User's Guide

August 1991
CMU/SEI-91-UG-7

Serpent: Ada Application Developer's Guide

User Interface Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Charles J. Ryan, Major, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

List of Examples

1 Introduction

Serpent is a user interface management system (UIMS) that supports the development and execution of a user interface of a software system. Serpent supports incremental development of the user interface from the prototyping phase through production to maintenance or sustaining engineering. Serpent encourages a separation of functionality between the user interface and functional portions of a software system. Serpent is also easily extended to support additional user interface toolkits.

1.1 This Manual

This manual describes how to develop applications using Serpent. Readers are assumed to have read and understood the concepts described in the *Serpent Overview*, as well as to have had experience using the Ada programming language.

1.1.1 Organization

The contents of this guide include:

- **Introduction** and **Overview**. This chapter provides a general description of the role of an application in a software system developed with Serpent. It also describes a conceptual framework for application development.
- **Specifying the Contract.** This chapter describes the tasks necessary to define the type, structure and values of data to be shared between an application program and Serpent and to establish runtime communications with Serpent.
- **Modifying Information**. This chapter describes the tasks necessary to add, modify or remove information to/from the Serpent shared database.
- **Retrieving Information**. This chapter describes the tasks necessary to define and retrieve changes to information from the Serpent shared database.
- **Finishing the Application.** This chapter describes the finishing touches that should be applied to the application, including error checking and exception handling.
- **Testing and Debugging**. This chapter describes utilities available to assist in the testing and debugging of the application.
- **Appendix A: Data Structures.** This appendix is a complete reference of all the constants, types, routines, and other data structures available to Serpent application developers using the Ada programming language.

- **Appendix B: Routines.** This appendix is a complete reference of all the routines available to Serpent application developers using the Ada programming language.
- **Appendix C**: **Commands for Testing Serpent Applications and Dialogues**. This appendix is a reference of commands available to Serpent application developers from the operating system.
- **Appendix D: Spider Example**. This appendix is a complete application example, developed in the Ada programming language.

1.1.2 Typographical Conventions

Code examples	`Courier typeface`
Code directly related to text	**`Bold, courier typeface`**
Variables, attributes, etc.	`Courier typeface`
Syntax	`Courier typeface`
Warnings and cautions	***Bold, italics***

1.2 Other Serpent Documents

The purpose of this guide is to provide the information necessary to develop Serpent applications. The following publications address other aspects of Serpent.

Serpent Overview
Introduces the Serpent system.

Serpent: System Guide
Describes installation procedures, specific input/output file descriptions for intermediate sites and other information necessary to set up a Serpent application.

Serpent: Saddle User's Guide
Describes the language that is used to specify interfaces between an application and Serpent.

Serpent: Dialogue Editor User's Guide
Describes how to use the editor to develop and maintain a dialogue.

Serpent: Slang Reference Manual
Provides a complete reference to Slang, the language used to specify a dialogue.

Serpent: C Application Developer's Guide
Describes how the application interacts with Serpent. This guide describes the runtime interface library, which includes routines that manage such functions as timing, notification of actions, and identification of specific instances of the data.

Serpent: Guide to Adding Toolkits
Describes how to add user interface toolkits, such as various Xt-based widget sets, to Serpent or to an existing Serpent application. Currently, Serpent includes bindings to the Athena Widget Set and the Motif Widget Set.

The following figure shows Serpent documentation in relation to the Serpent system:

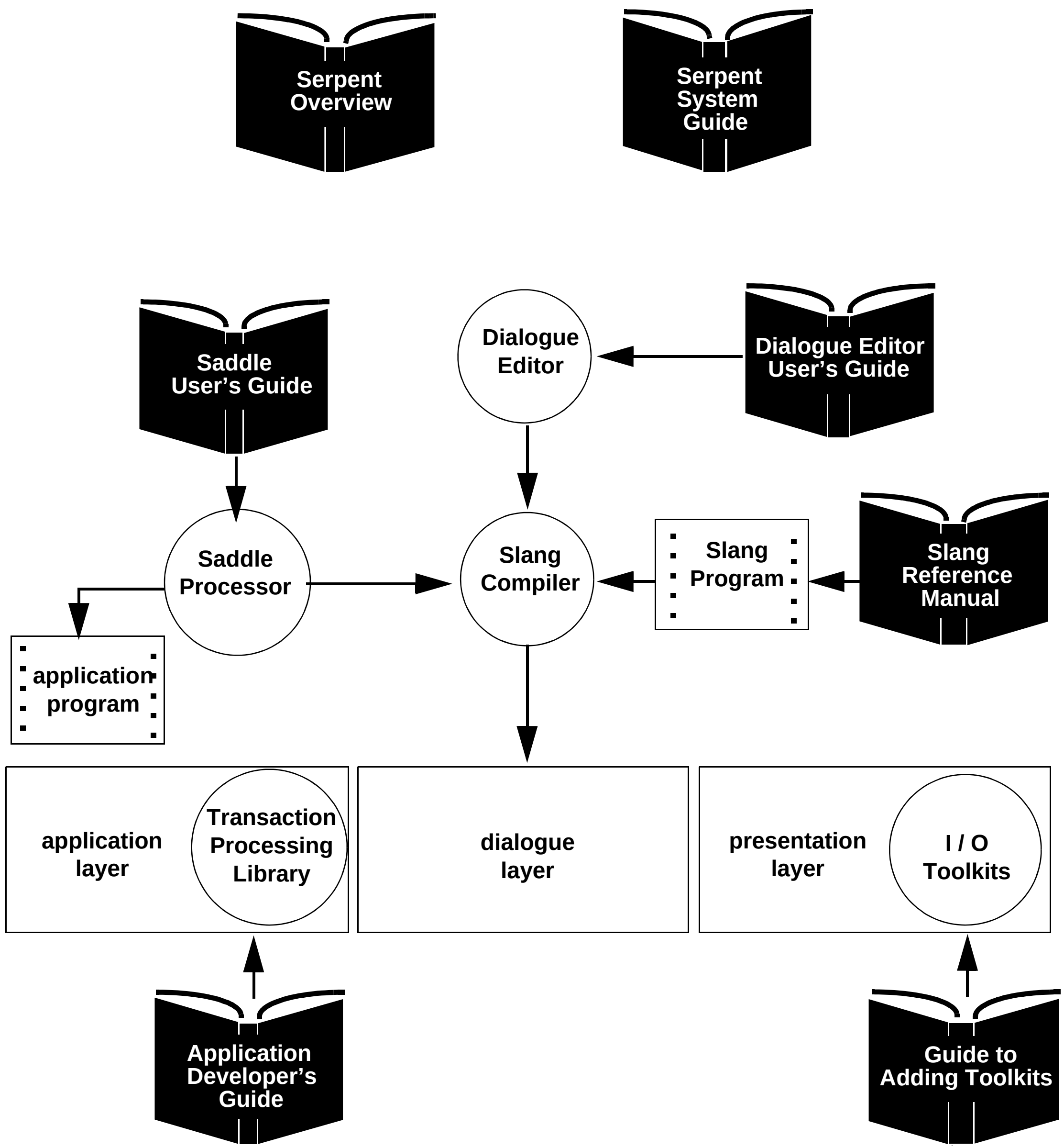

Figure 1-1 Serpent Documents

2 Overview

A main goal of Serpent is to encourage the separation of a software system into an application portion and a user interface portion to provide the application developer with a presentation-independent interface. The application portion consists of those components of a software system that implement the “core” application functionality of a system. The user interface portion consists of those components that implement an end-user dialogue. A dialogue is a specification of the presentation of application information and end-user interactions.

During the design stage, the system designer decides which functions belong in the application component and which belong in the user interface component of the system.

2.1 Serpent Architecture

Serpent is implemented using a standard UIMS architecture. This architecture (see Figure 2-1) consists of three major layers: the *presentation layer*, the *dialogue layer*, and the *application layer*. The three different layers of the standard architecture are viewed as providing differing levels of end-user feedback.

