

# Chiron 1.4 Client Design

Kari A. Nies

*Arcadia Document UCI-93-02*

Department of Information and Computer Science  
University of California, Irvine \*

July 27, 1993

## **Abstract**

The purpose of this document is to present the design and rationale for the *Chiron 1.4* client. High-level and low-level descriptions of the new and modified architectural components are given as is a description of the new client toolset.

---

\*This material is based upon work sponsored by the Defense 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.

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Chiron 1.4 client architecture</b>	<b>6</b>
2.1	Client events . . . . .	8
2.2	Universal artist type . . . . .	10
2.3	Artist Manager . . . . .	12
2.4	Wrapper . . . . .	12
2.5	Access Controller . . . . .	13
2.6	Dispatchers . . . . .	15
2.7	Artist . . . . .	15
2.7.1	Artist template . . . . .	17
2.7.2	Event processing . . . . .	20
2.8	Client Initializer . . . . .	20
<b>3</b>	<b>Chiron 1.4 client toolset</b>	<b>23</b>
3.1	Client configuration . . . . .	23
3.2	Artist manager generator . . . . .	24
3.3	Client initializer generator . . . . .	24
3.4	Client events generator . . . . .	25
3.5	Wrapper generator . . . . .	25
3.6	Dispatcher generator . . . . .	26
3.7	Artist template generator . . . . .	26
3.8	Client builder . . . . .	26
3.9	LoCAL processor . . . . .	27
3.10	Compiling and loading clients . . . . .	27
3.11	Executing clients . . . . .	28
<b>4</b>	<b>Unresolved issues</b>	<b>29</b>
	<b>References</b>	<b>29</b>
<b>A</b>	<b>Dialogue artist example</b>	<b>31</b>
<b>B</b>	<b>Execution of client_builder for flight simulator example</b>	<b>36</b>

**List of Figures**

1	Chiron 1.4 client runtime architecture . . . . .	6
2	Simple stack ADT . . . . .	8
3	Client event type . . . . .	9
4	Addition client event declarations . . . . .	10
5	Universal artist . . . . .	11
6	Artist manager interface . . . . .	13
7	Wrapper for stack ADT . . . . .	14
8	Wrapper's implementation of Pop . . . . .	14
9	Access Controller generic . . . . .	16
10	Client dispatcher interface . . . . .	16
11	Chiron 1.4 dispatching architecture . . . . .	17
12	Stack artist specification . . . . .	18
13	Stack artist template . . . . .	19
14	Event processing code . . . . .	21
15	Example client configuration file . . . . .	24

## 1 Introduction

This document is intended to present the design and rationale for the *Chiron.1.4* client. The *Chiron* server is not discussed here as there are no major changes to its design. This is intended to serve as a reference for *Chiron* developers. The presentation assumes some understanding of the original *Chiron-1* client, however, those with little client experience should still benefit from descriptions of the high-level components and the client toolset.

Section 2 examines the architectural components of the new design. First the new client runtime architecture is given, then new components are described in detail as well as modifications to existing components. Section 3 describes the set of tools that support the the new client architecture. Finally, section 4 lists unresolved issues.

The remainder of this section discusses rationale for the re-design of the *Chiron* client. The re-design was prompted by problems identified within the original *Chiron-1* client implementation. Some represent issues unaddressed in the original design, some represent unimplemented design features, while others represent implementation issues. Below we briefly enumerate problems with the original *Chiron-1* client.

**Artist/ADT binding** Artists are tightly bound to a single ADT. The definition (or interface) of an artist is defined by a single ADT, making it impossible to build an artist that can monitor the states of multiple ADTs. Dispatchers can only communicate with a single artist type (or more precisely, with artists whose interfaces exactly correspond with a specific ADT). As a result, the client events<sup>1</sup> that can be monitored by an artist are strictly limited to operations on a single ADT. They can not encompass multiple ADTs and can not be extended to support non-ADT related events.

**No dynamism** The original design for pre-defined artists described in the *Chiron-1: Concept and Design* document [BCJ<sup>+</sup>89] was never implemented. In fact, there is no interface by which new instances of pre-compiled artists can be invoked at runtime.

**Artist deadlock** All event processing within an artist is performed within the main task body select loop. This causes a high propensity for deadlock. In particular, callbacks were easily blocked on ADT calls waiting to notify the same artist. This problem was “fixed” by an inelegant work around that involved passing an additional *In\_Callback* flag when making ADT calls through the dispatcher. Also, self notification will always result in deadlock. This problem is avoided by ensuring that artists that make an ADT call though the dispatcher will not be notified of the operation. This however results in a duplication of code, requiring that the artist update code for a particular operation be repeated both in the artist entry for that operation and at the point the operation is called from a within a callback routine.

---

<sup>1</sup>Note that here and throughout we use the term client events to describe events that originate within the client process as opposed to the server process.

**Event registration/broadcast** In the *Chiron-1* design artists simply register their existence with the dispatcher (actually this is performed by the client manager and not the artists themselves). Broadcast is done on an all or nothing basis, determined by an arbitrary *Broadcast* flag to the dispatcher. Artists should be able to register interest in explicit operations and broadcast should be filtered by registration.

**ADT locking mechanism** The current locking mechanism to control access to ADTs is ad hoc and inconsistent. ADT locking is determined by an arbitrary *Broadcast* flag to the dispatcher (TRUE implies a write operation, FALSE implies read operation). This convention must be followed by the artist developer and used consistently. It also precludes ADT read operation from being broadcast as events.

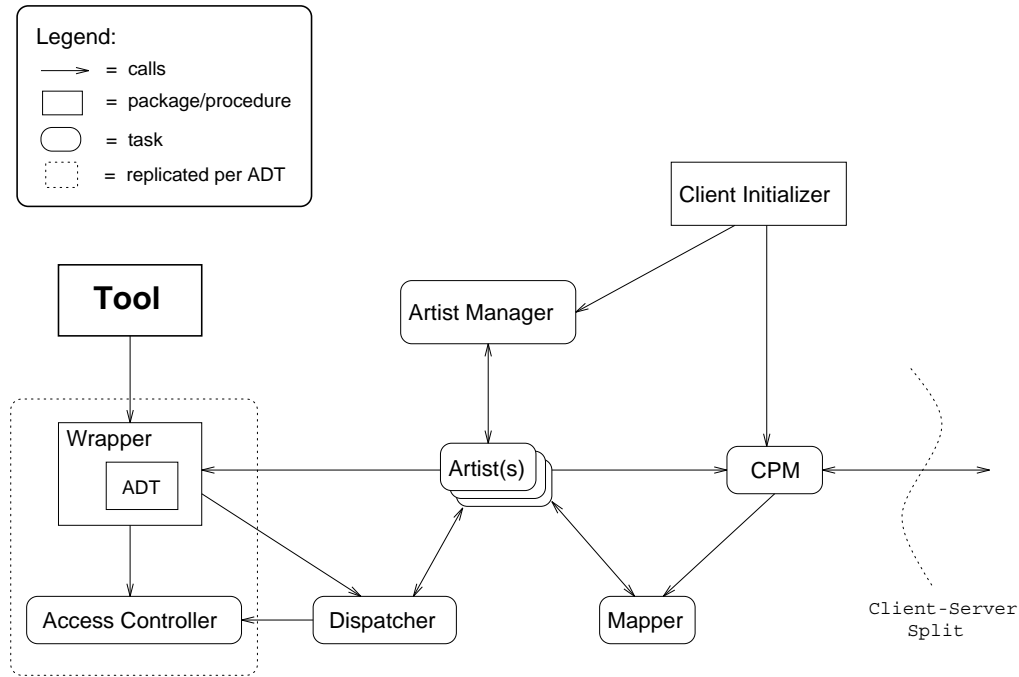


Figure 1: Chiron 1.4 client runtime architecture

## 2 Chiron 1.4 client architecture

This section describes the architectural components that make up the new *Chiron* client design. The runtime architecture of a *Chiron.1.4* client is illustrated in figure 1. Below are brief descriptions of the main components, followed by more detailed descriptions in the following subsections. Some components are new, while others are modifications of existing components. Components with no significant changes have been omitted from the detailed discussion. All components save the tool, mapper, and CPM are generated.

