configuration file.

This tool generates a client initializer, an artist manager, and a client dispatcher. A client event definition is generated that is based on the union of all operations on ADTs

```
tail_spec.a                     dispatcher
aileron_spec.a                  dispatcher
panic_spec.a                    dispatcher
throttle_spec.a                 dispatcher
speed_spec.a                    dispatcher
altitude_spec.a                 dispatcher
psi_spec.a                      dispatcher
attitude_spec.a                 dispatcher
theta_spec.a                    dispatcher
weight_spec.a                   dispatcher

Tail_Module_Artist              tail_spec.a
Aileron_Module_Artist           aileron_spec.a
Panic_Module_Artist             panic_spec.a
Throttle_Module_Artist          throttle_spec.a
Airspeed_Module_Artist          speed_spec.a
Speed_Module_Artist             speed_spec.a
Altimeter_Module_Artist         altitude_spec.a
Altitude_Module_Artist          altitude_spec.a
Rate_of_Climb_Module_Artist     altitude_spec.a
Rate_Module_Artist              altitude_spec.a
Compass_Module_Artist           psi_spec.a
Pitch_Artist                    attitude_spec.a
Horizon_Module_Artist           attitude_spec.a theta_spec.a
Turn_Module_Artist              theta_spec.a
Roll_Artist                     theta_spec.a
Weight_Module_Artist            weight_spec.a


Tail_Module_Artist              1    unix:0
Aileron_Module_Artist           1    unix:0
Panic_Module_Artist             1    unix:0
Throttle_Module_Artist          1    unix:0
Airspeed_Module_Artist          1    unix:0
Speed_Module_Artist             1    unix:0
Altimeter_Module_Artist         1    unix:0
Altitude_Module_Artist          1    unix:0
Rate_of_Climb_Module_Artist     1    unix:0
Rate_Module_Artist              1    unix:0
Compass_Module_Artist           1    unix:0
Pitch_Artist                    1    unix:0
Horizon_Module_Artist           1    unix:0
Turn_Module_Artist              1    unix:0
Roll_Artist                     1    unix:0
Weight_Module_Artist            1    unix:0
```

Figure 12: Example client configuration file

in the ADT specification list. A wrapper is generated for each ADT, and a dedicated ADT dispatcher is generated where indicated in the CCF file. An artist specification and template for an artist body is generated for each artist that is specified in the artist descriptions section of the CCF.

The client developer must hand-edit the client event definition in order to specify the correct access modes (Read or Write) for each ADT operation. In addition, the developer may want to add hand-coded client events.

3.3 Writing artists.

Artist writing is the most labor-intensive task in Chiron client building. Artist writers build and maintain depictions programmatically by making calls to the abstract depiction hierarchy. This includes defining how an artists will handle client and server events. We first discuss the necessary elements to hand-coding an artist and then turn to some work that has been done to simplify and automate this task.

Chiron provides an artist template as a starting point for artist development. A template includes an abstraction of an artist's architecture along with directives to aid the artist programmer in defining the artist.

An example of an artist template for the Throttle artist depicted in Figure 1 is given in Figure 13. The major tasks required in defining the artists are indicated by the comment directives. They are:

- Declare and create the initial graphical objects that make up the artists.
- Define handling routines for client and server events.
- Register handling routines for specific client and server events.
- Start processing on the artist's base frame.

Artists are written in a language called LoCAL that is described in section 4. It is a minor extension to Ada that enables the artist writer to declare, create, and manipulate (in an object-oriented manner) instances of the graphical objects within the server's abstract depiction hierarchy. LoCAL programs are translated by a pre-processor into valid Ada.

The fully defined artist is listed in Appendix A. To familiarize the reader with the internals of an artist, we briefly describe the implementation of the Throttle artist. The line numbers indicated in the following paragraphs refer to Appendix A.

Any graphical object that an artist will create and manipulate must first be declared. Lines 29–31 provide the LoCAL declarations for the artist's base frame, panel, and slider, which comprise all of the graphical objects within the Throttle artist. Lines 88-120 show the LoCAL *apply* calls to create instances of these initial graphical objects.

This artist defines only one handling routine for a server event (when the user adjusts the slider). The definition of this handler is given on lines 47-60. Server event handlers have two parameters: the object of the event (in this case, the slider) and the server event description. We use the event description to obtain the new desired Throttle position and use that information to change the state of the Throttle ADT.

A client event handler has only one argument, the client event description. If the application had an autopilot mode in which the throttle state would be modified program-

```
with Wrapper_Throttle_Module;
with Client_Dispatcher;

package body Throttle_Module_Artist is

  task body Throttle_Module_Artist is

    Self_Ptr       : Throttle_Module_Artist_Ptr;
    Local_Artist_ID : CSL.Artist_ID_Type;
    Local_Display   : CSL.Str;

    --<< declare artist objects here. >>

    --<< declare handler routines for both client and server >>
    --<< events plus any auxilliary routines here.          >>
    --
    --server events handlers must have the signature:
    --procedure <handler_name> (Object : CSL.Object_Type;
    --Event  : CSL.Chiron_Event_Ptr);
    --client events handlers must have the signature:
    --procedure <handler_name> (Event : Client_Event_Ptr);

  begin --task body

    accept Start_Artist (
           ID   : CSL.Artist_ID_Type;
           Self_Aptr : address;
           Display_Name : CSL.Str) do
      Self_Ptr := address_to_artist(Self_Aptr);
      Local_Artist_ID := ID;
      Local_Display := Display_Name;

      --<< register interests in client events with the >>
      --<< client_dispatcher here.                      >>

    end Start_Artist;

    --<< create initial graphical objects here. >>

    --<< set behaviors of graphical objects in response >>
    --<< to server events here.                         >>

    --<< call adl start_processing method here. >>

    loop
      select
        accept Notify_Client_Event (
               Client_Event   : Client_Events.Client_Event_Ptr;
               Handler_Routine : address);
      or
        accept Notify_Server_Event (
               Object         : CSL.Object_Type;
               Server_Event   : CSL.Chiron_Event_Ptr;
               Handler_Routine : address);
      or
        accept Terminate_Artist;
      end select;
    end loop;
  end Throttle_Module_Artist;
end Throttle_Module_Artist;
```

Figure 13: Template generated for Throttle_Module_Artist body

matically, we would have defined an event handler for whenever the Throttle state was modified. The client event could be used to obtain the new Throttle position and an ADH call would be made to reposition the slider accordingly.