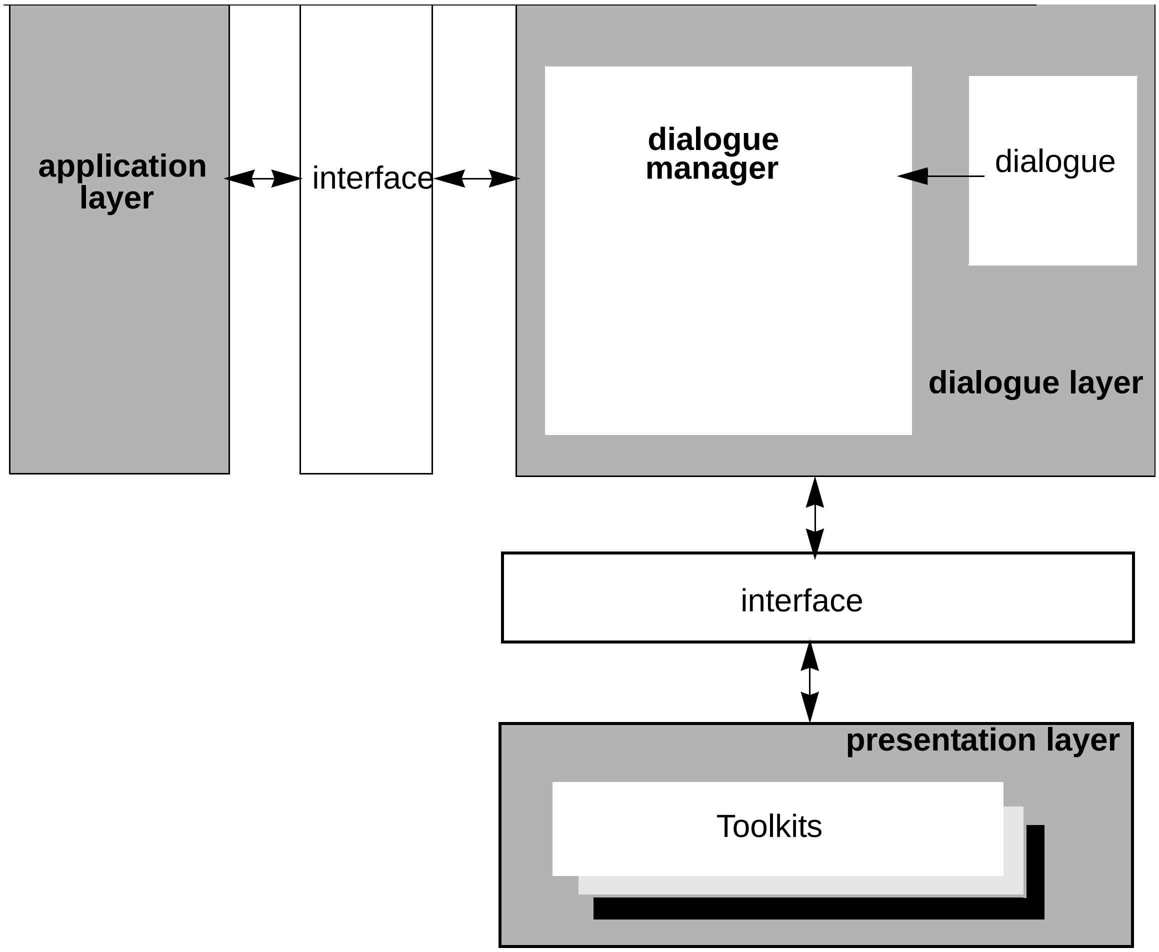

Figure 2-1 Serpent Architecture

The presentation layer consists of various input/output toolkits that have been incorporated into Serpent. Input/output toolkits are existing hardware/software systems that perform some level of generalized interaction with the end user. Serpent is being distributed with an interface to the X Window System, Version 11. Other input/output toolkits can be integrated with Serpent. See *Serpent: Guide to Adding Toolkits* for a discussion of how this can be accomplished.

One way of viewing the three levels of the architecture is the level of functionality provided for user input. The presentation layer is responsible for lexical functionality, the dialogue layer for syntactic functionality, and the application layer for semantic functionality. In terms of a menu example, the presentation layer has responsibility for determining which menu item was selected and for presenting feedback that indicates which choice is currently selected. The dialogue layer has responsibility for deciding whether another menu is presented and presenting it, or whether the choice requires application action. The application layer is responsible for implementing the command implied by the menu selection.

The end user interface for a software system is specified formally as a *dialogue*. The dialogue is executed by the dialogue manager at runtime in order to provide an end user interface for a software system. The dialogue specifies both the presentation of application information and end user interactions. The Serpent dialogue specification language (Slang) allows dialogues to be arbitrarily complex.

The application provides the functional portion of the software system in a presentation-independent manner. It may be developed in C, Ada, or other programming languages. The application may be either a functional simulation for prototyping purposes or the actual application in a delivered system. The actions of the application layer are based upon knowledge of the specific problem domain.

2.2 Shared Database

Serpent provides an active database model for specifying the user interface portion of a system. In an active database, multiple processes are allowed to update a database. Changes to the database are then propagated to each user of the database. This active database model is implemented in Serpent by a *shared database* that logically exists between the application and I/O toolkits. The application can add, modify, query, or remove data from the shared database. Information provided to Serpent by the application is available for presentation to the end user. The application has no knowledge of the presentation media or user interface styles used to present this information.

Information in the shared database may be updated by either the application or I/O toolkits. Figure 2-2 illustrates the use of the shared database in Serpent.

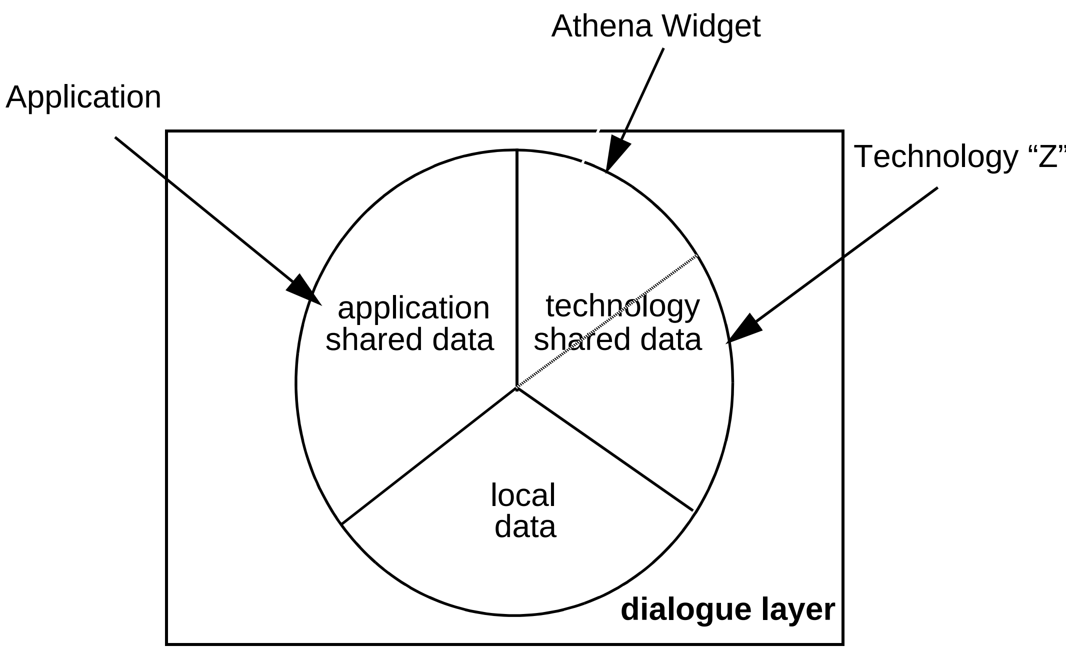

Figure 2-2 Shared Database

Serpent allows the specification of dependencies between elements in the shared database in the dialogue. These dependencies define a mapping among application data, presentation objects, and end user input. The dialogue manager enforces these dependencies by operating on the information stored in the shared database until the dependencies are met. Changes are then propagated to either the application or the I/O toolkits as appropriate. See the *Serpent: Slang Reference Manual* (CMU/SEI-91-UG-5) for a further discussion.