**Client Initializer** The Client Initializer is now responsible for bringing up the initial client configuration. This consists of starting up the CPM and invoking artist instances. The Client Initializer takes the place of the Client Manager, with two important distinctions. First, it is no longer loaded as the main executable of the client. The main executable is now the application or tool. Second, it will attempt to read the client configuration file at initialization, allowing configurations to be modified without recompilation. There is one Client Initializer per client.

**Artist Manager** The Artist Manager is a new component. It provides an interface through which new instances of pre-existing artists can be invoked dynamically. 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 per client.

**Tool** The tool is the main procedure of the application 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* the Client\_Init package. Otherwise it remains unchanged. There is only one tool per client.

**Artist** The basic functionality of the artist remains unchanged. Artists maintain the graphical depictions of ADTs. Notifications of changes in ADT state and possibly other events originating within the client are routed from a dispatcher and notifications of server events are routed from the Mapper. Artists manipulate their graphical depictions by making calls to the server which are routed through the CPM. They may now receive ADT events originating from multiple ADTs, allowing a single artist to maintain a depiction of multiple ADTs. The interface and internal structure of an artist has changed considerably. There may be multiple artists within a client.

**Wrapper** The Wrapper is a new component. It contains a subset of the functionality of the original *Chiron-1* dispatcher: mainly those aspects not directly related to dispatching. The Wrapper exports an interface identical to the ADT. It forms a “wrapper” around the ADT which intercepts calls to the ADT, protects its data structures from concurrent access, invokes the requested operation, and notifies the Client Dispatcher of the event. Both the tool and the artists indirectly call the ADT through the Wrapper. There is one wrapper for each depicted ADT.

**Access Controller** The Access Controller is used by the Wrapper to ensure CREW (Concurrent Read Exclusive Write) access to an ADT. The Wrapper obtains a read or write lock from the Access Controller before accessing the ADT. In addition, as in the previous design, when a write operation is performed all additional write operations are blocked until each interested artist has been notified. This lock is obtained and released by the dispatcher. There is one Access Controller for each depicted ADT.

**Dispatcher** The Client Dispatcher routes events from Wrappers to the appropriate artists. Artists register directly with the Client 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. The default architecture provides one centralized Client Dispatcher per client. However, it is possible to generate dedicated dispatchers for ADTs. If such an ADT dispatcher exists, the Client Dispatcher, will forward all calls pertaining to a particular ADT to its corresponding dispatcher.

**Mapper** The Mapper routes server events to the appropriate artists. Its functionality is unchanged. The mapper is provided in the client library.

**CPM** The CPM, Client Protocol Manager, handles communication from the *Chiron* server. Its implementation remains unchanged. The CPM is provided in the client library.

For the remainder of this section, we will discuss the new or modified components of the client architecture in detail. But first we must discuss the type structure that enables this new architecture. All examples will refer to the familiar stacks dialogue example given in the *Chiron-1* Users Manual [CFM92]. The ADT interface for this example is given in figure 2.

---

```

package Stacks is
  type Stack is private;
  Stack_Empty: exception;
  function Create (Size : INTEGER) return Stack;
  procedure Destroy (S : in out Stack);
  procedure Push (S : in out Stack;
                 X : in INTEGER);
  function Pop (S : Stack) return INTEGER;
  function Top (S : Stack) return INTEGER;
  function Depth (S : Stack) return INTEGER;
  function Max (S : Stack) return INTEGER;
private
  .
end Stacks;

```

Figure 2: Simple stack ADT

---

## 2.1 Client events

The key to the *Chiron.1.4* client design is the introduction of the client event type. It transforms the notion of an event from a call to a first class entity. Events can now be easily passed between components without the requirement that they export ADT interfaces, and different kinds of events can be implicitly selected and reasoned about.

The client event type is generated from the set of ADTs that a client will be depicting (with one or more artists). An example of a client event type generated from the stack ADT in figure 2 is shown in figure 3.

Type *Client\_Event\_Kind* enumerates all possible client events. Each enumeration is given as the ADT name appended with the operation name. If a client depicts more than one ADT, all of the operations from each ADT will be enumerated within this type. If the ADT overloads an operator, the second occurrence will be appended with a 2, and the third with a 3, and so on.

The *Client\_Event\_Type* is defined as a record discriminated by the *Client\_Event\_Kind*. Each variant of the record specifies the data associated with each event kind, where the data corresponds to the formal parameters of the corresponding ADT operation. Each field is given as the *Client\_Event\_Kind* name appended by the parameter name, and each field type is the type of the corresponding formal parameter.

The client event type may be extended by the artist writer to include any event. For example, if an artist needs to pass a canvas to another artist, it could define an event where the data passed is an *ADL\_Object*. In fact, events need not be based on ADT operations at all. If no ADT is specified, the client event will include only the *Null\_Event*. The type structure of the client will be satisfied, and no wrappers or access controllers need be

---

```

with Stacks; use Stacks;
package Client_Events is
  type Client_Event_Kind is (
    Stacks_Create,
    Stacks_Destroy,
    Stacks_Push,
    Stacks_Pop,
    Stacks_Top,
    Stacks_Depth,
    Stacks_Max,
    Null_Event
  );

  type Client_Event_Type (Kind : Client_Event_Kind := Null_Event) is record
    case Kind is
      when Stacks_Create =>
        Stacks_Create_Size: INTEGER;
        Stacks_Create_Result: Stack;
      when Stacks_Destroy =>
        Stacks_Destroy_S: Stack;
      when Stacks_Push =>
        Stacks_Push_S: Stack;
        Stacks_Push_X: INTEGER;
      when Stacks_Pop =>
        Stacks_Pop_S: Stack;
        Stacks_Pop_Result: INTEGER;
      when Stacks_Top =>
        Stacks_Top_S: Stack;
        Stacks_Top_Result: INTEGER;
      when Stacks_Depth =>
        Stacks_Depth_S: Stack;
        Stacks_Depth_Result: INTEGER;
      when Stacks_Max =>
        Stacks_Max_S: Stack;
        Stacks_Max_Result: INTEGER;
      when Null_Event => null;
    end case;
  end record;

  type Client_Event_Ptr is access Client_Event_Type;

  :
  :
  :
end Client_Events;

```

Figure 3: Client event type



---

```

type Client_Event_Sources is (Stacks, None);
Client_Event_Source : constant array (Client_Event_Kind)
  of Client_Event_Sources := (
  Stacks_Create => Stacks,
  Stacks_Destroy => Stacks,
  Stacks_Push => Stacks,
  Stacks_Pop => Stacks,
  Stacks_Top => Stacks,
  Stacks_Depth => Stacks,
  Stacks_Max => Stacks,
  Null_Event => None
);
type Client_Event_Modes is (Read, Write, None);
Client_Event_Mode : constant array (Client_Event_Kind)
  of Client_Event_Modes := (
  Stacks_Create => Write,
  Stacks_Destroy => Write,
  Stacks_Push => Write,
  Stacks_Pop => Write,
  Stacks_Top => Read,
  Stacks_Depth => Read,
  Stacks_Max => Read,
  Null_Event => None
);

```

Figure 4: Addition client event declarations

---

generated.