Registering for client events consists of making a call to the client dispatcher, providing the client event for which we are registering and the address of a handling routine for that event. This artist does not handle any client events, but if it did, registration calls would occur at line 77. To register for server events, the user defines the *behaviors* (pointers to handling routines) for graphical objects. A behavior associates a handling routine with each possible server event that can occur on an object or class of objects. If no handling routine is provided, the event is ignored. On lines 125-127 we define a behavior for all objects of class ADL_slider (a behavior for a slider instance would override this behavior) then register this behavior by making a LoCAL *set_behavior* call.

Finally we must activate the artist by invoking the *start_processing* method on the artist's base frame. This call is found on lines 132. This will cause the artist to appear and begin listening for server events.

Front-end builders The burden or writing artist bodies can be substantially reduced through use of a graphical interface builder. Many of these are on the market, and perform roughly the same services. Rather than adding another interface builder to the market we examined how we could leverage the investment in existing tools. We discovered that the internal form generated by Sun Microsystem's Developer's GUIDE[8] tool [Sun91] (GIL) was suitable for translation to LoCAL. Accordingly we developed two tools, *gil2local* and *gab* (guide artist builder), which enable the use of GUIDE in creating Chiron artists.

gil2local translates the output of GUIDE to LoCAL, and *gab* allows users to update an artist template by inserting the gil2local generated LoCAL calls into the appropriate places in the artist template. The artist writer thus has a very substantial basis for creating an artist quickly, using the set of tools provided with Chiron.

An interesting side-effect of gil2local and gab is that, though GUIDE was created to support development of applications that employ the OpenLook look-and-feel, it can now be used to create applications exhibiting the Motif look-and-feel since LoCAL is independent of that choice.

Although this approach can significantly reduce the time required to prototype a user interface, it suffers several drawbacks that make it less than an ideal approach. Manual effort is still required to create and manipulate drawing objects (such as circles or polylines) which are not supported by GUIDE, and to set the dynamic behaviors of those objects. Because the corresponding graphical objects provided by the Motif toolkit differ in dimensions with those provided by the XView toolkit, the corresponding Motif layout generated by GUIDE is sometimes not aesthetically pleasing. Finally, the gab implementation expects an empty artist template. Any iterative development would involve hand-pasting non-generated code. The gab tool could be modified, however, to allow interative development as supported by GUIDE's GIL to C generator, GXV.

[8] OpenWindows Developer's Graphical User Interface Development Environment

Artist generators. Chiron has been developed as part of a larger project in software engineering environments called Arcadia. Since graphs are such a pervasive data structure in software engineering environments, the Arcadia project developed a persistent graph package generator, called *PGraphite* [WWFT88]. With this tool, the user specifies the particular type of graph desired and the tool generates an Ada package encapsulating the graph and providing creation/deletion routines, traversal routines, and persistence functions. We have extended PGraphite with artist generator capabilities. Specifically, in addition to all the other graph attributes specified in the input to PGraphite, the user may specify which nodes should be displayed, and with what graphical attributes. A *complete* artist for the graph package is then generated.

Reusable artist components. The pervasiveness of graphs in software engineering environments has also motivated the development of a generic routine for the the layout of hierarchical directed graphs called GLAD (Generic LAyout for Directed-graphs), which may be used within artists for automatically laying out a graph and reducing edge crossings [Sni91]. This capability has been used in several Chiron applications, including computer aided software engineering applications and graph-based analysis tools.

We have also found that artist writers often need to maintain relationships between program objects with graphical objects. For example, if a user selects a rectangle representing a node in a graph, the artist will want to locate the corresponding ADT node instance. For this, we provide a generic bidirectional association table that can be instantiated to associate any application object with any graphical object. Additional context data may also be stored with each relation.

4 Multi-lingual support

Chiron's client-server architecture, employing event-based integration, permits different client modules to be written in different languages. For Chiron-1 only the language support tools for Ada were implemented, but, assuming the same client architecture, these tools could be changed to support other languages in addition to mixed-language clients. To test this claim, various client prototypes were built for use with the current implementations[9] of the Chiron server. The requirements Chiron places on the client development language in addition to our experiences with these prototypes are described in the balance of this section.

Chiron's exploitation of concurrency and event-based integration of client components places few restrictions on the application architecture. Nevertheless it requires language support for object-oriented concepts and run-time support for concurrency. These requirements may not map well into various implementation and development languages, and in some cases necessitate language extensions in order to preserve the Chiron design.

4.1 LoCAL

Current user interface languages are relatively weak from the standpoint of software engineering concerns such as support for large-scale programming, flexibility, and code reuse. LoCAL is designed to overcome some of these weaknesses.

LoCAL is designed as an extension to Ada that allows effective use of the abstract depiction hierarchy (ADH) in an object-oriented manner while meeting two additional user interface development constraints. First, it is desired that artists be written in the same (or very similar) programming language as clients. This is so that developers can avoid the burden of learning a new or specialized user interface language and it also eliminates the possibility of two different languages' run-time systems conflicting within a single process. Ada is our language of choice for writing large-scale client applications, so an Ada-like language for writing artists is needed. Second, the artist language must be able to interface in a reasonable way to the ADH, since this library is used to define the components of the user interface that the artist implements. Because the ADH is an object-oriented class library that relies on inheritance and subclasses to achieve flexibility and reuse, and since Ada does not have the language features necessary to support inheritance, it was decided to implement LoCAL as an object-oriented extension to Ada.

The LoCAL features added to Ada are summarized in figure 14. The first form of Apply creates an instance of the indicated class, using the designated constructor method. The second and third forms of Apply invoke the designated method of the specified object. One form is for methods that return results, and the other form is for methods that do not return results. The Set_Behavior function establishes the artist's event handlers that execute in response to specified user manipulations of the given ADH object.

The Chiron development toolset includes two tools that implement this extension: the LoCAL pre-processor (*lcc*), and the *adl_compiler*. It is important to note that although the

[9] All client prototypes while developed in various programming languages are compatible with both the unmodified versions of the Motif and XView servers.

```
--call a constructor method to create a new instance

  function ~Apply (
    class : class_name;
    method : method_name;
    [<parameters to class method>] )
    return object_type;

--call a method that does not return a value

  procedure ~Apply (
    class : class_name;
    instance : object_type;
    method : method_name;
    [<parameters to class method>] );

--call a method that returns a value

  function ~Apply (
    class : class_name;
    instance : object_type;
    method : method_name;
    [<parameters to class method>] )
    return <return type of class method>;

--establish handler for each of the "behaviors"
--(user manipulations of the visual representations of
--the object)

  procedure ~Set_Behavior (
    object : ADH_object;
    behaviors : Behavior_Array_Type );
```

Figure 14: LoCAL extensions to the application language

adl_compiler and the lcc tool were written to provide the interface between Ada artist code and the Chiron ADH, they provide a general capability for object-oriented interoperation between Ada and C++.

lcc translates LoCAL into standard Ada. It reads in a symbol table that specifies a mapping from a set of class names and member functions to integers. A LoCAL call to a member function of a given object is translated into a standard Ada procedure call. The parameters of the procedure call are the integers that specify the class of the object and the function to be invoked on the object. A reference to the object instance and the parameters of the function are also passed as parameters to the Ada procedure call.