The *type* and *structure* of information that can be maintained in the shared database is defined externally in a *shared data definition file*. This corresponds to the database concept of *schemas*. A shared data definition file is required for each application.

A shared data definition file consists of both aggregate and scalar data structures. Top-level data structures become *shared data* elements that may be instantiated at runtime. Nested data structures become components that are considered part of the shared data element. Serpent does not allow nesting of records.

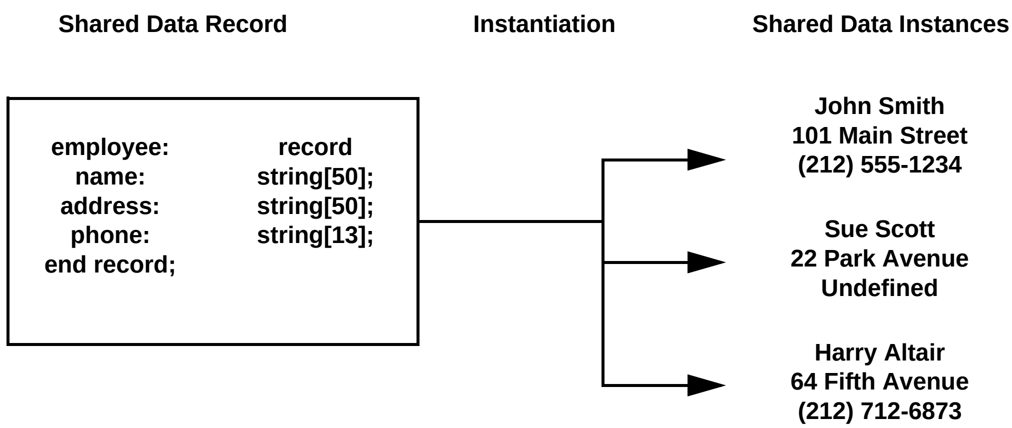

Figure 2-3 Shared Data Instantiation

It is possible to define multiple instances of a single shared data element. Shared data elements are instantiated by specifying the element name. Each *shared data instance* is identified by a unique *ID*. IDs must be maintained by the application to identify shared data instances when multiple instances of a single shared data element exist. Figure 2-3 provides an illustration of shared data instantiation.

Since the dialogue manager, the application, and any toolkits participating in a particular execution of Serpent are separate system processes that use the shared database, they can potentially modify the database concurrently, possibly compromising the integrity of the database. This problem is solved in Serpent through the use of database concurrency control techniques. Updates to the Serpent shared database are packaged in transactions. Transactions are collections of updates to the shared database that are logically processed at one time. Transactions can be *started*, *committed*, or *aborted*. A transaction which has been started but neither committed nor aborted yet is said to be *open*. Multiple transactions may be open at the same time. Committing a transaction causes the updates to be made to the shared database. Aborting a transaction causes termination of the transaction without any update of the shared database.

Communicating with Serpent

The application communicates with Serpent using the shared database model described earlier in this document. Information added to shared data is available to be presented to the end user by the dialogue. Changes to application data are automatically communicated back to the application.

2.3 Application Development

The application, or non-user interface portion of the system, provides the “core” functionality of a software system developed using Serpent. The application can be written in Ada, C, or other programming languages and can be either a simulation or an actual application.

An application may only add information to shared data or it may only retrieve information from shared data. For example, an application that monitors and displays the status of a computer network may only need to add information to shared data to update the display. An application such as an automatic teller machine (ATM) might only need to retrieve data from the user interface.

All transactions to and from the application are handled explicitly in the application using the routines and data structures available in the Serpent application interface. This document describes the usage and definitions of these routines and data structures.

Error Checking and Recovery

Each routine in Serpent sets status on exiting. It is the responsibility of the application developer to check this status to perform appropriate error recovery. Serpent provides routines to both check and print the status.

Testing and Debugging

Serpent provides a record/playback feature that can be used in testing and debugging. Transactions between the application and dialogue manager or between the dialogue manager and the various toolkits can be recorded, then played back at a later time. This is useful in isolating problems or in performing regression/stress testing of an application, dialogue, or toolkit.

Spider Example

The spider application is an example of an application system developed using Serpent. Figure 2-4 is an illustration of a "spider chart" display that is one possible end-user interface for the application.

Adapted from a command and control application, the spider application monitors and displays the status of various sensor sites and their associated communication lines to the two correlation centers (Figure 2-4).

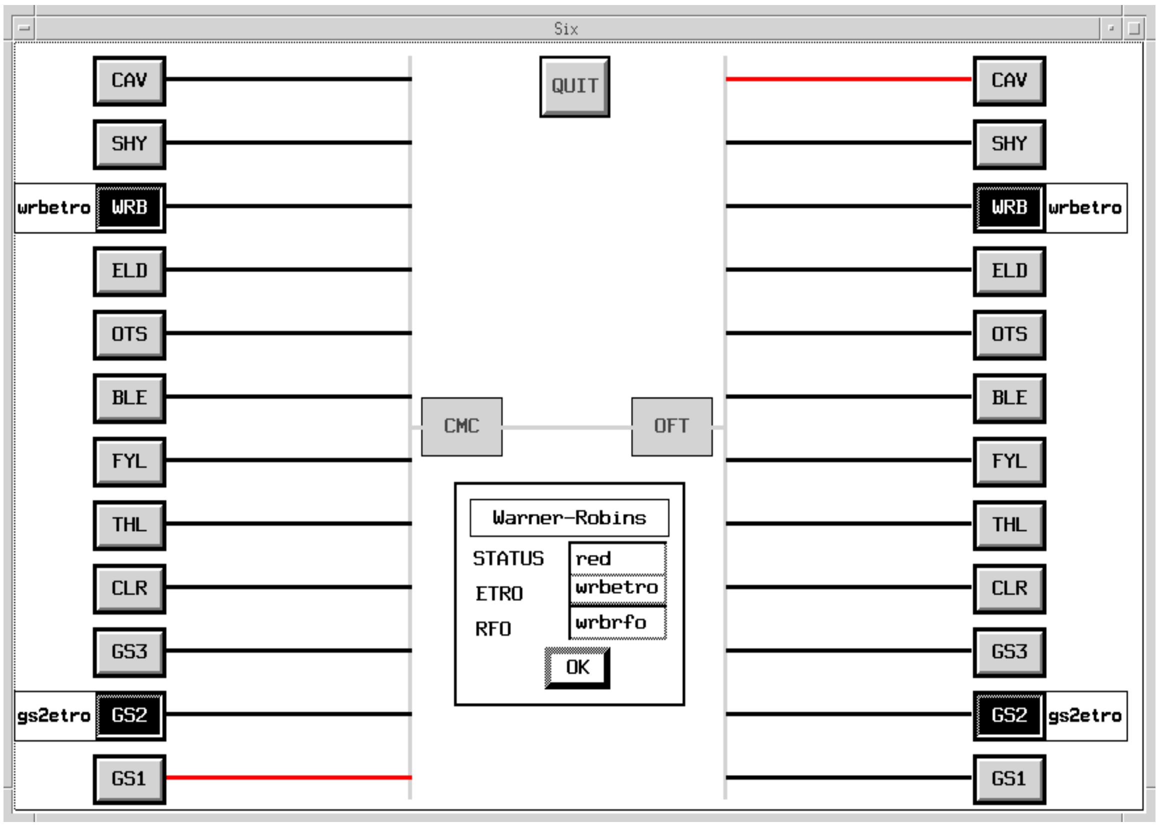

Figure 2-4 Spider Chart Display

The columns of rectangular boxes on the right and left sides of the spider chart display (for example, GS1, GS2) represent sensor sites. The rectangles in the middle of the display represent the correlation centers that collect information from the sensors. Each sensor site communicates with both correlation centers; this is represented by the duplication of sensor site boxes on both the right and left sides of the display. The lines represent communication lines between the sensor sites and the correlation centers. The status of sensors is represented by the shading of the rectangles. On a color display, the status would be represented using different background colors.