In addition to the client event type, the `Client_Events` package also defines two arrays that hold useful information about the various client events. These definitions for the same stack ADT are given in figure 4. The `Client_Event_Source` array is used by the Client Dispatcher to determine which ADT an event originated from (if any). It is used to locate the appropriate ADT `Access_Controller`.

The `Client_Event_Mode` is used by the Wrapper to automatically determine whether an ADT event corresponds to a read or write operation. In the previous design, this was determined by a Broadcast flag that was added as an additional parameter to each ADT operation. (TRUE implied a write operation and FALSE implied a read operation). But the artist writer had to set this flag correctly and consistently for each ADT call.

Since it would take a substantial amount of semantic analysis to determine this information, the values in the array are all initially set to Write. It is the artist writers's responsibility to set these values correctly. Failure to do this could result in runtime deadlock.

## 2.2 Universal artist type

The definition of an explicit client event type, allows the specification of a universal artist type (see figure 5). In the previous design, client events were communicated to artists by making entry calls to the artist interface. There was exactly one entry corresponding to each operation in the ADT. This convention resulted in an artist type structure that was tightly bound to a single ADT.

---

```

with Chiron_Standard_Library;
with SYSTEM;

package Universal_Artist is

  type Universal_Client_Event_Ptr is private;

  package CSL renames Chiron_Standard_Library;

  task type Universal_Artist is

    entry Start_Artist (
      Artist_ID : CSL.Artist_ID_Type;
      Artist_Ptr : SYSTEM.ADDRESS;
      Display_Name : CSL.Str);

    entry Notify_Client_Event (
      Client_Event : Universal_Client_Event_Ptr;
      Handler_Routine : SYSTEM.ADDRESS);

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

    entry Terminate_Artist;

  end Universal_Artist;

  type Universal_Artist_Ptr is access Universal_Artist;

private
  type Hidden;
  type Universal_Client_Event_Ptr is access Hidden;
end Universal_Artist;

```

Figure 5: Universal artist

---

All artists now export the same interface – that of the universal artist. The interface is no longer defined by the operations on an ADT. Instead, only four entries are provided, one to start the artist, one to terminate the artist, one to receive client events, and one to receive server events. Now all artists can be type cast to the universal artist type and entry calls can be made through a universal artist without type casting it back to its original type.

Note that the definition of the *Universal\_Client\_Event\_Ptr* is not given in the *Universal\_Artist* specification, but deferred to the package body where it is declared as a pointer to the client event type. This avoids recompilation of the universal artist (and subsequently the Mapper and CPM) for each new client event type. The specification of the *Universal\_Artist* package is defined in the client library. The body is inserted into the same file as the *Artist\_Manager* body in order to avoid an additional compilation.

The universal artist type gives a solution to several typing problems in the original design. For example, the original dispatcher had to register artists that, although they monitored the same ADT, were not necessarily of the same task type. This was handled by declaring a generic task type in the dispatcher specification. The generic task type exported the same interface so that all artists for a particular ADT could be type cast to it. All task types were converted to this generic task type at the point of registration with the

dispatcher.

The mapper had to manage artists of any type for any ADT. This was done by converting artists to addresses before passing them to the mapper. However when the mapper had to call into an artist's callback entry in order to notify it of a server event, the mapper could not determine on which artist task type to cast the address. This problem was evaded by passing the mapper the address to a *callback\_proc* procedure located in the outer artist package. The mapper could make an address call (through *c*) to this procedure passing it the artist address and the address to the callback procedure. From within this procedure it was possible to type cast the artist address to the appropriate artist type and invoke the callback procedure through the callback entry.

The introduction of the universal artist type gave a clean solution to both of the above typing problems. The dispatcher no longer needs to declare a generic task type since all artists can be type cast to a single universal type. Dispatchers now store registered artists as universal artist types and can communicate with any artist by calling its *Notify\_Client\_Event* entry, allowing artists to receive notification for multiple ADTs. The mapper can also store all artists as universal task types. This means that the mapper no longer needs to call an artist indirectly through a procedure declared in its outer package, but can pass the address of the callback procedure directly to the artist through its *Notify\_Server\_Event* entry.

### 2.3 Artist Manager

The Artist Manager provides a new interface through which new instances of artist may be invoked and shutdown at runtime. Figure 6 shows an Artist Manager specification for a client that includes two artists, *Dialogue\_Artist* and *Cafeteria\_Artist*, giving two different depictions of a stack.

The Client Initializer uses this interface to dynamically configure a client. Artists can use this artist to invoke other artists as well. Only one Artist Manager is generated per client from a list of possible artists within the client.

### 2.4 Wrapper

The Wrapper plays the role of a listener and a notifier. It listens for ADT events and relays them to the Client Dispatcher. 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. Given the stack ADT of figure 2, a wrapper generated for that ADT is shown in figure 7. There is exactly one wrapper generated per depicted ADT.

Notice that the three additional parameters that are present in the *Chiron 1.3* dispatcher, *Broadcast*, *Artist\_ID*, and *In\_Callback*, are no longer needed. The *Broadcast* flag is no longer necessary because broadcasting is now registration-based and handled entirely by the new Dispatcher and the read and write locks are obtained based on the *Client\_Event\_Mode* information defined in the *Client\_Events* package. The *Artist\_ID* was only used by the dispatcher to avoid notifying an artist of any ADT calls made by that artist. This would cause the *Chiron 1.3* client to deadlock. *Chiron.1.4* allows artists to be notified of their own ADT events, so this information is no longer required. Finally, the

---

```

with Universal_Artist;
use Universal_Artist;
with Chiron_Standard_Library;

package Artist_Manager is

  package CSL renames Chiron_Standard_Library;

  type Artist_Types is (Dialogue_Artist_type, Cafeteria_Artist_type);

  function Invoke_Artist (New_Artist : Artist_Types;
                        Display_Name : Str := To_Str("unix:0"))
    return CSL.Artist_ID_Type;

  procedure Shutdown_Artist (Artist_ID : CSL.Artist_ID_Type);

  function Get_Artist_Ptr (Artist_ID : CSL.Artist_ID_Type)
    return Universal_Artist_Ptr;

  Invalid_Artist_ID : exception;

end Artist_Manager;

```

Figure 6: Artist manager interface

---

In\_Callback flag was added as a temporary fix for a design flaw in the artist. It is also obviated by the new client architecture.

The Wrapper's implementation of the *Pop* operation is given in figure 8. First the Wrapper creates an event corresponding to the Pop operation. Before calling the ADT to perform the operation, it obtains the proper lock from the ADT's Access Controller. After the operation is invoked, the input and output data is inserted into the event which is then shipped to the Dispatcher. The ADT lock is then released. The ADT lock is not released until after the Dispatcher is notified for a reason. If the Dispatcher has at least one artist to notify of this event, it will obtain a special lock from the Access Controller to block out all write operations (not read operations) until all interested artists have updated their depictions. It is important that no write operations are allowed before the dispatcher can grab this lock. The Dispatcher performs the actual artist notification outside of the *Notify\_Artists* rendezvous, so that read operations will not be blocked while artist notification is in being performed.