The adl_compiler compiles C++ class declarations into the symbol table files required by lcc. This allows the automation of bindings to the C++ code in the ADH. After a change to the C++ interface, all that is required is to run the adl_compiler to create a new symbol table file, and use lcc to translate LoCAL code to Ada using the new mappings. This process is easily automated using a standard tool such as *make*.

4.2 Language prototype examples

Because Chiron maintains a strong separation of concerns between the application and the user interface software, various architectural components of the client can be coded in different programming languages as long as the run-time support for each language can safely coexist within a single process. For components written in languages that are not compatible, the event-based client design lends itself well to distributing components across process boundaries, although in the current version there are no client support tools to automate this restructuring. In order for Chiron to support clients in other languages, bindings must be established between the ADH and the application language. For object-oriented application languages this binding is straightforward; for languages like Ada that lack object-oriented language features the changes are more substantial, as described above.

In order to demonstrate the generality of the client design, prototypes using Fortran, Lisp, and C++ are described in the following paragraphs.

Mixed Ada-Fortran Client. An interface for manipulating and visualizing the infrared signatures of the tailpipe of Northrop's B-2 aircraft was constructed by extending pre-existing Fortran code with a Chiron artist. This was accomplished by wrapping the internal data structures of the tool in such a way that data state changes are propagated as Chiron client events. In addition, the Chiron artist can change the data state and visualization in response to user actions. Although most of the support code for this application was hand-generated, automatic support for code generation and wrapping could be provided by Fortran-specific versions of the client support tools described in subsection 2.1.

Lisp Client. Another way to exploit multi-lingualism in the Chiron design is through the client-server separation. Because the client and the server run in different processes and the mechanism for inter-process communication allows for multi-lingual messages ([MOS90]), client processes can be built in various language. One such prototype was built using Common Lisp [WH89]. The focus of this particular Lisp client was to provide development

support for specification of artists through interactive LoCAL calls to the ADH. This was done while still maintaining a strict separation of concerns from the underlying graphical substrates through the client-server split. Current functionality of the Chiron Lisp client includes the mechanisms for registering with a Chiron server and the automatic generation of CLOS[10] primitives from the Chiron Standard Library's symbol table file (see 4.1) that provide an interface to the Chiron server. A full suite of Chiron client tools supporting Lisp development was never realized. In addition, the implementation of the Lisp client relied on a multi-processing package external to the language for providing concurrency.

C++ Client. Another client prototype demonstrating the benefits of the client-server separation was built using C++. C++'s support for object-oriented design and data abstraction through class mechanisms allow a natural wrapping of C++ objects (where they play the role of Ada ADT's). Unfortunately this places the burden of inheritance tracing on the client support tools since C++ objects are not required to include virtual function prototypes (signatures) in their description. Inherited interfaces from these functions need to be exhaustively enumerated at the time of client generation to allow for the appropriate client events to be sent to the interested artists. This problem, while not unsurmountable, was not addressed in the C++ client development tools, and a robust and complete collection was never built. Similar to the Lisp client, concurrency was provided through using a threads package external to the implementation language.

4.3 Extended language support

Two good candidates for future Chiron client languages are Ada9X [Ada93] and Modula-3 [Har92]. Ada9X's new mechanisms for handling inheritance and polymorphism in conjunction with established support for modularization, data abstraction, and concurrency allow this language to map cleanly into the Chiron client architecture. Modula-3's object-oriented programming constructs, high-level support for concurrency and concurrent access to data structures also make this language a suitable candidate for future client support. Chiron's design allows various client implementations to easily leverage off of these properties and cleanly address the weaknesses of current user interface languages.

[10] Common Lisp Object System[WH89].

5 Performance

In order to effectively address the objectives of architectural flexibility, robustness in the presence of change, and reuse of software artifacts, a tradeoff between performance and maintainability is employed in the overall Chiron design. In this respect, the implementation of Chiron is a compromise where the overhead of each technology is carefully weighed against the overall system performance so that the goal of equivalent performance to *ad hoc* systems is achieved. To demonstrate the scope of the overall system and to assess the performance costs the developer incurs through using the Chiron user interface technologies, several size and speed statistics are provided in the balance of this section.

5.1 Space

Chiron is a robust collection of user interface support tools. In terms of lines of code, the union of all Chiron generator tools and run-time code consists of 65KSLOC (thousand-Source-Lines-Of-Code) of hand-written code (37K Ada and 29K of C++). 32KSLOC were generated, for a total of 98K.

Architecture	*Size of binary*	CSLOC[11]
X11/Ada	1.97M	43
GXV/Ada	2.01M	195
GXV/C	1.48M	133
GXV/C++	1.41M	125
Chiron[12]	1.00M	121+client-code

Figure 15: Architectural Space Comparison

In terms of space, the Motif implementation of the Chiron server is 7.3M, while the XView implementation is 10.6M. A trivial Chiron client takes about 1M, 320K of which can be attributed to the Ada run-time system.

In figure 15, a comparison is made between the implementations of a *hello world* program which consists of a labeled base-frame. Each of these architectures are generated from the same GUIDE file using the GUIDE XView Ada (GXV_ADA), C[13](GXV), C++(GXV++), and Chiron's gil2local/gab tools. The X11/Ada example, however, is hand-coded using the Ada bindings to the XView toolkit. Chiron client tools generate about 720 lines of support code for a single client regardless of how many Artists and ADT's are specified in the client configuration. This code provides functionality for artist management and client initialization in addition to dispatching. It is important to note that the *hello world* Chiron client needs to run in conjunction with one of the two Chiron servers. The flight simulator

[11] Commented Source Lines of Code

[12] 121 (LoCAL) and 715 (Ada) → 1055 (Ada)

[13] Linked with *-Bstatic* to disable dynamic loading.

client illustrated in figure 1 has a binary size of 1.7M. The uncommented number of lines of non-generated source code (including ADTs, application, and artists) is 1590.