An operator may display detailed information concerning a sensor site by selecting a sensor site box corresponding to that sensor. This causes a detailed window to appear, displaying the status of the sensor, the date and time of the last message, the reason for outage (RFO) and the estimated time to returned operation (ETRO). These fields may be modified by the operator. Sensors may be in one of three states: operational, impaired, or down. For sensors that are not fully operational (i.e., the status is yellow) the ETRO is displayed to the outside of the sensor site box. ETROs are also displayed over communication lines that are not fully operational. The operator may also dynamically reconfigure the network[1] by adding/deleting sensors to/from the network.

[1]The capability of dynamically reconfiguring the network does not exist in the spider chart example distributed with Serpent Version 1.0.

3 Specifying the Contract

The first step in creating a software system using Serpent is to apportion system functionality between the dialogue and the application. This involves creating a contract between the two components: defining the type and structure of information to be communicated, or shared, between the two components; establishing the range of values of this data; and establishing runtime communication between the components.

3.1 Defining Shared Data

Shared data is information that is communicated or shared between the application and dialogue. Defining shared data involves two steps:

1. Create the shared data definition file.
2. Run the created file through the Saddle processor.

The following is a brief description of each of these two steps. The *Serpent: Saddle User's Guide* contains a more complete description of both these steps.

Step 1: Create the shared data definition file. The shared data definition file defines the type and structure of information that can be shared between the application and dialogue. The shared data definition is specified in Saddle. By convention, the file is given the name of the application, followed by the extension `.sdd`.

Example 3-1 is an example of a shared data definition file for the spider application. The content of the shared data definition file is independent of the implementation language used. Note that these shared data record templates contain only information to define the application objects; they do not specify how the information is presented to the end user.

```
<< spiderA >>

spider: shared data

 sensor_sdd: record
    site_abbr: string[3];
    status: integer;
    site: string[32];
    last_message: string[8];
    rfo: buffer[32];
    etro: string[8];
 end record;

 cc_sdd: record
    name: string[3];
    status: integer;
 end record;
```

```
 communication line_sdd: record
    from_sensor: id of sensor_sdd;
    to_cc: id of cc_sdd;
      etro: string[8];
      status: integer;
 end record;

end shared data;
```

Example 3-1   Spider Shared Data Definition File

The file shown in Example 3-1 contains definitions for the data shared between the application and the dialogue for the spider application. The first line of the file contains the name (and possible path information) of the executable image of the application. This application is automatically executed by the Serpent command at runtime. (*Serpent: System Guide* contains a complete explanation of this process.) The three shared data record templates define the type and structure of the sensor, correlation center, and communication line application objects.

Step 2: Run the created file through the Saddle processor. Once the shared data has been defined in the file, it can be processed by Saddle to generate an Ada Package. This package will have the same name as the shared data definition file with a different extension. For example, the shared data file `spiderA.sdd` will generate the file `spiderA.ada`. This package can then be `with`ed in the Ada application and used to declare local variables of the shared data types. The Ada package generated by running the shared data definition file shown in Example 3-1 through the Saddle processor is illustrated in Example 3-2.

```
MAIL_BOX: constant string := “SPIDERA_BOX”;
ILL_FILE: constant string := “spiderA.ill”;

type sensor_sdd is record
       self:   id_type;        -- (no element pointer)
        site_abbr:      string (1..4);
        status: integer;
        site:   string (1..51);
        last_message:   string (1..9);
        rfo:    string (1..51);
        etro:   string (1..9);
end record;

type cc_sdd is record
     name: string(1..4);
     status: integer;
end record;

type communication_line_sdd is record
       from_sensor:   id_type;       -- (no element
```

```
pointer)
                to_cc:  integer;
                etro:   string (1..9);
                status: integer;
        end record;

        type communication_line_sdd_ptr is access
communication_line_sdd;

end spiderA;
```

Example 3-2 Ada Language Package

In Example 3-2, the first two lines in the file define two well-known constants: `MAIL_BOX` and `ILL_FILE`. These constants will be used in initializing Serpent. The three structures correspond to the record templates defined within the shared data definition file.

3.2 Data Types and Values

One output of processing the shared data definition file through the Saddle processor is an Ada package containing corresponding Ada structures for the shared data records. These Ada structures can be used to declare local variables that correspond in size and structure to shared data records. Components of shared data records can be declared as any of the following types: boolean, integer, real, string, ID or buffer. The Ada records generated from these declarations depend on the type of the components. Example 3-3 is unrelated to the spider example used throughout this guide but includes a description of a shared data record that contains an example of each type of component.

```
employee_sdd: record
 name: string[32];
 salary: integer;
 exempt: boolean;
 experience: real;
 job_desc: buffer;
 self: id of employee_sdd;
end record;
```

Example 3-3 Shared Data Definition

Example 3-4 shows the Ada package that is generated when the `employee_sdd` record is processed by Saddle processor.

```
type employee_sdd is record
   name: string (1..33);
   salary: integer;
   exempt: boolean;
   experience: float;
   job_desc: buffer;
   self: id_type;
end record;
```

Example 3-4 Generated Ada Package

Although each shared data component is now represented using an Ada language specific type, there is still a Serpent data type associated with each of them. The Serpent data type can be determined at runtime using the `get_shared_data_type` function illustrated in Example 3-5. The `serpent_data_type` is an enumeration of the different Serpent data types and is defined in Appendix A.

```
serpent_data_type type;
--
-- Get the Serpent type of the employee record salary
-- component.
--
 type := S.Get_Shared_Data_Type("employee", "salary");
```

Example 3-5 Serpent Data Type

Shared data values specified as strings in the shared data definition file are represented by strings in the Ada package generated by the Saddle processor. It is therefore not necessary to allocate memory for these strings, although it is necessary to convert the strings to null terminated strings.

```
--
-- Declare a local shared data variable.
--
 employee: employee_sdd;
--
-- null terminate string.
--
 employee.name := "Harry Alter" & ASCII.NUL;
```

Example 3-6 Assigning Values to String Components

Shared data components of type integer, boolean, real, or ID can be assigned directly to Ada language variables. IDs are returned from a number of Serpent routines and are `id_type`. Saddle integers and booleans correspond to the equivalent Ada types and Saddle reals are actually of Ada type float. (See Example 3-7.)

```
--
-- Integer, boolean, or real components can be set
-- directly.
--
 employee.salary := 45000;
 employee.exempt := FALSE;
 employee.experience := 3.2;
```

Example 3-7 Assigning Values to Integer, Boolean, or Real Components

Buffer is the only dynamic shared data type in that neither the size nor the type of the information is predefined. Example 3-8 describes the buffer structure. Buffer type is required and specifies the type of information stored in the buffer. Buffer length is the size in bytes of the data and is required even if the data is of a well known type (i.e., integer). Buffer body is a pointer to the actual data. The space used to maintain this data is not part of the buffer structure and must be managed by the user.

```
type buffer is record
     type: shared_data_types
     length: integer;
     body: system.address;
```

Example 3-8 Buffer Structure

Buffers can be used to:

- Share untyped, contiguous data.
- Share large amounts of contiguous data (i.e., large strings).
- Provide variant records.