## 2.5 Access Controller

An Access Controller is generated for each depicted ADT. Each Access Controller is declared as a generic instantiation of generic package ADT\_Controller (see figure 9) in the client library. The seize and release lock procedures are used by the wrapper to ensure CREW access to the ADT and the artist update procedures are used by the Client Dispatcher (or an dedicate ADT dispatcher) to ensure that following and ADT write operation, all additional writes (not reads) are blocked until all interested artists have updated their depictions in response to the original write event. The Dispatcher obtains this lock while passing the number of artists that will be notified of the update. The Dispatcher then notifies the Access Controller whenever an artist has completed its update. When the total number

---

```

with Stacks;
package Wrapper_Stacks is
  subtype stack is Stacks.stack;
  stack_empty : exception renames Stacks.stack_empty;
  function create(
    size : integer) return stack;
  procedure destroy(
    s : in out stack);
  procedure push(
    s : in out stack;
    x : in integer);
  function pop(
    s : stack) return INTEGER;
  function top(
    s : stack) return INTEGER;
  function depth(
    s : stack) return INTEGER;
  function max(
    s : stack) return INTEGER;
end Wrapper_Stacks;

```

Figure 7: Wrapper for stack ADT

```

function Pop(
  S : Stack) return INTEGER is
  Result : INTEGER;
  Event : Client_Event_Ptr;
begin
  Event := new Client_Event_Type(Stacks.Pop);
  --get the appropriate ADT lock
  Get_ADT_Lock (Stacks.Pop);

  --call the adt and initialize adt_event record
  Result := Stacks.Pop(S);
  Event.Stacks_Pop_S := S;
  Event.Stacks_Pop_Result := Result;

  Client_Dispatcher.Dispatcher.Notify_Artists(Event);
  Release_ADT_Lock (Stacks.Pop);
  return Result;
exception
  when others =>
    Release_ADT_Lock (Stacks.Pop);
    raise;
end Pop;

```

Figure 8: Wrapper's implementation of Pop

of updating artists have responded, the Access Controller releases the lock automatically.

## 2.6 Dispatchers

The Client Dispatcher routes client events to the appropriate artists. Its interface is given in figure 10. Generally, events are detected by the Wrapper and forwarded to the Client Dispatcher via a call to its *Notify\_Artists* entry, but hand-coded non-ADT events can be sent to the Client Dispatcher from any component, including artists.

Artists register the events for which they want to be notified by calling the Client Dispatcher's *Register\_Event* entry. They provide their artist id, the event kind that they are interested in, and the address of a handling routine for that event. Client event registration is now somewhat analogous to server event registration via a call to *Set\_Behavior*. Artists can discontinue notification of an event at any time by making a call to the *Unregister\_Event* entry. The Dispatcher only notifies Artists of events for which they have pre-registered.

A dispatcher must also ensure that once an ADT write operation is performed all subsequent write operations are blocked until all interested artist have updated their depictions. This is done by calling the ADT's Access Controller *Start\_Artist\_Update* procedure, giving it the number of artists that will be notified of the update. Artists indicate to the Dispatcher that they are finished updating by calling the *Notify\_Done* entry and the Client Dispatcher in turn calls the ADT's Access Controller *Artist\_Update\_Complete* procedure which will allow write operations once again once all notified artists have registered their completion.

By separating out ADT-specific functionality into a separate Wrapper entity it is now possible to allow for a more flexible dispatching architecture. The default architecture is a single centralized Client Dispatcher. All artists and wrappers *always* communicate with the Client Dispatcher. In addition, it is possible to generate a dedicated dispatcher for one or more ADTs. If such an ADT dispatcher exists, the Client Dispatcher, will forward all calls pertaining to a particular ADT to its corresponding dispatcher. A dedicated ADT dispatcher will keep track of event registration for its own ADT, will perform artist notification, and will ensure that the write lock is maintained while artists are updating. By stipulating that all dispatching calls are directed to the Client\_Dispatcher, we can experiment with different architectures without modifying dependent code. Figure 11 illustrates the new dispatching architecture.

## 2.7 Artist

Artists have not changed conceptually in this design. Their purpose is still to create and maintain graphical depictions in response to client and server events. The means by which graphical objects are created and manipulated has not changed at all. The LoCAL language is identical as is the LoCAL translation of ADL commands.

What has changed is the concept of a client event and the means by which both client events and server events are processed. Event processing now has a consistent model within the client. Both client and server events are communicated by passing event objects to the artist through a simplified, universal interface. Artists explicitly register for events. Client events are registered with the Client Dispatcher and server events are registered with the

---

```

generic
package ADT_Controller is

    --Prevents writes to the ADT, allows reads to the ADT
    procedure Seize_ADT_Read_Lock;
    procedure Release_ADT_Read_Lock;

    --Prevents reads and writes to the ADT
    procedure Seize_ADT_Write_Lock;
    procedure Release_ADT_Write_Lock;

    --Prevents any ADT writes until all artists have updated their depiction
    procedure Start_Artist_Update (Number_Of_Artists_Updating : NATURAL);
    procedure Artist_Update_Complete;

end ADT_Controller;

```

Figure 9: Access Controller generic

---

```

with Chiron_Standard_Library;
with Client_Events;
with SYSTEM;

package Client_Dispatcher is

    package CSL renames Chiron_Standard_Library;

    task Dispatcher is

        entry Register_Event (
            Artist_ID : CSL.Artist_ID_Type;
            Event_Kind : Client_Events.Client_Event_Kind;
            Handle_Routine : SYSTEM.ADDRESS);

        entry Unregister_Event (
            Artist_ID : CSL.Artist_ID_Type;
            Event_Kind : Client_Events.Client_Event_Kind);

        entry Notify_Artists (
            Event : Client_Events.Client_Event_Ptr);

        entry Notify_Done (
            Event : Client_Events.Client_Event_Ptr);

    end Dispatcher;

end Client_Dispatcher;

```

Figure 10: Client dispatcher interface

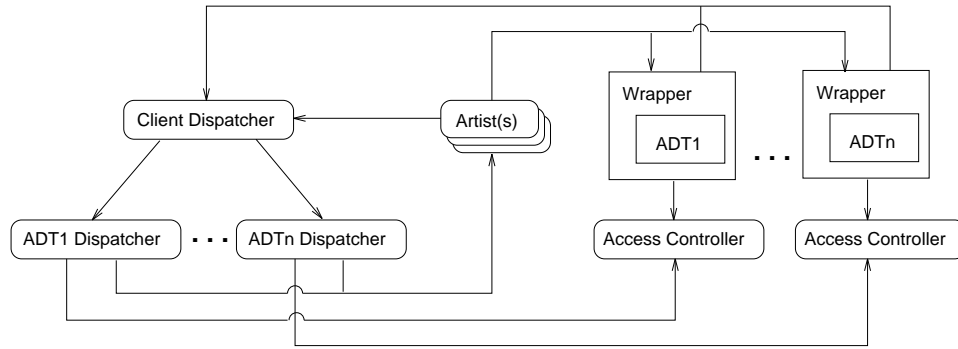


Figure 11: Chiron 1.4 dispatching architecture

Mapper. The artist passes the address of a callback routine for handling the event as part of registration. Event broadcasting is filtered by registration and events are handled by invoking the callback routines specified at registration. Events are now processed in a separate thread of control from the main artist event loop.

### 2.7.1 Artist template

The artist template is now generated into two separate files, one containing the artist specification which is never modified, and the other containing the body of the artist into which the artist write inserts new LoCAL code. An example of the new artist template for the stack dialogue artist example is given in figures 12 and 13. Note that the specification of the artist (fig. 12) exports the same interface as the Universal Artist (fig. 5). It consists of four entry calls: 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. The *Notify\_Server\_Event* entry is identical to the *Callback* entry in the original client design. It is called only by the Mapper in response to a server event. The *Notify\_Client\_Event* entry replaces what used to be one entry per single ADT operation. Now an artist can receive any client event specified in the *Client\_Events* type. The *Notify\_Client\_Event* entry is invoked by a dispatcher in response to a client event. The *Start\_Artist* and *Terminate\_Artist* entries are invoked only by the Artist Manager.

The artist template body (fig. 13) contains directives to the artist writer on how to define an artist. Appendix A gives the completed LoCAL listing for the dialogue artist.