5.2 Speed

Architecture	Low	High	Average
X11/Ada	0.35	1.20	0.60 Sec.
GXV/Ada	0.37	1.27	0.60 Sec.
GXV/C	0.33	0.86	0.59 Sec.
GXV/C++	0.42	0.90	0.64 Sec.
Chiron	0.62	1.85	0.98 Sec.

Figure 16: Execution Time Comparison

Performance degradation can be expressed in terms of the number of increased instructions, and the extent of this overhead is discussed in the following paragraphs.

Figure 16 provides a comparison of times for the creation and display of graphical objects for the simple *hello world* program described in section 5.1. In addition to these numbers, the overhead to bring up an interface with about 20 graphical objects is negligible and takes about 3 seconds. An end-to-end user interaction takes approximately 15 milliseconds. This time reflects the processing of an event generated from the concrete depiction of a graphical object. The event traverses the following route through the Chiron architecture: the X11 event is transferred (C to Ada) to Chiron, scheduled and interpreted in the Chiron server, sent to the Client (process to process), mapped to the artist to which the event is relevant, executed inside of the particular artist event handler, and possibly routed to the ADT with which the Artist is associated. The time was measured over 1000 events and then averaged. Measurements were made on Sun Sparc-2s, with the client and server on separate machines.

ADH. The charge against performance through using Chiron's ADH is concentrated in two areas: the cost of maintaining a structured object hierarchy in the server and the cost of providing the mechanism allowing the client to access the ADH through LoCAL.

One requirement of the ADH is to provide a higher level of abstraction to the user beyond that of graphical toolkits and the windowing system. This however creates the situation where a single ADH call may take a large number of substrate calls to enact the instruction. In addition to this, maintenance of the concrete depiction of an object as a result of changing its attributes, efficient screen refreshing and redrawing, and layout concerns and constraints are all added into the ADH and need to be constantly re-evaluated to maintain graphical consistency. For toolkit objects this overhead is relatively small, approximately 2-3 instructions overhead, but for graphical drawing objects not provided by the toolkits, this overhead can be as much as 10-15 instructions per single call. Also, the Chiron ADH, through providing a higher level of abstraction and greater degree of flexibility, relies heavily on the object-oriented aspects of its implementation language (C++). The use

of polymorphism, class derivation, multiple inheritance, and user-customizable constructor and destructor functions in C++ all incur large performance costs, especially for complex objects embedded in deep-derivation trees as seen in the graphical portions of the ADH. A more general discussion of performance issues using object-oriented and language dependent techniques can be found in [Ree92a] [Ree92b].

In order to maintain the advantages of flexibility that the LoCAL approach to accessing the ADH provides, the LoCAL code is translated and expanded into *pure* Ada calls. This expansion degrades performance of the system because the translation of LoCAL into Ada results on average in three times more instructions to be executed. It is interesting to note that historically this is roughly the same rate of expansion as first seen when adding object-oriented extensions to the C language through C++.

Client Architecture. In addition to the code expansion and performance overhead through using LoCAL and the ADH, Chiron's client architecture provides support code for event-based dispatching to maintain multiple coordinated views and the application-interface separation. For each ADT described in the client configuration file, a wrapper is generated to provide concurrency and dispatching support. However, this introduction of an intermediate software layer results in a performance overhead for each ADT interface call of two procedure calls for concurrency control and two for event dispatching. Time for dispatching is essentially the time to do a round-trip Ada rendezvous, which for N-artists is N+2 task entry calls where an Ada task entry call is approximately equal to a procedure call + some small time δ for task synchronization which is compiler and operating system dependent.

Client-Server Split. The inter-process communication mechanism chosen to implement Chiron's client-server split allows Chiron clients and servers to communicate to each other across different networks, languages, operating systems, and platforms. While this allows architectural flexibility, this advantage comes at the price of performance. In addition to the expense of performing inter-process communication, messages sent between the client and server need to be encoded and decoded to allow for interoperability between heterogeneous platforms and languages. For simple, pre-defined types (i.e. integer, boolean, etc.) there is a four instruction overhead (two for encoding, two for decoding), but for complex composite types (i.e. records possibly with embedded structures) there is also a four instruction overhead for each structure in addition to the encoding of its components. This overhead basically represents the tagging and linearization of a complex data type. To avoid heavy inter-process communication, the ADH is designed to minimize the number of messages sent across the split through abstraction and reuse.

In conclusion, therefore, the use of a structured user interface object hierarchy (ADH), a message-based dispatching client architecture, and a client-server separation, while addressing Chiron's high-level design objectives, represents a tradeoff between functional advantages and potential performance penalties. Because of Chiron's architectural flexibility, systems developed using Chiron can be tailored to meet specific functional and performance and goals.

6 Summary comparison to other work

In addition to the other systems discussed elsewhere in this paper, several key systems influenced the design and evolution of Chiron-1. Chiron-1's predecessor, Chiron-0 [YTT88], from which several key ideas were taken, was influenced by Smalltalk's [GRs83] model-view-controller (MVC) paradigm and its separations of concerns. Artists for data structures were introduced by Myers in the Incense symbolic debugging sytem [Mye83], as noted elsewhere. Loops [SBK86] uses a special form of inheritance called *annotation* that binds the equivalent of artists to objects. This annotation concept inspired Chiron's wrapper technology. Finally, Anson's *device model* [Ans82], provided some key insights in developing concurrent interfaces.

Ongoing research projects in the user interface area offer approaches that contrast in various ways with that of Chiron. Rather than comparing all systems on the basis of the same criteria (which is desirable but which would consume far too much space) we instead comment only on the most relevant features of some of these other systems.

A general architectural comparison can be made with two well-known systems, Interviews and Garnet. InterViews [LVC89] uses C++, and is tightly coupled to that language. Application and user interface code are closely tied together, and a user interface listener/dispatcher is in control of the single control thread. The application is dormant until InterViews invokes application code. Application code takes the form of virtual functions in user-defined sub-classes of the InterViews class hierarchy. This is in contrast with Chiron's focus on clear separation of the user interface from the application, as well as Chiron's concurrent control model.

Garnet [MGD+90] is written in Lisp, and applications must be coded in Lisp. Garnet makes use of a combined constraint manager and object manager. For the application to respond to user events, a Garnet listener is given control, and application code is again dormant until invoked by the user interface layer in response to user interactions. While architecturally this is again in contrast to Chiron, we are investigating ways of incorporating constraint mechanisms into Chiron to achieve many of the benefits exhibited in Garnet (and Gilt [Mye91]).