Example 3-9 contains the example of the `employee.job_desc` buffer being used as a string.

```
--
-- This buffer is being used as a string.
--
 employee.job_desc.type := sd_string;
 string_variable := "Look busy";
 employee.job_desc.length := string_variable'length;
 employee.job_desc.body := string_variable'address;
```

Example 3-9 Assigning Values to Buffer Components

Shared data values can also be undefined. All uninitialized components of a shared data record instance created using the `add_shared_data` function are initialized by Serpent to be undefined. On the other hand, components of a local, shared data variable have whatever values are left by the system—most likely zeros. If this structure is used to initialize the shared data instance (with the `add_shared_data` or `put_shared_data` routines), all the components of the instance are initialized with these values. Components of local, shared data variables can be explicitly set to undefined using the `set_undefined` routine illustrated in Example 3-10. The `is_undefined` function can be used to determine if a component value is undefined.

```
--
-- The set_undefined function is used to set the value of
-- a component to undefined.
*/
 set_undefined(sd_buffer, employee.job_desc'address);
```

Example 3-10   Setting Component Values to Undefined

3.3 Initialization and Cleanup

The first task of any Serpent application is to initialize the system. Serpent initialization establishes communication between the application and the dialogue. The final application task is to clean up the Serpent system environment before exiting. The code segment from the spider application shown in Example 3-11 illustrates the basic operations necessary for Serpent initialization and cleanup.

```
with Serpent;
with S_Types; use S_types;

begin
 Serpent.Serpent_Init(MAIL_BOX,ILL_FILE);
 Serpent.Serpent_Cleanup;
end
```

Example 3-11   Serpent Initialization

Specification Steps:

1. ***Include Serpent package***. The `Serpent` and `S_Types` packages contain the external definition for the Serpent interface.
2. ***Initialize Serpent.*** The `serpent_init` procedure is used to initialize Serpent. It takes as parameters the `MAIL_BOX` and `ILL_FILE` constants generated by the Saddle processor. This procedure establishes communication between the application and the dialogue manager.

3. ***Clean up.*** The `serpent_cleanup` routine must be invoked before exiting the application. It is important to complete this step to release allocated system resources.

4 Modifying Information

The application can add, change, or remove information to and from the shared database using the transaction mechanism described in the introductory chapter of this document. Together, these are considered modifications to the shared database. The collection of application data in the shared database is known as the *view*. This is the information that is available to the dialogue writer to be presented to the end user. The view can be modified by either the application or the dialogue.

4.1 Sending Transactions

Before information can be modified in the shared database, it is necessary to start a transaction. All modifications to the shared database must be performed as part of a transaction.

It is possible to have multiple transactions open at one time. Each transaction has a unique transaction handle. Every operation performed on or to a transaction must specify this transaction handle.

The actual change to the shared database does not occur until the transaction is committed. Up to this point it is also possible to roll back the transaction so that none of the changes to shared data occur.

The code segment from the spider application in Example 4-1 shows the operations necessary for sending transactions. Code and comments directly related to the task are emphasized in bold type.

```
begin
 transaction : S_Types.Transaction_Type; --transaction handle

 Serpent.Serpent_Init(MAIL_BOX,ILL_FILE);
 transaction := Serpent.Start_Transaction;
 Serpent.Commit_Transaction(transaction);
 Serpent.Serpent_Cleanup;
end
```

Example 4-1 Sending Transactions

Specification Steps:

1. Declare ***transaction variable***. A local variable of `transaction_type` can be used to maintain a transaction handle.

2. ***Start a transaction.*** The `start_transaction` function returns a transaction handle that must be passed to any subsequent commands operating on the transaction.
3. ***Commit the transaction.*** The actual change to shared data does not occur until the transaction is committed. Up to this point it is also possible to roll back the transaction using the `rollback_transaction` routine so that none of the changes to shared data occur.

4.2 Adding Static Information

This section makes some simplifying assumptions about the application that may in fact hold true for simple programs. The primary assumption is that the application will create only a fixed number of shared data instances so that the IDs of these instances can be maintained in local variables. A secondary assumption is that the application will create no more than one instance of each shared data element.

At any given moment, there can be up to three different versions of any given shared data instance. First, there is a local copy in the application. Second, there can be a copy that is part of an open transaction. Third, there is a copy in the shared database. Depending upon whether the shared data instance has been last modified by the application or by the end-user, the more current copy could be either the local application or shared database copy. A shared data instance that is part of an open transaction is the delta from the more current to less current copy of the shared data instance. The shared data copy being affected by any given operation should be apparent from the context.

Variables of generated shared data types are referred to as shared data variables. The first step in adding information to shared data is to assign values to these shared data variables. The method for doing this is based on the Serpent types of the components and is explained in detail in Section 3.2. These variables can then be used to initialize a record instance, either a component at a time or the entire record at once.

Once a transaction has been started, you can begin to add, change or remove information to/from the shared database as part of this transaction. These changes are made as part of the transaction and are not applied to the shared database until the transaction is committed.

The code segment from the spider application in Example 4-2 illustrates the operations involved in adding information to the shared database. Code and comments directly related to the task are emphasized in bold type.

```
with Serpent;          -- serpent interface definition
with S_Types; use S_Types;
with SpiderA           -- application data structures
GREEN_STATUS: constant := 0;
YELLOW_STATUS: constant := 1;
RED_STATUS: constant := 2;

begin
 transaction :S_Types.Transaction_Type; -- transaction handle
 cmc: SpiderA.cc_sdd            -- shared data variables
 gs1: SpiderA.sensor_sdd        -- shared data variables
 cmc_id,gs1_id: id_type         -- object instances

 Serpent.Serpent_Init(MAIL_BOX,ILL_FILE);
--
-- Initialize shared data variables.
--
 cmc.name := "CMC" & ASCII.NUL;
 cmc.status := GREEN_STATUS;
 gs1.status := RED_STATUS;
--
-- Start a transaction to be sent to the dialogue.
--
 transaction := Serpent.Start_Transaction;
--
-- Create an instance of the correlation center shared data
-- record in the transaction and initialize using the shared
-- data variable.
--
 cmc_id := Serpent.Serpent.Add_Shared_Data(
    transaction,"correlation_center", "", cmc'address
 );
--
-- Create an instance of the sensor shared data record but
-- this time update only the name component.
--
 gs1_id := Serpent.Add_Shared_Data(
   transaction,"sensor", "name", gs1.name'address
 );

 Serpent.Commit_Transaction(transaction);

 Serpent.Serpent_Cleanup;
end;
```

Example 4-2 Adding Information to the Shared Database

Specification Steps:

1. ***With Saddle generated header file***. This file (spiderA.h in the example) defines the structure of the shared data. The packages Serpent and S_Types must be specified before spiderA.h because SpiderA uses types defined in S_Types..

2. ***Define constants***. The spider example contains three constants: `GREEN_STATUS`, `YELLOW_STATUS`, and `RED_STATUS`. These constants are not required but help increase the clarity of the example.

3. ***Define shared data variables.*** Variables `cmc` and `gs1` are both instances of generated shared data structures. These variables are used to initialize instances of shared data in the shared database.

   The variables `cmc_id` and `gs1_id` are used to store the ids of the created shared data instances. These variables are declared to be of `id_type`. The ids are necessary to perform further operations on these instances in the shared database.

4. ***Assign values to shared data variables.*** The mechanism for accomplishing this task depends on the component types. This is explained in detail in Section 3.2.

5. ***Add information to the shared database***. The `add_shared_data` routine creates a shared data instance as part of the specified transaction and returns the ID of the instance. The routine allows you to initialize a single component of the instance by specifying the name of the component and providing a *pointer* to the initial value. Any uninitialized fields of the instance are left undefined. It is also possible to initialize the entire instance by providing a pointer to the structure and specifying " " for the component name.

4.3 Modifying Information

Shared data instances in transactions or in the shared database can be modified using the `put_shared_data` procedure. This procedure takes as a parameter the ID of the shared data instance.