First the graphical objects and any local data are declared. Neither the ADL hierarchy nor the LoCAL syntax has changed. Therefore this step remains the same for *Chiron 1.4* artists as it was for previous *Chiron-1* artists.

Handling (callback) routines are then defined for both client and server events. Note that in the previous design, only server events were handled in this manner. Client events were handled in the accept bodies of the ADT operation entries within the main artist select loop. All client and server callbacks must have the signatures specified within this block of comments. When defining server event handling routines, it is important to note that, in the *Chiron.1.4* client, artists are now notified of any ADT operations that they themselves

---

```

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

package Dialogue_Artist is

  package CSL renames Chiron_Standard_Library;

  task type Dialogue_Artist is

    entry Start_Artist (
      ID      : CSL.Artist_ID_Type;
      Aptr    : SYSTEM.ADDRESS;
      Display_Name : CSL.Str);

    entry Notify_Client_Event (
      Client_Event : Client_Events.Client_Event_Ptr;
      Handler_Routine : SYSTEM.ADDRESS);

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

    entry Terminate_Artist;

  end Dialogue_Artist;

  type Dialogue_Artist_Ptr is access Dialogue_Artist;
end Dialogue_Artist;

```

Figure 12: Stack artist specification

---

perform. This was not true in previous versions of the client, where self notification would result in deadlock. This meant that if an ADT call was made from within a callback, the code to update the corresponding depiction would have to be included immediately after the ADT call as well as in the corresponding accept body. This resulted in a good deal of code replication, making artist more difficult to reason about and to maintain. Now the code to update a depiction in response to an ADT call is specified solely in the handling routine for the corresponding client event.

An artist should initially register for client events within its *Start\_Artist* accept body. Client event registration and unregistration can occur anywhere, but placing the initial registration calls in the *Start\_Artist* accept body ensures that no events will be missed. This is because, at client initialization, all artists are started before the main tool is invoked. Registration is achieved by making a call to the Dispatcher's *Register\_Event* entry, passing it the local artist id, the client event kind, and the address of a callback routine to handle the event.

The LoCAL code for creating instances of graphical objects, describing their behaviors (specifying callback routines) in response to server events via calls to *Set\_Behavior* (server event registration), and invoking the *Start\_Processing* method on the artist's base frame remains unchanged from previous versions of the *Chiron-1* client.

The *Terminate\_Artist* entry may be edited in include code that will be invoked before

---

```

with Wrapper_Stacks;
with Client_Dispatcher;

package body Dialogue_Artist is
  task body Dialogue_Artist is
    --<< declare artist objects here. >>

    --<< declare handler routines for both client and server >>
    --<< events plus any auxiliary 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;
    Aptr : SYSTEM.ADDRESS;
    Display_Name : CSL.Str) do
    Self_Ptr := address_to_artist(Aptr);
    Local_Artist_ID := ID;
    Local_Display := Display_Name;

    --<< register interests in client events with the >>
    --<< appropriate dispatchers 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 : SYSTEM.ADDRESS);
  or
  accept Notify_Server_Event (
    Object : CSL.Object_Type;
    Server_Event : CSL.Chiron_Event_Ptr;
    Handler_Routine : SYSTEM.ADDRESS);
  or
  accept Terminate_Artist;
  end select;
  end loop;

  end Dialogue_Artist;
end Dialogue_Artist;

```

Figure 13: Stack artist template

the Artist Manager attempts to shut it down. This should include unregistering for client events and destroying graphical objects created by the artist.

The artist code that actually processes client and server events once the artist is notified of them is omitted from the artist template. Note that no accept bodies are given for *Notify\_Client\_Event* and *Notify\_Server\_Event*. This code is inserted by the LoCAL translator (see section 2.7.2).

### 2.7.2 Event processing

In the original *Chiron-1* design, all event processing was done in the main artist select loop. This proved to be problematic because other event notifications were blocked while an event was being processed. This led to a whole array of deadlock situations. This problem has been corrected in the *Chiron 1.4* client design by giving event processing its own thread of control.

The event processing code is not present in the artist template, but is inserted by the LoCAL translator. The actual event processing code that would be inserted into the dialogue artist example is listed in figure 14.

The artist maintains two concurrent unbounded buffers, one for the storage of unprocessed client events and the other for the storage of unprocessed server events. When the artist is notified of an event, it simply bundles up all of the information about the event and places it on the appropriate buffer.

Two concurrent tasks are defined, one to process client events and the other to process server events. Each event processing task continuously removes elements from its designated buffer. In order to handle the event, the task must invoke the callback routine located at specified address. As in previous *Chiron-1* clients, this is done by making calls to machine code. The *go\_thru\_machine\_code* procedure (still generated by the LoCAL translator) is now overloaded to handle both client and server callbacks.

Once the client event processing task returns from executing the callback routine, it automatically notifies The Client Dispatcher that the update is complete. The artist writer is no longer responsible for making sure this call is made.

In the *Chiron 1.4* artist design, event notification is never blocked. Also, a higher degree of concurrency exists; client event callbacks, server event callbacks, and event notification may now (potentially) execute concurrently. It is possible, however, for event handler tasks to block. For example, if you call an ADT write operation in a client event handler routine for an event on the same ADT because the call will block waiting for the previous update to complete. This is analogous to calling a write operation from within an artist accept body in the *Chiron 1.3* design.

## 2.8 Client Initializer

The Client Initializer is a package responsible for bringing up the initial client configuration. The initialization is performed as part of the package's own elaboration. This consists of starting up the CPM and invoking artist instances. The Client Initializer takes the place of what was the Client Manager in the *Chiron 1.3* client. However, it is no longer loaded

---

```

task body Dialogue_Artist is
    :
    :
    :
    task Client_Event_Task is
    end Client_Event_Task;

    task body Client_Event_Task is
        Client_Event_Rec : Client_Event_Record;
    begin
        loop
            Client_Event_Buffer.Get(Client_Event_Rec);
            go_thru_machine_code (
                Client_Event_Rec.Handler_Routine,
                Client_Event_Rec.Client_Event);
            Client_Dispatcher.Dispatcher.
                Notify_Done (Client_Event_Rec.Client_Event);
            end loop;
        end Client_Event_Task;

    task Server_Event_Task is
    end Server_Event_Task;

    task body Server_Event_Task is
        Server_Event_Rec : Server_Event_Record;
    begin
        loop
            Server_Event_Buffer.Get(Server_Event_Rec);
            go_thru_machine_code (
                Server_Event_Rec.Handler_Routine,
                Server_Event_Rec.Object,
                Server_Event_Rec.Server_Event);
            end loop;
        end Server_Event_Task;

begin --task body
    :
    :
    :
    loop
        select
            accept Notify_Client_Event (
                Client_Event : Client_Events.Client_Event_Ptr;
                Handler_Routine : SYSTEM.ADDRESS) do
                Temp_Client_Event_Rec := (Client_Event, Handler_Routine);
            end Notify_Client_Event;
            Client_Event_Buffer.Put(Temp_Client_Event_Rec);

            or

            accept Notify_Server_Event (
                Object : CSL.Object_Type;
                Server_Event : CSL.Chiron_Event_Ptr;
                Handler_Routine : SYSTEM.ADDRESS) do
                Temp_Server_Event_Rec := (new CSL.Object_Type'(Object),
                    Server_Event, Handler_Routine);
            end Notify_Server_Event;
            Server_Event_Buffer.Put(Temp_Server_Event_Rec);

            or

            accept Terminate_Artist;

        end select;
    end loop;
end Dialogue_Artist;

```

Figure 14: Event processing code

as the main executable of the client. The main executable is now the application or tool and the tool must now *with* the Client\_Init package.