In the Dynamic Windows/Common Lisp Interface Manager [McK91], presentation types are built through inheritance from application objects. This is similar in concept to the way Chiron artists are built, since they obtain much of their interface from the package specifications of the object they display. Chiron, in its Ada implementation, cannot use inheritance for this purpose, however.

The Suite system [Dew90] implements a distributed architecture that supports a groupware service . The distributed architecture of Suite is similar to Chiron's client-server orientation, but is more fully exploited in Suite. In Suite, however, editing is the main metaphor for user interaction and graphics are not supported.

The Picasso toolkit [RKS+91] uses a programming language metaphor. User interface objects are created and destroyed on scope entry and exit, and new parameter passing modes have been devised to support user interface programming idioms. Chiron's LoCAL has some similarities, though it is much simpler in objective. Chiron has avoided any toolkit development, in contrast to Picasso. Picasso's triggering (based on its constraint

mechanism) is similar in purpose to Chiron's dispatcher, but Chiron uses multiple, strongly-typed, concurrent mechanisms.

Serpent [SEI89] uses a client/server relational database trigger model with restricted data types. This is similar to Chiron's dispatcher, but Chiron places no restriction on the set of types that may be displayed and does not demand that the application relinquish, to the user interface system, storage control of the objects to be displayed.

Three papers at CHI'92 are especially deserving of comment. The Rendezvous project [Hil92] focuses on the use of the abstraction-link-view paradigm for the structuring of applications and user interfaces. Rendezvous' abstractions are akin to Chiron's application's ADTs; its links are related to Chiron's dispatchers, and its views are similar to Chiron's artists. In terms of capabilities the two systems have much in common, though Rendezvous uses constraint technology where Chiron uses a simpler event-based architecture. A key difference, however, is that to use Rendezvous the application developer must write the application in Rendezvous, "an extended object-oriented Lisp".

The AT&T Display Construction Set UIMS [RBW92] separates applications from the UIMS via Unix pipes. The UIMS can thus receive data events from multiple processes. The data messages are not richly typed, however, restricting application to situations where simple database records suffice to capture the object to be displayed.

NASA's TAE Plus system [Szc92] is focused on making the generation of applications with GUI's easier than by using toolkits directly. TAE insulates the application from changes in the toolkit. TAE Plus is like many commercial systems, however, which generate an application code *template* into which the remainder of the application code must be inserted. The UIMS thus strongly determines the architecture of the application.

The Proteus document presentation system [GHM92] provides a set of services that allow the presentation of software development documents (i.e. programs and design specifications) to be determined by a formal style specification. Proteus maintains a separation of concerns between document structure and presentation issues. This separation, similar to Chiron's separation of user interface and application, enables coordinated, multiple views of documents with support for simultaneous editing.

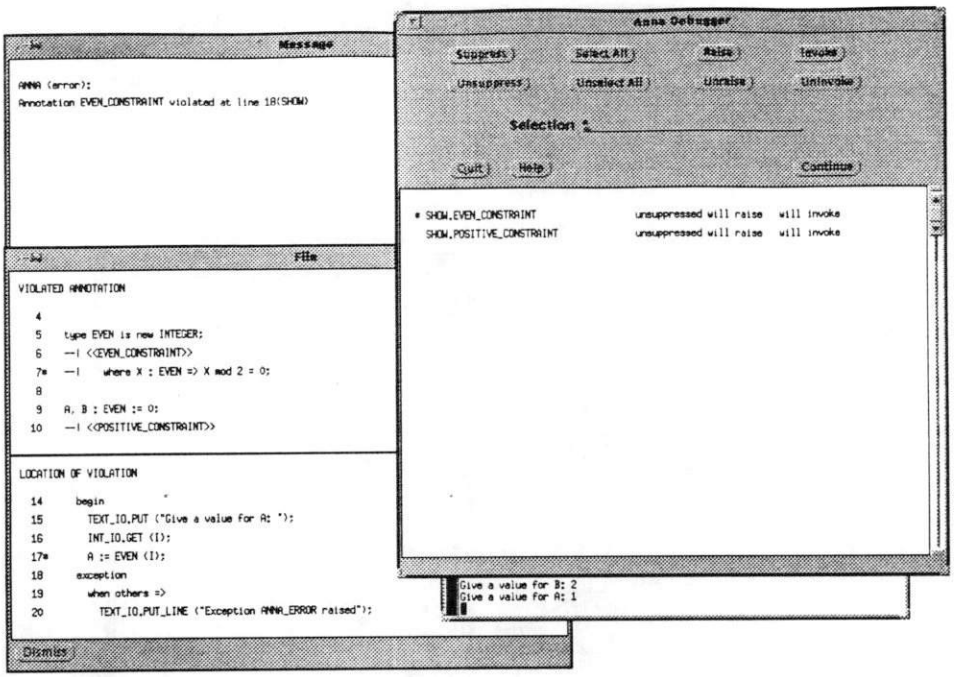

Figure 17: The Anna Debugger reporting a violation.

7 Example uses of Chiron

Chiron has been applied to a wide range of problems, by undergraduate and graduate students at UCI and the University of Colorado, the development team, and unaffiliated research groups both at UCI and elsewhere. Several of these applications are briefly described below. These descriptions are designed to illustrate the range of applications for which Chiron is appropriately used as well as to indicate the degree to which its separations of concerns and other technical characteristics have been effective in practice.

7.1 The Anna debugger

The Anna Debugger was built at Stanford University by p[rogrammers not affiliated with the Chiron project. It debugs Ada programs which use Anna assertions. The debugger notifies the user when an Anna assertion is violated and displays the assertion in question (See Figure 17). The user can then perform any number of operations and then continue the execution of the Ada program. The Anna Debugger's user interface has progressed from a textual command-line interface to an X-based, viz. Xlib, interface, and finally to a Chiron interface. In fact all three interfaces coexist. The choice of which user interface to use is made at application load time. The key to this achievement is a clean interface to the application which all three user interfaces respect.