It is possible to modify any single component of a shared data record instance, or the entire record. Unmodified components in the transaction are marked as unchanged and maintain their current values. This is different from components that are explicitly set to undefined, which is actually a value.

The code segment from the spider application in Example 4-3 illustrates the operations involved in adding dynamic information to the shared database. Code and comments directly related to the task are emphasized in bold type.

```
with Serpent;          -- serpent interface definition
with S_Types; use S_Types;

begin
 transaction :S_Types.Transaction_Type; transaction handle
 gs1: SpiderA.sensor_sdd        -- shared data variables
```

```
 cmc_id,gs1_id: id_type         -- object instances

 Serpent.Serpent_Init(MAIL_BOX,ILL_FILE);
 transaction := Serpent.Start_Transaction;
--
-- Update the name component of the sensor using a
-- string constant.
--
 Serpent.Put_Shared_Data(
    transaction, gs1_id, "sensor", "status", "GS1"'address
 );
 Serpent.Commit_Transaction(transaction);
 Serpent.Serpent_Cleanup;
end;
```

Example 4-3 Modifying Information in the Shared Database

Specification Task

Modifying information in the shared database. The `put_shared_data` routine modifies the values of shared data instances that have already been created and are part of a transaction. This routine works in an identical manner to the `add_shared_data` call except that it takes an extra parameter, the ID of the shared data instance to be modified. The `put_shared_data` routine in Example 4-4 is used to assign a value (a string) to the name component of the first shared data instance.

4.4 Removing Information

Shared data instances in transactions or in the shared database can be removed using the `remove_shared_data` procedure. It is not possible to remove components of shared data record instances.

The code segment from the spider application in Example 4-4 illustrates the operations involved in removing information from the shared database. Code and comments directly related to the task are emphasized in bold type.

```
with Serpent;          -- serpent interface definition
with S_Types; use S_Types;

begin
 transaction :S_Types.Transaction_Type; transaction handle
 gs1: SpiderA.sensor_sdd        -- shared data variables
 cmc_id,gs1_id: id_type         -- object instances

  Serpent.Serpent_Init(MAIL_BOX,ILL_FILE);
  transaction := Serpent.Start_Transaction;
--
-- Update the name component of the sensor using a
-- string constant.
--
  Serpent.Remove_Shared_Data(transaction, "sensor_sdd",
```

```
gs1_id);
 Serpent.Commit_Transaction(transaction);
 Serpent.Serpent_Cleanup;
end;
```

Example 4-4 Removing Information from the Shared Database

Specification Task

Removing information from the shared database. The `remove_shared_data` procedure is used to remove a shared data instance from either the transaction or the shared database. The procedure takes a transaction handle, the element name, and the ID of the shared data instance to be deleted as parameters.

5 Retrieving Information

Serpent implements an *active database model* from the perspective of the application interface. This means that changes to application data resulting from end-user interactions with the system are automatically communicated back to the application, using the same transaction mechanism described in Section 4.3.

Transactions from the dialogue to the application consist of a list of changed shared data instances. The following assumptions are true about incoming transactions:

- Incoming transactions are guaranteed to have at least one changed shared data instance since empty transactions are automatically discarded by the interface.
- Changed shared data elements appear in random order in the transaction.
- Transactions remain unmodified in memory until the transaction is purged. This allows the application developer, for example, to reexamine changed instances.

5.1 Retrieving Transactions

The code segment from the spider application shown in Example 5-3 illustrates the basic operations of retrieving information from the shared database.

Specification Steps:

1. ***Get the transaction***. The Serpent interface provides both synchronous and asynchronous calls for getting information from the shared database. The `get_transaction` routine waits until a transaction is available and then returns a handle for this transaction. The `get_transaction_no_wait` routine returns `not_available` when no transaction is available.
2. ***Get each changed shared data instance.*** The `get_first_changed_element` routine returns the first changed shared data element instance in the transaction and marks it as the *current* element. The `get_next_changed_element` routine returns the element directly following the current element and marks it as current. The `null_id` is returned if there is no next element instance on the list.
3. ***Purge the transaction.*** Once the transaction has been fully processed, it should be purged from the system. This frees system resources that could otherwise run out.

Code and comments directly related to the task are emphasized in bold type.

```
Serpent.Serpent_Init(MAIL_BOX,ILL_FILE);
                :
```

```
--
-- Retrieve information from shared database.
--
 done := false;
 while  not done loop
--
--  get next transaction.  If there is none, the process
--   is blocked until one arrives.
--
    transaction := Serpent.Get_Transaction;

    id := Serpent.Get_First_Changed_Element(transaction);
--
-- Get each changed instance in the transaction.
--
    while id /= null_id loop
       id := Serpent.Get_Next_Changed_Element(transaction);
    end loop;

    Serpent.Purge_Transaction(transaction);

 end loop;
```

Example 5-1 Transaction Processing

5.2 Incorporating Changes

Changed element instances from the dialogue need to be processed for any changes in the application domain to be affected. The Serpent application interface provides several routines for the purpose of processing changed shared data elements.

This section makes some simplifying assumptions about the application that may in fact hold true for simple programs. The primary assumption is that the application has created only a fixed number of shared data instances so that the IDs of these instances can be maintained as static, local variables. A secondary assumption is that the application has created no more than one instance of each shared data record.

The code segment from the spider application in Example 5-2 illustrates the operations involved in incorporating changes to shared data elements in static, local variables. Code and comments directly related to the task are emphasized in bold type.

```
--
-- Get each changed record instance in the transaction.
--
 while id /= null_id loop

   element_name := Serpent.Get_Element_Name(transaction, id);
--
-- If the record is a correlation center then this must
-- be the cmc shared data variable.
--
```

```
        if element_name = "cc_sdd" then
          Serpent.Incorporate_Changes(
            transaction, id, cmc'address);
--
-- Otherwise, this must be the gs1 variable.
--
       else
          Serpent.Incorporate_Changes(
           transaction, id, gs1'address);
       end if;

        id := Serpent.Get_Next_Changed_Element(transaction);

    end loop;
```

Example 5-2 Processing Changes to Shared Data Records (Simple Programs)

Specification Steps:

1. ***Get the element name.*** This is a simple call that returns a pointer to the element name. For simple programs that have no more than one instance of a particular shared data record, the element name can be used to identify the shared data instance. In larger, more complex systems it is often useful in determining a class of shared data instances.
2. ***Update local database.*** Shared data variables can be updated using the `incorporate_changes` routine. This routine directly incorporates changes in the shared data instance into the local variable. Components of the shared data record that have not been changed are left untouched. By continually incorporating changes into the initial shared data variable, the application developer is guaranteed that application data remains consistent with user input.
3. ***Update the local database based on the change type.*** The exact type of processing required to update the local database is based primarily on the change type. If this is a new shared data element (e.g., the change type is `create`) the `get_shared_data` function can be used to create a copy of the record instance. If the change type is `modify`, the local shared data instance can be obtained from the hash table. The `incorporate_changes` routine can then be used to update the contents of this instance with changed component values.

5.3 Examining Changes by Component

The Serpent application programmer's interface provides routines that allow the application developer to examine each changed component in a changed record individually.

The operations are illustrated in Example 5-3, taken from the spider chart example. Code and comments directly related to the task are emphasized in bold type.

```
                :
    id := Serpent.Get_First_Changed_Element(transaction);
--
-- Get each changed record instance recording the transaction.
--
    while id /= null_id loop

   Serpent.Get_Element_Name(transaction,id,element_name);
     changed_components :=
Serpent.create_changed_component_list(
                        transaction,id
                        );

     component_node :=
Serpent.Get_First_Node(changed_components);

     while component_node /= null_id loop

       Serpent.Get_Component_Name(
              component_node, component_name);

       type := Serpent.Get_Shared_Data_Type(
         element_name, component_name);

       if type = serpent_id then
          id_data := Serpent.Get_Shared_Data_id(
                     transaction, id, component_name);
        end if;

     component_node :=
Serpent.Get_Next_Node(changed_components,
component_node);

end loop;              -- end loop through list



id := Serpent.Get_Next_Changed_Element(transaction);
    end loop;
                :
```

Example 5-3 Processing Changes to Shared Data Records (Large Systems)

Specification Steps:

1. ***Get the list of changed components.*** A list of changed components can be obtained by using the `create_changed_component_list` function.
2. ***Loop through the list.*** The `get_first_node` and `get_next_node` routines provide a mechanism to sequence the changed components. `Get_component_name` provides a mechanism to get the name from the node.