The artist configuration is obtained from the client configuration file (.ccf file) which specifies, among other things, the artist names and the number of instances of each artist that make up the initial configuration. The artists are invoked via calls to the Artist Manager. The initial artist configuration is programmed into the Client Initializer when it is generated. However, the Client Initializer differs from the Client Manager in that it will attempt to find the .ccf file from which it was generated and read in the artist configuration information at runtime before resorting to it's own static information.

## 3 Chiron 1.4 client toolset

The *Chiron 1.4* toolset includes a set of tools for the automatic generation of several client components. This set includes:

- Artist Manager generator
- Client Events generator
- Wrapper generator
- Dispatcher generator
- Artist template generator
- Client Initializer generator

With so many generated components, getting a client off the ground can be complicated and confusing. To address this, we have given each tool a programmatic interface. All of the information necessary to construct a client is specified in the *client configuration file*. A new tool called *client\_builder* is provided that can read the client configuration file and invoke the generator tools in order to construct all of the initial components of the specified client. In addition, each generator tool has a driver procedure allowing it to be invoked individually.

### 3.1 Client configuration

Users configure their clients by creating a *client configuration file*. The information in this file is used by the generator tools to create appropriate client components. It is also read during client initialization in order to determine the runtime configuration of a client. The runtime 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 15.

A client configuration file contains the following information:

**ADT specification list:** A list of the filenames, each on its own line, of all ADT specifications for 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 (see section 2.6).

**Artist descriptions:** This portion of the configuration file describes each artist type in terms of its name and any ADTs for which it will want to receive events. Each line contains an artist name followed by a list of ADT specification filenames. Each filename given must also appear in the ADT specification list.

**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 display should be used. If no display is specified, `unix:0` is the default. Each artist name given must appear on the artist description list.

---

```

aileron_spec.a
altitude_spec.a
attitude_spec.a          dispatcher
psi_spec.a                dispatcher
speed_spec.a
tail_spec.a              dispatcher
theta_spec.a             dispatcher
throttle_spec.a

Aileron_Module_Artist    aileron_spec.a
Tail_Module_Artist       tail_spec.a
Throttle_Module_Artist   throttle_spec.a

Pitch_Artist              attitude_spec.a
Roll_Artist              theta_spec.a
Altitude_Module_Artist   altitude_spec.a

Altimeter_Module_Artist  altitude_spec.a
Airspeed_Module_Artist  speed_spec.a
Compass_Module_Artist   psi_spec.a
Turn_Module_Artist      theta_spec.a
Horizon_Module_Artist   attitude_spec.a theta_spec.a

Aileron_Module_Artist    1   jasmin:0
Tail_Module_Artist       1   jasmin:0
Throttle_Module_Artist   1   jasmin:0

Pitch_Artist              0   jasmin:0
Roll_Artist              0   jasmin:0
Altitude_Module_Artist   0   jasmin:0

Altimeter_Module_Artist  1   jasmin:0
Airspeed_Module_Artist  1   jasmin:0
Compass_Module_Artist   1   jasmin:0
Turn_Module_Artist      1   jasmin:0
Horizon_Module_Artist   1   jasmin:0

```

Figure 15: Example client configuration file

---

### 3.2 Artist manager generator

usage: `artist_manager_generator <file.ccf>`

The artist manager generator, given a client configuration file, creates an artist manager (see section 2.3) specification and body in two files *artist\_manager\_spec.a* and *artist\_manager\_body.a* respectively. First the ccf file is parsed into a common data structure, then the artist descriptions portion of that data structure is passed to the *Create\_Artist\_Manager* procedure which generates the output files.

### 3.3 Client initializer generator

usage: `client_init_generator file.ccf`

Given a client configuration file, the client initializer generator generates a client initialization package (see section 2.8) into the file *client\_init.a*. First the ccf file is parsed into a common data structure, then the runtime configuration portion of that data structure is

passed to the *Create\_Client\_Init* procedure which generates the output file. The generated package, *Client\_Init*, contains code to initialize the client during its own elaboration and must be with'ed by the main tool. The generated initialization code will by default, bring up the configuration specified in the ccf file at the time the client initializer was generated. However, it will first try to locate the ccf file and configure the client dynamically based on it's current contents before resorting to the default configuration.

### 3.4 Client events generator

usage: `client_events_generator <file.ccf> <[unconstrained_types_file]>`

The client events generator, given a client configuration file, generates a client event type definition (see section 2.1). First the ccf file is parsed into a common data structure, then the ADT specification list within that data structure is passed to the *Create\_Client\_Events* procedure which generates the type definition into the file *client\_events.a*. All ADT specification files that are listed in the ccf must be present in the directory from which the client event generator is executed.

Because the fields of a client event record copy the type marks of the corresponding ADT operation parameters, unconstrained parameter types present a problem. This is due to the fact that Ada does not allow an unconstrained type as a record field type mark. A similar problem existed in the previous *Chiron-1* dispatcher and was handled hand-editing the generated code. The new client provides an automated solution to this problem, by allowing the user to specify an optional file that contains a list of all unconstrained type names. The client event generator will create a new access type for each unconstrained type and used the access type in place of the unconstrained type within the client event record definition. Note that if there are unconstrained types, the wrapper generator must be passed the unconstrained types file as well (see section 3.5).

The *Client\_Event\_Mode* array **must** be edited to indicate the Read/Write permissions of ADT operations before the client is compiled (see section 2.1.) Failure to do so could result in deadlock.

### 3.5 Wrapper generator

usage: `wrapper_generator <adt_specification_file> <[unconstrained_types_file]>`

Given an ADT specification file name, the wrapper generator generates a wrapper (see section 2.4) and an access controller (see section 2.5) for that ADT. Given a file that contains the ADT, `<adt_name>`, the wrapper generator produces three files: *wrapper\_<adt\_name>.spec.a*, *wrapper\_<adt\_name>.body.a*, and *<adt\_name>.controller.a*. The wrapper generator passes the ADT specification file name to the *Create\_Wrapper* procedure which generates the output files. The given ADT specification file must be present in the directory from which the wrapper generator is invoked.

Like the client event generator, the wrapper generator must also be able to handle unconstrained types. If the client event type has generated special access types in place of unconstrained types, the wrapper must be able to allocate and initialize these new access

types when creating event instances. This special code is automatically included if the unconstrained types file is given as a second argument to the wrapper generator. All wrappers in a client should be generated with the same unconstrained types file that was provided to the client events generator.

### 3.6 Dispatcher generator

usage: `dispatcher_generator <file.ccf>`

Given a client configuration file, the dispatcher generator generates the specified dispatching architecture (see section 2.6). The Client Dispatcher is always generated to the files, *client\_dispatcher\_spec.a* and *client\_dispatcher\_body.a*. Any dedicated ADT dispatchers are generated to files named, *dispatcher\_<adt\_name>.spec.a* and *dispatcher\_<adt\_name>.body.a*. The dispatcher generator parses the client configuration file into a common data structure, then passes the ADT specification list data to the *Create\_Dispatcher* procedure which generates the output files. The specified ADT files must be present in the directory from which the dispatcher generator is invoked.

### 3.7 Artist template generator

usage: `artist_template_generator <artist_name> [<adt_specification_file> ...]`

Given an artist name followed by an optional list of ADT specification filenames, the artist template generator generates an artist template (see section 2.7.1) specification and body into the files *<artist\_name>.spec.a* and *<artist\_name>.cal*. The artist name and ADT specification file list are passed to the *Create\_Artist\_Template* procedure which generates the output files. Any ADT specification files listed on the command line must be present in the directory from which the artist template generator is executed.