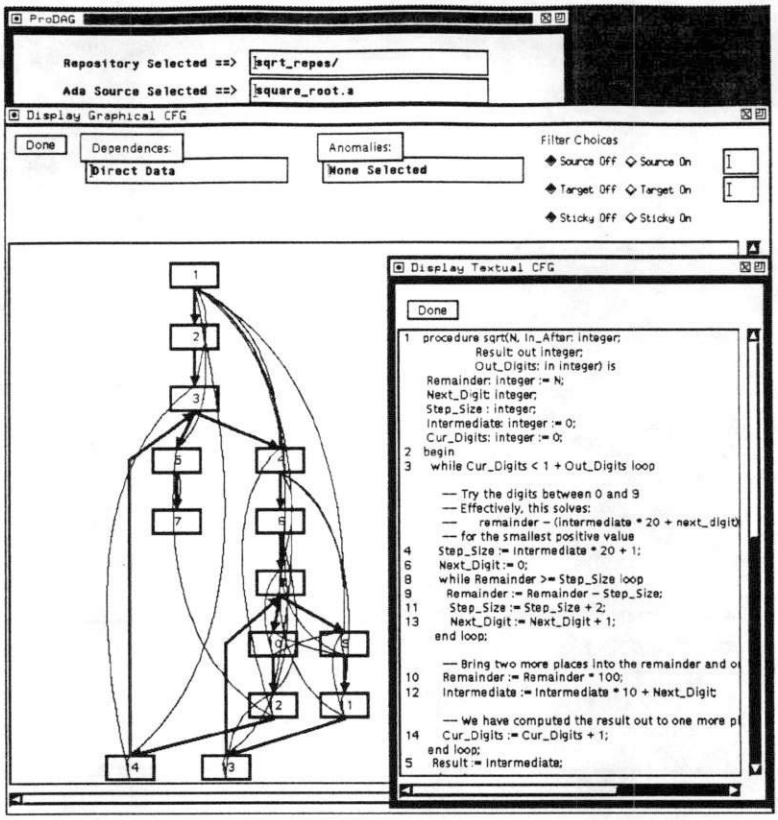

Figure 18: The ProDAG artist annotating a CFG

7.2 ProDAG/TAOS

The Software Testing Group at UCI, distinct from the Chiron group, has developed a graphical user interface using Chiron for two systems called ProDAG (PROgram Dependence Analysis Graph) and TAOS (Testing with Analysis and Oracle Support). ProDAG is a tool set that performs analysis of dependencies between program statements. TAOS is an environment which supports the analysis and testing process of software development. Both systems provide an application programmatic interface (API) to all of their functionality. The GUI for ProDAG consists of six Chiron artists that use the API to provide access to all of the tools in ProDAG. In the same manner, four artists are used for the TAOS system. In ProDAG, the GUI allows a user to analyze a program, display its control flow graph (CFG), and then annotate the CFG to display the dependency information in a variety of ways (See Figure 18). In TAOS, the GUI is used to develop test cases and test data and then execute, verify, and analyze test execution. ProDAG/TAOS consists of eighty-five thousand lines of code produced by four to six programmers. Forty-nine thousand lines of code are contained in the API and supporting application. The GUI takes up the remaining thirty-six thousand lines of code with greater than seventy percent of that code generated by Chiron and the Pleiades object management system [TC93]. The GUI developer thus

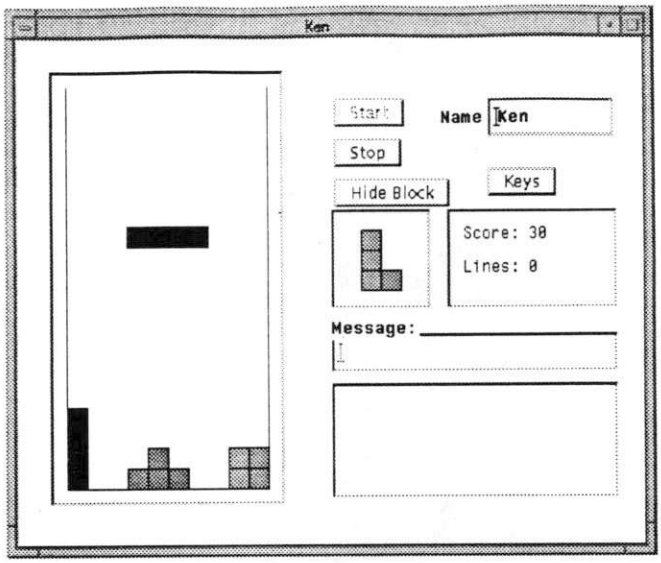

Figure 19: The Tetris artist

wrote about ten thousand lines of code between the two systems.

The API approach used in ProDAG/TAOS lent itself naturally to Chiron's ADT-based artists. In fact, once the API was set, the programmer who developed the Chiron GUI for ProDAG/TAOS did so independently of the programmers who developed the code for the tools. In addition, the GUI developer was able to make use of Chiron's generator tools to quickly generate an initial interface which could then be refined. The GUIDE and gil2local/gab tools allowed the developer to create the initial layout of the artists and then generate skeleton code to implement that layout. The GLAD layout system speeded development of the CFG artist. The final task was to flesh out the generated code to respond meaningfully to calls made on the API.

7.3 Multi-player Tetris game

A Chiron developer produced a multi-player Tetris artist (See Figure 19). On each players' screen, Tetris blocks fall into a well. The players use the keyboard to rotate and position the blocks. When a player clears a row of blocks, this row, minus the block which caused the row to become complete, appears at the bottom of the well of the other player's displays. Thus, while players do not interact with each other on the screen (i.e. they don't see each other's falling blocks) their actions do eventually affect the other players. The run-time configuration of the game (how many players and their machines) can be changed using the standard client configuration file that all Chiron clients use. This file allows changes in the client's configuration between invocations without having to compile or link those changes into the artist. The strength of Chiron's client-server approach is displayed by this artist. During a game, each instance of the Tetris artist runs independently of the

others, focusing only on its user's actions and sending the server the appropriate messages to update its display. This artist also illustrates Chiron's concurrency advantages. Each instance must listen and respond to key events from the user, row completion events from the other artists, as well as animating the falling Tetris blocks.

8 Summary and conclusions

Several principles guided the design of Chiron. Primary impacts of three of them on Chiron are described below.

Separation of concerns. The two areas in which the notion of separation of concerns were considered important were the separation of the application and the user interface code, and the separation of the user interface code and the underlying toolkit and windowing system. The artist concept provides separation between the application and the user interface code. The notion of the abstract depiction hierarchy, coupled with the client-server architecture, provides separation between the user interface code and the underlying toolkit and windowing system.

Concurrency. Concurrency was achieved along a couple of dimensions. The application code can be concurrent, since multiple Ada tasks can be active in an application process. Artists run concurrently with the application. Concurrency also arises because of the client-server split. One large application could consist of several clients and servers, each running on a different machine.

Language and platform independence. The client-server architecture of Chiron makes it possible to have application components written in different languages work together. Artists in the clients communicate via inter-process communication links to the server, and it does not matter what language they are written in. The abstract depiction hierarchy hides the details of the particular underlying toolkit.