3. ***Examine the type and/or data.*** The Serpent application programmer's interface provides routines to examine both the type and the data at the component level. The `get_shared_data_type` returns a `serpent_data_type`. The `get_shared_data_id` routines return the component value.

6 Finishing the Application

Other than sending and retrieving data, the application can determine errors from the use of Serpent, record communication between the application and Serpent and exit according to a signal received from the dialogue.

6.1 Error Checking

Each routine in Serpent sets status on exit. It is good software engineering practice to check status after every call to make sure that the routine has executed correctly, and provide appropriate recovery actions if it has not. Example 6-1 illustrates the routines provided by Serpent for examining the status.

```
transaction := Serpent.start_transaction;
if Serpent.get_status /= 0 then
 Serpent.print_status("error during start_transaction");
 Serpent.serpent_cleanup;
end if;
```

Example 6-1   Examining Status

The first of these routines is `get_status`, which returns an enumeration of status codes. Valid statuses returned by each routine in Serpent are defined in Appendix B. Successful execution (or “OK”) is always set to zero; hence, it is possible to make a simple boolean comparison for bad status.

The `print_status` routine prints a user-defined error message and the current status.

6.2 Recording Transactions

Transactions between the application and the dialogue can be recorded using the `start_recording` and `stop_recording` procedures available in the Serpent application programmers interface. After the call to `start_recording` is made, transactions may be sent across the interface. Any number of transactions containing any type or amount of data can be sent. Once `start_recording` has been called, all transactions and associated data will be written to the specified file until the `stop_recording` routine is invoked.

Transactions can be examined using the `format` command described in Section 7.1. This is useful in debugging since it allows the examination of information flow across the interface. Transactions can also be played back to simulate either application or dialogue functionality using the `playback` command described in Section 7.2.

Before testing the application or the dialogue, first record the transactions to be used in testing. Example 6-2 illustrates the basic operations for recording transactions.

```
--
-- Start recording.
--
 Serpent.start_recording("recording", "test data: 5.7.3");
--
-- Send test data.
--
 transaction := Serpent.start_transaction;
                 :
 Serpent.commit_transaction(transaction);

 transaction := Serpent.start_transaction;
                 :
 Serpent.commit_transaction(transaction);

 transaction := Serpent.start_transaction;
                 :
 Serpent.commit_transaction(transaction);

--
-- Stop recording.
--
 Serpent.stop_recording;
                 :
```

Example 6-2 Recording Transactions

Specification Steps:

1. ***Start recording***. The `start_recording` routine takes as parameters both the name of the file in which to save the recording and a message to help identify the recording.
2. ***Send transactions.*** After the call to `start_recording` is made, transactions may be sent across the interface.
3. ***Stop recording.*** The `stop_recording` function closes the current recording file.

6.3 Dialogue Initiated Exit

The dialogue can terminate at any time using the `exit` command available to the dialogue specifier. The `exit` command sends a SIGINT signal to the application. This signal will cause the application to exit immediately, unless a signal handler has been registered with the operating system.

The signal handler describes the steps to be taken when the dialogue initiates an exit. Typically, this involves saving data structures out to permanent storage and exiting the system.

The steps necessary to accomplish this are compiler-dependent.

7 Testing and Debugging

The recording capability discussed in Chapter 3 provides a mechanism to assist in testing and debugging.

7.1 Formatting Recordings

Application recordings are saved in a binary format file. The `format` command distributed with Serpent converts this file into a formatted, easy-to-read report. The information in the file can be useful in isolating problems to either the application or the dialogue.

```
% format recording
FORMATTING JOURNAL FILE: recording

HEADER:
   dialogue name:
   message: no comment at this time
OWNER:
   ill file name: se.ill
   mailbox name: SE_BOX

PARTICIPANT:
   ill file name: se.ill
   mailbox name: DM_BOX

TRANSACTION:
 Fri Jan 25 15:17:13:800 1991
   Sender: SE_BOX
   Receiver: DM_BOX
   Element name: dialogue_sdd Change type: create ID: 955
      shared_data            buffer     UNDEFINED_BUFFER
      termination            buffer     UNDEFINED_BUFFER
      macros                 buffer     UNDEFINED_BUFFER
      externs                buffer     UNDEFINED_BUFFER
      initialization         buffer     UNDEFINED_BUFFER
      count                  integer    0
      name                   string     UNDEFINED_STRING
      prologue               buffer     UNDEFINED_BUFFER
%
```

Example 7-1 Formatting the Recording File

7.2 Playback

Once you have made a recording, it is possible play back the recording to simulate one or more of the Serpent processes. To simulate the spider application, for example, you would run the `playback` command provided with Serpent specifying the name of the recording file and the mailbox of the process to be simulated, as illustrated in Example 7-2.

```
% app-test recording SPIDERA_BOX
Playing back journal file: recording
Message: regression test data, 5.7.3
Playback completed successfully
% _
```

Example 7-2 Testing the Application

Appendix C Commands for Testing Serpent Applications and Dialogues

This appendix contains definitions of commands provided with Serpent to assist in testing Serpent applications and dialogues. The following is a list and short description of each of these commands. A more complete description immediately follows:

Command	Description
`format`	converts a recording file into an easy-to-read report
`playback`	used to play back a recording file

COMMAND

format

Description	The `format` command converts a binary Serpent transaction log to a formatted, easy-to-read report. The report is written to standards output.	
Definition	`format recfile`	
Parameters	`recfile`	The transaction log to be converted.
Returns	`0`	ok

COMMAND

playback

Description	The `playback` command can be used to reenact a session based on a recording file.
Definition	`playback recfile host_mailbox correspondents`

Parameters	`recfile`	The name of the file containing the recording to be played back.
	`host_mailbox`	The mailbox for the process to be simulated.
	`correspondents`	List of correspondents (the default is “all”).
Returns	`0`	ok
	`1`	dialogue not found
	`2`	playback file not found
	`3`	error during playback