### 3.8 Client builder

usage: `client_builder [-ci] [-am] [-ce] [-w] [-d] [-at] \`  
`[-types <unconstrained_types_file>] <file.ccf>`

- ci : generate the client inializer
- am : generate the artist manager
- ce : generate the client events
- w : generate wrappers for each adt
- d : generate dispatchers
- at : generate artist templates
- default : generate all of the above
- types unconstrained\_types\_file :  
specify file containing information that the client events generator and the wrapper generator can use to handle unconstrained types correctly. Only valid with -ce, -w or default options.

The *client\_builder* tool is provided to help users generate all of a client's initial components. The *client\_builder* parses the ccf file into a common data structure, then uses the data to invoke the *create...* procedures for the specified components. The artist description list is used to generate an artist manager, the runtime configuration is used to generate a client initializer, and the ADT specification list is used to generate a client event type and dispatchers. A wrapper and access controller are generated for each ADT in the ADT specification list, and an artist template is generated for each artist in the artist description list. Several command line options are provided to allow the user to invoke any subset of client generator tools, but the default is to generate all components. The output for the *client\_builder*, given the ccf file in figure 15 and no command line options, is listed in appendix B.

### 3.9 LoCAL processor

usage: lcc <file.cal>

The LoCAL processor has been modified to work with the new artist template (see section 2.7.1). It also inserts code to handle the new event processing for both client and server events (see section 2.7.2). The ADL interfaces and the translation of ADL commands have remained unchanged, therefore the LoCAL language for applying ADL methods and setting object behaviors is the same.

### 3.10 Compiling and loading clients

The ada paths and environment variables necessary to compile and run a client have not changed from the previous *Chiron-1* design. However, the compilation order and load command have changed.

Once all necessary components have been generated, the client events file has been edited, and all artists have been processed by lcc, the order of compilation is as follows:

```
ADTs
client_events.a
artist_manager_spec.a
controllers
adt dispatchers
client_dispatcher
wrappers
artists
artist_manager_body.a
client_init.a
main tool
```

To load the Client:

```
a.ld client_init <tool's_main_procedure_name>
-o <executable_name> $Q_PATH/build.sun4-cc/libQ.a
```

*Adamakegen*<sup>2</sup> can be used to generate ada makefiles for artists. A couple of rules can be added to handle invoking lcc and loading the client. We suggest copying an existing makefile from the chiron installation applications and editing the main procedure and executable name and the path to the Makefile.config file. Then run adamakegen on your makefile to generate dependencies for your own client. If you are using a *Chiron* makefile as a template, you should invoke adamakegen on your makefile by typing:

```
make -f Makefile.<name> makedepend
```

This will ensure portability because it uses *adamakegen nicknames* for all installation-dependent paths.

### 3.11 Executing clients

*Chiron 1.4* clients are executed in the same manner as *Chiron 1.3* clients. However, the *Chiron 1.4* installation provides a new script, *chiron*, in its bin directory which can simplify the process. It takes the client binary as its argument. This script will execute both the client and the server, set any necessary environment variables, check for pid locks, and kill both processes on a <ctrl-c>.

---

<sup>2</sup>The adamakegen utility generates a Makefile based on the dependencies of a set of Ada source files. It can be obtained via anonymous ftp. Contact omalley@ics.uci.edu for more information.

## 4 Unresolved issues

Below are some issues that are not addressed in the new *Chiron-1* design but that should be considered in the *Chiron-2* design.

- Dynamic binding of ADT instances to artists.
  - Dispatching by instance
  - ADT locking by instance
  - Artist initialization at runtime
- Self referencing ADTs
  - Locking mechanism does not support this, can result in deadlock
  - Need some kind of transaction support for ADT access
- Artist composibility

## Acknowledgements

The author would like to acknowledge Mary Cameron, Greg Bolcer, Dennis Troup and Ruedi Keller for their work on the original *Chiron-1* client from which much of the new client is based. Craig MacFarlane contributed to the design and implementation of this work and Greg Bolcer, Ken Anderson and Craig Snider have given valuable feedback. Also, a special thanks to David Levine for providing us with the chiron execution script.

**References**

- [BCJ<sup>+</sup>89] Gregory Alan Bolcer, Mary Cameron, M. Gregory James, Rudolf K. Keller, Richard N. Taylor, and Dennis B. Troup. Chiron-1: Concept and design. Arcadia Technical Report UCI-89-12, University of California, Irvine, October 1989. (Revised, January 19, 1990).
- [CFM92] Mary Cameron, Kari Forester, and Craig MacFarlane. Chiron user manual. Arcadia Technical Report UCI-91-04, University of California, April 1992.

## A Dialogue artist example

```

with Client_Dispatcher_Stacks;
with Wrapper_Stacks;

package body Dialogue_Artist is

  task body Dialogue_Artist is

    --<< declare artist objects here. >>

    artist_frame      : ~CSL.ADL_base_frame;
    artist_panel      : ~CSL.ADL_panel;
    push_button       : ~CSL.ADL_button;
    pop_button        : ~CSL.ADL_button;
    top_field         : ~CSL.ADL_text fld;
    depth_field       : ~CSL.ADL_text fld;
    pop_up_frame      : ~CSL.ADL_popup_frame;
    pop_up_panel      : ~CSL.ADL_panel;
    pop_up_field      : ~CSL.ADL_num fld;
    behaviors         : CSL.Behavior_Array_Type := (others => System.No_Addr);
    value             : integer;
    artist_stack      : wrapper_stacks.stack;

    --<< 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_Select (object : CSL.Object_Type;
                             event : CSL.Chiron_Event_Ptr) is
      value : integer;
      result : integer;
    begin
      if object = push_button then
        --show dialogue box:
        ~Apply(ADL_popup_frame,
              pop_up_frame,
              set_show,
              true);
      elsif object = pop_button then
        --if stack not empty:
        if ( wrapper_stacks.depth (artist_stack) > 0 ) then

          --pop:
          value := wrapper_stacks.pop (artist_stack);
        end if;
      elsif object = pop_up_field then
        --if stack not full:
        if ( wrapper_stacks.depth (artist_stack) <
            wrapper_stacks.max (artist_stack) ) then
          --push:
          wrapper_stacks.push (artist_stack, event.num_val);
        end if;
      end if;
    end Handle_Select;
  end Dialogue_Artist;

```

```

procedure Handle_Stacks_Create (Event : Client_Event_Ptr) is
begin
  --update depiction
  artist_stack := Event.Stacks_Create_Result;
end Handle_Stacks_Create;

procedure Handle_Stacks_Push (Event : Client_Event_Ptr) is
  value : integer;
begin
  --update depiction
  artist_stack := Event.Stacks_Push_S;

  ~Apply(ADL_text fld,
         top_field,
         set_value,
         to_str(integer'image (Event.Stacks_Push_X)));

  value := wrapper_stacks.depth (artist_stack);
  ~Apply(ADL_text fld,
         depth_field,
         set_value,
         to_str(integer'image (value)));
end Handle_Stacks_Push;

procedure Handle_Stacks_Pop (Event : Client_Event_Ptr) is
  value : integer;
begin
  --update depiction
  artist_stack := Event.Stacks_Pop_S;
  value := wrapper_stacks.top (artist_stack);
  ~Apply(ADL_text fld,
         top_field,
         set_value,
         to_str(integer'image (value)));

  value := wrapper_stacks.depth (artist_stack);
  ~Apply(ADL_text fld,
         depth_field,
         set_value,
         to_str(integer'image (value)));
end Handle_Stacks_Pop;

begin --task body

  TEXT_IO.PUT_LINE("Begin dialogue artist.");

  accept Start_Artist (
    ID   : CSL.Artist_ID_Type;
    Aptr : SYSTEM.ADDRESS) do
    self_ptr := address_to_artist(Aptr);
    local_artist_id := id;

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

    Dispatcher_Stacks.Dispatcher.Register_Event
      (Local_Artist_ID, Client_Events.Stacks_Create,
       Handle_Stacks_Create'ADDRESS);
    Dispatcher_Stacks.Dispatcher.Register_Event
      (Local_Artist_ID, Client_Events.Stacks_Push,
       Handle_Stacks_Push'ADDRESS);
  end accept;