The Chiron user interface system provides separation of concerns to enhance reusability and robustness to change, architectural extensibility and flexibility in developing applications, and support for multiple-user and networked applications. At the same time Chiron has been able to realize a level of performance necessary for applications typical to software development environments, the original context for Chiron.

Chiron is particularly well suited for distributed, long-lived, object-based applications. Applications of substantial complexity particularly benefit from Chiron's separations, for the design of the application's software architecture is independent of user interface concerns. The Chiron architecture is also ideal for heterogeneous applications that may include multiple languages, multiple platforms, and COTS software.

A wide variety of users, from undergraduate students to professional programmers, have successfully applied Chiron to a wide range of applications. We believe therefore that the principles, architectural schemes, and mechanisms we have explored are both workable and desirable; we encourage other UI researchers to benefit from our efforts. The system and additional papers are available by anonymous ftp to *liege.ics.uci.edu* in */pub/arcadia/chiron*, or on the World-Wide Web (URL *http://www.ics.uci.edu/Arcadia/chiron.html*).

Acknowledgments

Mary Cameron, Ruedi Keller, and Dennis Troup were key contributors to the original Chiron-1 architecture. Other important contributions been made by John Self, Craig Snider, and Robert Zucker. Critical performance improvements were made by David Levine.

References

[Ada93] Ada 9X Mapping/Revision Team. *Ada 9X Reference Manual.* Intermetrics, Inc., Cambridge, Massachusetts, September 1993. Version 4.0.

[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).

[BBW92] Jeff Barr, Andrew Bernard, and Jay Wettlaufer. *Galaxy Application Environment Technical Description.* Visix Software Inc., 11440 Commerce Park Drive, Reston VA 22091, 1992.

[Bro92] Marc H. Brown. Zeus: A system for algorithm animation and multi-view editing. Technical Report 75, Digital Systems Research Center, Palo Alto, CA, February 1992.

[Cag90] Martin R. Cagan. The HP SoftBench environment: An architecture for a new generation of software tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.

[Dew90] Prasun Dewan. A tour of the suite user interface software. In *Proceedings of the Third Annual Symposium on User Interface Software and Technology*, pages 57–65, Snowbird, UT, October 1990. Association for Computing Machinery.

[Gas92] Tom Gaskins. *PHIGS Programming Manual.* O'Reilly and Associates, Inc., 103 Morris Street, Suite A, Sebastopol, CA 95474, 2nd edition, 1992.

[GHM92] Susan L. Graham, Michael A. Harris, and Ethan V. Munson. The Proteus presentation system. In *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments*, pages 130–138, Washington D. C., December 1992.

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

[Har92] Samuel P. Harbison. *Modula-3.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[Hil92] Ralph D. Hill. The abstraction-line-view paradigm: Using constraints to connect user interfaces to applications. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 335–342, Monterey, CA, May 1992. Association for Computing Machinery.

[LCV87] Mark A. Linton, Paul R. Calder, and John M. Vlissides. Interviews: A C++ graphical interface toolkit. In *Proceedings of the USENIX C++ Workshop*, Sante Fe, NM, November 1987. Appeared as *The Design and Implementation of Interviews* and this is a later version.

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

[McK91] Scott McKay. CLIM: The Common Lisp Interface Manager. *Communications of the ACM*, 34(9):58–59, September 1991.

[MGD+90] Brad A. Myers, Dario A. Guise, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, November 1990.

[MOS90] M. Maybee, L. J. Osterweil, and S. D. Sykes. Q: A multi-lingual interprocess communications system for software environment implementation. Technical Report CU-CS-476-90, University of Colorado, Boulder, June 1990.

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

[Mye91] Brad A. Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology*, pages 95–105, Hilton Head, South Carolina, November 1991.

[Neu91] Neuron Data Inc., 156 University Avenue, Palo Alto CA 94301. *Neuron Data Open Interface Technical Overview*, 1991.

[O'R92] Tim O'Reilly. *Xlib Reference Manual.* O'Reilly and Associates, Inc., 103 Morris Street, Suite A, Sebastopol, CA 95474, 2nd edition, 1992.

[RBW92] Joseph P. Rotella, Amy L. Bowman, and Catherine A. Wittman. The AT&T display construction set user interface management system (UIMS). In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 73–74, Monterey, CA, May 1992. Association for Computing Machinery.

[Ree92a] David R. Reed. Efficiency considerations in C++, part 1. *C++ Report*, 4(3):27–30, March/April 1992.

[Ree92b] David R. Reed. Efficiency considerations in C++, part 2. *C++ Report*, 4(5):23–27, June 1992.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, July 1990.

[RKS+91] Lawrence A. Rowe, Joseph A. Konstan, Brian C. Smith, Steve Seitz, and Chung Liu. The PICASSO application framework. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology*, pages 95–105, Hilton Head, South Carolina, November 1991.

[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.

[SEI89] SEI. Serpent overview. SEI Technical Report CMU/SEI-89-UG-2, ESD-TR-89-08, Carnegie-Mellon University Software Enginnering Institute, August 1989.

[SFG93] Piyawadee "Noi" Sukaviriya, James D. Foley, and Todd Griffin. A second generation user interface design environment: The model and runtime architecture. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 375–382, Amsterdam, April 1993. Association for Computing Machinery.

[SLN93] Pedro Szekely, Ping Luo, and Robert Neches. Beyond interface builders: Model-based interface tools. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 383–390, Amsterdam, April 1993. Association for Computing Machinery.

[Sni91] Craig Snider. GLAD user's manual. Arcadia Technical Report UCI-91-15, University of California, October 1991. Arcadia technical note: User's manual for a generic hierarchical graph layout package. Version 1.0.

[Sun91] SunSoft, 2550 Garcia Avenue, Mountain View, CA 94043. *OpenWindows Developer's Guide 3.0 User's Guide*, 1991.

[Szc92] Martha R. Szczur. Transportable applications environment (TAE) plus user interface designer workbench. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 231–232, Monterey, CA, May 1992. Association for Computing Machinery.

[TC93] Peri Tarr and Lori A. Clarke. Pleiades: An Object Management System for Software Engineering Environments. In *ACM SIGSOFT '93: Proceedings of the Symposium on the Foundations of Software Engineering*, Los Angeles, California, December 1993.

[TJ93] Richard N. Taylor and Gregory F. Johnson. Separations of concerns in the Chiron-1 user interface development and management system. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 367–374, Amsterdam, April 1993. Association for Computing Machinery.