Appendix D Spider Example

```
--  -------------------------------------------------------------------
--  Title:       Spider chart demo.
--  Creation:    June 21, 1991
--  Author:      Len Bass
--  Description: Demonstrate use of Ada interface to Serpent
--               This program places data for the spider chart
--               into shared data and retrieves the data entered by
--               the operator
--  -------------------------------------------------------------------

with Text_IO; use Text_IO;
with Serpent;
with S_Types; use S_types;
with SpiderA;
GREEN_STATUS: constant := 0;
YELLOW_STATUS: constant := 1;
RED_STATUS: constant := 2;

procedure Spider is

  package Int_IO is new Integer_IO(integer); use Int_IO;
  package Flt_IO is new Float_IO(long_float); use Flt_IO;


  --**********************
  -- Serpent-specific defs
  --**********************

  Transaction :S_Types.Transaction_Type;      -- transaction handle
 cmc: SpiderA.cc_sdd                          -- shared data variables
 sensor_record: SpiderA.sensor_sdd            -- shared data variables
 cc1_id, cc2_id, sensor_id: id_type           -- object instances
  Changed_id           : S_Types.Id_Type;     -- ID of returned
                                                 shared data
  temporary           : integer;
  Change_Instance_Type : change_type;
  Component_Type      : shared_data_types;
  Change_List         : LIST;
  Component_Node      : NODE;
  Component_Name      : string(1..32);
  String_Data         : string(1..32);
  Integer_Data        : integer;
  Real_Data           : long_float;


************************************************************************
```

```
  procedure Get_Data_Value is

      -- PURPOSE
    --  This procedure retrieves only the changed components for a record.
      --
  begin
--
--   verify that change type is modify
--   if not there is something wrong
--
    Change_Instance_Type := Serpent.Get_Change_Type (Transaction,
                      Changed_id);
    If Change_Instance_Type /= MODIFY then
        Text_IO.Put_Line("Error in Change Type");
    end if;
--
--  now get list of changed components
--
    Change_List := Serpent.Create_Changed_Component_List(Transaction,
                                                    Changed_id);

    Component_Node := Serpent.Get_First_Node(Change_List);

    while Component_Node /= NULL loop
        Serpent.Get_Component_Name(Component_Node, Component_Name);
        Text_IO.Put(Component_Name);
        Text_IO.Put(":  ");
            Component_Type :=
              Serpent.Get_Shared_Data_Type("sensor_sdd",Component_Name);
--
--      Switch based on type of component
--
        case Component_Type is
          when sd_string =>
             Serpent.Get_Shared_Data_String( Transaction,
                     Changed_id,
                     Component_Name,
                     String_Data);
           Text_IO.Put(String_Data);
           Text_IO.Put_Line("");

          when sd_real =>
             Serpent.Get_Shared_Data_Real( Transaction,
                       Changed_id,
                       Component_Name,
                       Real_Data);
           Flt_IO.Put(Real_Data);
           Text_IO.Put_Line("");

          when sd_integer =>
             Serpent.Get_Shared_Data_Integer(Transaction,
                       Changed_id,
```

```
                       Component_Name,
                       Integer_Data);
            Int_IO.Put(Integer_Data);
           Text_IO.Put_Line("");
          when OTHERS =>
            Text_IO.Put_Line("type not in list to process");
        end case;

        Component_Node := Serpent.Get_Next_Node(Change_List,
                                  Component_Node);
        end loop;
        Text_IO.Put_Line("************************************");
        return;
   end;
  --
************************************************************************
  procedure Initialize_Sensor_Record (
               site_abbreviation : in string;
               status : in integer;
                site : in string;
                 etro : in string;
                 )  is

      -- PURPOSE
      --  This procedure initializes all of the data for a sensor shared
data record.
      --
  begin
      sensor_record.site_abbr := site_abbreviation & ASCII.NUL
      sensor_record.status := status;
      sensor_record.site := site & ASCII.NUL;
      set_undefined(sd_string.sensor_record.last_message);
      set_undefined(sd_buffer, sensor_record.rfo);
      sensor_record.etro := etro & ASCII.NUL
      sensor_id := Serpent.Add_Shared_Data(
             Transaction,"sensor_sdd","",sensor_record'address);
--
--    now add two communication lines for the new sensor
--
      comm_line.from := sensor_id;
      comm_line.to := cc1_id;
      set_undefined(sd_string,comm_line.etro);
      comm_line.status := GREEN_STATUS;
      Changed_id := Serpent.Add_Shared_Data(
            Transaction,"communication_line_sdd","",comm_line'address);
      comm_line.to := cc2_id;
      Changed_id := Serpent.Add_Shared_Data(
            Transaction, "communication_line_sdd","",comm_line'address);
       return;
 end;
  --
************************************************************************
```

```
begin

  Serpent.Serpent_Init(FD.MAIL_BOX, FD.ILL_FILE);
  Transaction := Serpent.Start_Transaction;
  if Serpent.get_status /= ok then
    Serpent.print_status (“bad status from start transaction”);
    return;
  end if;

--
--    create shared data for the two correlation centers
--
   cc1_id := Serpent.Add_Shared_Data(
          Transaction,”cc_sdd”,”name”,”CMC”’address);
    temporary := GREEN_STATUS;
    Serpent.Put_Shared_Data(
          Transaction, cc1_id, “cc_sdd”,”status”,temporary’address);
   cc2_id := Serpent.Add_Shared_Data(
          Transaction,”cc_sdd”,”name”,”OFT”’address);
    Serpent.Put_Shared_Data(
          Transaction, cc2_id, “cc_sdd”,”status”,temporary’address);

--
--     create sensor and communication records in shared data
--
  Initialize_Sensor_Record(
            “GS1”, GREEN_STATUS,”Ground Station 1”, “16/1245Z”);
  Initialize_Sensor_Record(
            “GS2”, GREEN_STATUS,”Ground Station 2”, “16/1634Z”);
  Initialize_Sensor_Record(
            “GS3”, GREEN_STATUS,”Ground Station 3”, “12/1245Z”);
  Initialize_Sensor_Record(
            “CLR”, YELLOW_STATUS,”Clear”, “10/1145Z”);
  Initialize_Sensor_Record(
            “THL”, GREEN_STATUS,”Thule”, “16/1255Z”);
  Initialize_Sensor_Record(
            “FYL”, RED_STATUS,”Fylingdales”, “16/1245Z”);
  Initialize_Sensor_Record(
            “BLE”, GREEN_STATUS,”Beale”, “06/1325Z”);
  Initialize_Sensor_Record(
            “OTS”, YELLOW_STATUS,”OTS”, “08/1245Z”);
  Initialize_Sensor_Record(
            “ELD”, GREEN_STATUS,”El Dorado”, “13/0245Z”);
  Initialize_Sensor_Record(
            “WRB”, RED_STATUS,”Warner Robins”, “11/1856Z”);
  Initialize_Sensor_Record(
            “SHY”, GREEN_STATUS,”Shemya”, “14/1254Z”);
  Initialize_Sensor_Record(
            “CAV”, GREEN_STATUS,”Cavalier”, “09/0529Z”);
--
--  commit transaction.  After this procedure call, the data is available
```

```
--  to Serpent for display to the end user
--
  Serpent.Commit_Transaction(Transaction);
  if Serpent.get_status /= ok then
    Serpent.print_status ("bad status from Commit_Transaction");
    return;
  end if;

--
--  get  changes
--
     loop
    Transaction := Serpent.Get_Transaction;
    Changed_id := Serpent.Get_First_Changed_Element(Transaction);

    while Changed_id /= S_Types.Null_ID loop
      Get_Data_Value;
      Changed_id := Serpent.Get_Next_Changed_Element(Transaction);
    end loop;

    Serpent.Purge_Transaction(Transaction);
  end loop;

  Serpent.Serpent_Cleanup;
end Spider;
```

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S CMU/SEI-91-UG-7	5. MONITORING ORGANIZATION REPORT NUMBER(S) CMU/SEI-91-UG-7
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute 6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (City, State and ZIP Code) Carnegie Mellon University Pittsburgh PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/AVS Hanscom Air Force Base, MA 01731
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office 8b. OFFICE SYMBOL (if applicable) ESD/AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS (City, State and ZIP Code) Carnegie Mellon University Pittsburgh PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO: 63752F PROJECT NO.: N/A TASK NO: N/A WORK UNIT NO.: N/A

11. TITLE (Include Security Classification)
Serpent: Ada Application Developer’s Guide

12. PERSONAL AUTHOR(S)
User Interface Project

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM TO	June 1991	~100

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse of necessary and identify by block number)
Serpent, UIMS, user interface management system, user interface generators, Ada, application development

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
Serpent is a user interface management system (UIMS) that supports the development and implementation of user interfaces, providing an editor to specify the user interface and a runtime system that enables communication between the application and the end user. This manual describes how to develop applications using Serpent. Readers are assumed to have read and understood the concepts described in the *Serpent Overview*, as well as to have had experience using the Ada programming language.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution
22a. NAME OF RESPONSIBLE INDIVIDUAL John S. Herman, Capt, USAF	22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7630 22c. OFFICE SYMBOL ESD/AVS (SEI

ABSTRACT —continued from page one, block 19