```

```

Dispatcher_Stacks.Dispatcher.Register_Event
(Local_Artist_ID, Client_Events.Stacks_Pop,
Handle_Stacks_Pop'ADDRESS);

end Start_Artist;
--<< create initial graphical objects here. >>

artist_frame := ~Apply(ADL_base_frame,
  create,
  frame_label => "Dialogue View",
  x           => 20,
  y           => 20,
  width      => 300,
  height     => 275,
  foreground => WHITE,
  background => FOREST_GREEN,
  show_footer => true);

artist_panel := ~Apply(ADL_panel,
  create,
  parent      => artist_frame,
  layout     => Layout_Horizontal,
  width      => 300,
  height     => 300,
  auto_placement => true,
  win_below  => false,
  below_sibling => null,
  win_right_of => false,
  right_of_sibling => null,
  x          => 0,
  y          => 0,
  x_gap     => 40);

push_button := ~Apply(ADL_button,
  create,
  parent      => artist_panel,
  button_id   => 1,
  label       => "Push",
  auto_placement => false,
  foreground  => WHITE,
  x          => 70,
  y          => 40);

pop_button := ~Apply(ADL_button,
  create,
  parent      => artist_panel,
  button_id   => 2,
  label       => "Pop",
  foreground  => WHITE,
  auto_placement => true);

top_field := ~Apply(ADL_text fld,
  create,
  parent      => artist_panel,
  label       => "Top Element: ",
  text_value  => "",
  layout      => layout_horizontal,
  display_length => 3,
  stored_length => 3,
  read_only   => true,
  mask_char   => ascii.nul,
  foreground  => WHITE,
  auto_placement => false,
  x          => 50,
  y          => 90);

```

```

depth_field := ~Apply(ADL_text fld,
    create,
    parent      => artist_panel,
    label       => "Stack Depth: ",
    text_value  => "",
    layout      => layout_horizontal,
    display_length => 3,
    stored_length => 3,
    read_only   => true,
    mask_char   => ascii.nul,
    auto_placement => false,
    x           => 50,
    y           => 130);

pop_up_frame := ~Apply(ADL_popup_frame,
    create,
    parent      => artist_frame,
    frame_label => "Push",
    x           => 275,
    y           => 50);

pop_up_panel := ~Apply(ADL_panel,
    create,
    parent => pop_up_frame,
    layout => Layout_Vertical);

pop_up_field := ~Apply(ADL_num fld,
    create,
    parent      => pop_up_panel,
    label       => "Enter Integer Value:",
    numeric_value => 0,
    layout      => Layout_Horizontal,
    min_value   => 0,
    max_value   => 100,
    display_length => 3,
    stored_length => 3,
    read_only   => false,
    foreground  => WHITE,
    auto_placement => true,
    x           => 0,
    y           => 0,
    foreground  => ADL_color_not_specified,
    show       => true);

~Apply(ADL_panel,
    pop_up_panel,
    ADL_window_fit);

~Apply(ADL_popup_frame,
    pop_up_frame,
    ADL_window_fit);

~Apply(ADL_panel,
    artist_panel,
    ADL_window_fit_height);

~Apply(ADL_base_frame,
    artist_frame,
    ADL_window_fit);

```

```
--<< set behaviors of graphical objects in response >>
--<< to server events here. >>
behaviors(Select_Event) := Handle_Select'ADDRESS;

~Set_Behavior(ADL_button,
              behaviors);
~Set_Behavior(ADL_num_fld,
              behaviors);

--<< call adl start_processing method here. >>
~Apply(ADL_base_frame,
        artist_frame,
        start_processing);

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

end Dialogue_Artist;
end Dialogue_Artist;
```

## B Execution of client\_builder for flight simulator example

<<Generating an Artist Manager>>

Files Created: artist\_mangager\_spec.a  
artist\_manager\_body.a

<<Generating a Client Initializer>>

Files Created: client\_init.a

<<Generating Client Events>>

Parsing file: aileron\_spec.a ...

# 0 error(s) found.

Parsing file: altitude\_spec.a ...

# 0 error(s) found.

Parsing file: attitude\_spec.a ...

# 0 error(s) found.

Parsing file: psi\_spec.a ...

# 0 error(s) found.

Parsing file: speed\_spec.a ...

# 0 error(s) found.

Parsing file: tail\_spec.a ...

# 0 error(s) found.

Parsing file: theta\_spec.a ...

# 0 error(s) found.

Parsing file: throttle\_spec.a ...

# 0 error(s) found.

File Created: client\_events.a

<<Generating Wrappers>>

Parsing file: aileron\_spec.a ...

# 0 error(s) found.

Files Created: wrapper\_aileron\_module\_spec.a  
wrapper\_aileron\_module\_body.a  
aileron\_module\_controller.a

Parsing file: altitude\_spec.a ...

# 0 error(s) found.

Files Created: wrapper\_altitude\_module\_spec.a  
wrapper\_altitude\_module\_body.a  
altitude\_module\_controller.a

Parsing file: attitude\_spec.a ...

# 0 error(s) found.

Files Created: wrapper\_attitude\_module\_spec.a  
wrapper\_attitude\_module\_body.a  
attitude\_module\_controller.a

Parsing file: psi\_spec.a ...

# 0 error(s) found.

Files Created: wrapper\_psi\_module\_spec.a  
wrapper\_psi\_module\_body.a  
psi\_module\_controller.a

Parsing file: speed\_spec.a ...  
# 0 error(s) found.

Files Created: wrapper\_speed\_module\_spec.a  
wrapper\_speed\_module\_body.a  
speed\_module\_controller.a

Parsing file: tail\_spec.a ...  
# 0 error(s) found.

Files Created: wrapper\_tail\_module\_spec.a  
wrapper\_tail\_module\_body.a  
tail\_module\_controller.a

Parsing file: theta\_spec.a ...  
# 0 error(s) found.

Files Created: wrapper\_theta\_module\_spec.a  
wrapper\_theta\_module\_body.a  
theta\_module\_controller.a

Parsing file: throttle\_spec.a ...  
# 0 error(s) found.

Files Created: wrapper\_throttle\_module\_spec.a  
wrapper\_throttle\_module\_body.a  
throttle\_module\_controller.a

<<Generating Dispatchers>>

Files Created: dispatcher\_attitude\_module\_spec.a  
dispatcher\_attitude\_module\_body.a  
dispatcher\_psi\_module\_spec.a  
dispatcher\_psi\_module\_body.a  
dispatcher\_tail\_module\_spec.a  
dispatcher\_tail\_module\_body.a  
dispatcher\_theta\_module\_spec.a  
dispatcher\_theta\_module\_body.a  
client\_dispatcher\_spec.a  
client\_dispatcher\_body.a

<<Generating Artist Templates>>

File Created: aileron\_module\_artist.cal  
File Created: tail\_module\_artist.cal  
File Created: throttle\_module\_artist.cal  
File Created: pitch\_artist.cal  
File Created: roll\_artist.cal  
File Created: altitude\_module\_artist.cal  
File Created: altimeter\_module\_artist.cal  
File Created: airspeed\_module\_artist.cal  
File Created: compass\_module\_artist.cal  
File Created: turn\_module\_artist.cal  
File Created: horizon\_module\_artist.cal