[WBB+90] C. Wiecha, W. Bennett, S. Boies, J. Gould, and S. Greene. ITS: A tool for rapidly developing interactive applications. *ACM Transactions on Information Systems*, 8(3):204–236, July 1990.

[WG92] John Warnock and Chuck Geschke. *PostScript Language Reference Manual.* Adobe Systems Incorporated, Addison-Wesley, Menlo Park, California, 7th edition, November 1992.

[WH89] Patrick Henry Winston and Berthold Klaus Paul Horn. *Lisp.* Addison-Wesley, Reading, Massachusetts, 3rd edition, 1989.

[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, MA, November 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.

A Throttle artist example

```
--Author:      Robert S. Zucker (rzucker@bonnie.ics.uci.edu)
--Date:        13 April 1992
--Subject:     Throttle_Module_Artist.cal
--Note:        Adapted from David Levine's Ada source and Brewster,
--Larry Thomas, "Modeling Flight", IEEE Potentials 9:2,
--April 1990, pages 34-41; and from Brewster's Pascal
--source: "Flight Simulator, Subsonic Jet Aircraft, Version
--4.26" dated February 26, 1990.  Brewster made his
--software available only to individuals and only for
--educational purposes, and strictly prohibits any and all
--commercial use.

with Wrapper_Throttle_Module;
with Client_Dispatcher;

with TEXT_IO;

package body Throttle_Module_Artist is


  task body Throttle_Module_Artist is

    Self_Ptr       : Throttle_Module_Artist_Ptr;
    Local_Artist_ID : CSL.Artist_ID_Type;
    Local_Display   : CSL.Str;

    --<< declare artist objects here. >>

    Artist_Frame          : ~CSL.ADL_Base_Frame;
    Artist_Panel          : ~CSL.ADL_Panel;
    Throttle              : ~CSL.ADL_Slider;

    Slider_Behavior       : ~Behavior_Array_Type := (
                            Others => System.No_Addr);


    --<< declare handler routines for both client and server >>
    --<< events plus any auxilliary routines here.          >>
    --
    --server events handlers must have the signature:
    --procedure <handler_name> (Object : CSL.Object_Type;
    --Event  : CSL.Chiron_Event_Ptr);
    --
    --client events handlers must have the signature:
    --procedure <handler_name> (Event : Client_Event_Ptr);

    procedure Handle_Slider (Object: ~CSL.Object_Type;
                             Event: ~Chiron_Event_Ptr) is
      Float_Result: Float;

    begin

      --User has moved the slider, get the new value from the event
      --information and convert it to type Float range 0.0 to 1.0.
      Float_Result := Float(Event.Num_Val) / 100.0;

      --Update throttle ADT
      Wrapper_Throttle_Module.Adjust_Throttle (Float_Result);
```

```
  end Handle_Slider;                                                    60
                                                                        61
                                                                        62
                                                                        63
begin --task body                                                       64
                                                                        65
  accept Start_Artist (                                                 66
         ID   : CSL.Artist_ID_Type;                                     67
         Self_Aptr : address;                                           68
         Display_Name : CSL.Str) do                                     69
    Self_Ptr := address_to_artist(Self_Aptr);                           70
    Local_Artist_ID := ID;                                              71
    Local_Display := Display_Name;                                      72
                                                                        73
    --<< register interests in client events with the >>               74
    --<< appropriate dispatchers here.              >>                 75
                                                                        76
    --this artist does not receive ADT-based events                    77
                                                                        78
  end Start_Artist;                                                     79
                                                                        80
                                                                        81
  --<< create initial graphical objects here. >>                       82
  Text_Io.Put_Line ("Throttle Artist");                                 83
                                                                        84
  --Upon startup, a base frame will be created that will               85
  --be used to hold panel window.                                      86
                                                                        87
  Artist_Frame := ~Apply(ADL_Base_Frame,                                88
        create,                                                         89
        Frame_Label          => "Throttle Control",                     90
        Show_Footer          => True,                                   91
        X                  => 10,                                       92
        Y                  => 773);                                     93
                                                                        94
  --Upon startup, a panel window will be created that will             95
  --be used to hold the slider.                                        96
                                                                        97
  Artist_Panel := ~Apply(ADL_Panel,                                     98
        create,                                                         99
        Parent             => Artist_Frame,                            100
        Foreground          => Black,                                  101
        Background          => Black);                                 102
                                                                       103
  --Upon startup, a slider will be created with the following         104
  --attributes.                                                       105
                                                                       106
  Throttle := ~Apply (ADL_Slider,                                      107
        create,                                                        108
        Parent             => Artist_Panel,                            109
        Label              => "Throttle",                              110
        Slider_Value        => 43,                                     111
        Min_Value           => 6,                                      112
        Max_Value           => 100,                                    113
        Show_Value_Field     => True,                                  114
        Foreground         => White,                                   115
        Slider_Width       => 200);                                    116
                                                                       117
  ~Apply(ADL_Panel, Artist_Panel, ADL_window_fit);                     118
  ~Apply(ADL_Base_Frame, Artist_Frame, ADL_window_fit);                119
                                                                       120
```

```
                                                                        121
    --<< set behaviors of graphical objects in response >>              122
    --<< to server events here.                          >>             123
                                                                        124
    Slider_Behavior(Select_Event) := Handle_Slider'ADDRESS;             125
                                                                        126
    ~Set_Behavior (ADL_Slider, Slider_Behavior);                        127
                                                                        128
                                                                        129
    --<< call adl start_processing method here. >>                     130
                                                                        131
    ~Apply(ADL_Base_Frame, Artist_Frame, start_processing);             132
                                                                        133
                                                                        134
    loop                                                                135
      select                                                            136
        accept Notify_Client_Event (                                    137
                Client_Event    : Client_Events.Client_Event_Ptr;       138
                Handler_Routine : address);                             139
      or                                                                140
        accept Notify_Server_Event (                                    141
                Object          : CSL.Object_Type;                      142
                Server_Event    : CSL.Chiron_Event_Ptr;                 143
                Handler_Routine : address);                             144
      or                                                                145
        accept Terminate_Artist do                                      146
          --destroy all graphical objects                               147
          ~Apply(ADL_base_frame,                                        148
                artist_frame,                                           149
                destroy,                                                150
                artist_frame);                                          151
        end Terminate_Artist;                                           152
                                                                        153
      end select;                                                       154
    end loop;                                                           155
                                                                        156
  end Throttle_Module_Artist;                                           157
                                                                        158
end Throttle_Module_Artist;                                             159
                                                                        160
```