User's Guide

CMU/SEI-91-UG-8
May 1991

Serpent: Guide to Adding Toolkits

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

signature on file

John Herman
SEI Joint Program Office

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

The Software Engineering Institute is not responsible for any errors contained in these files or in their printed versions, nor for any problems incurred by subsequent versions of this documentation.

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 the 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 the separation of concerns between the user interface and the functional portions of an application, and is easily extended to support additional input/output toolkits.

1.1 This Manual

This manual describes how to add toolkits to Serpent. A generic description of how to integrate any toolkit into Serpent is followed by descriptions of two tools. These tools are Glue, a generalized widget integration facility, and Six, a generic Serpent-to-Xt binding driver. Readers of this guide are assumed to have programming experience in C or Ada, and to have read and understood the concepts described in the *Serpent Overview* and *Serpent: System Guide*.

1.1.1 Organization

This guide is divided into the following chapters:

- **Overview:** A general description of the nature of toolkit integration within the framework of Serpent.
- **Interface Definition:** A description of the process of defining the data interface between a toolkit and Serpent.
- **Interface Mapping Development:** A description of the process of coding the mapping among the Serpent interface data and the toolkit entities.
- **Glue:** A description of how to use Glue, a widget description language which, when used with Six, make the process of integrating Xt-based toolkits easier.
- **Six:** A description of how to use Six, an execution engine that uses the widget description tables produced by Glue to help make integrating Xt-based toolkits easier.

1.1.2 Typographical Conventions

The following conventions are observed in this manual.

Code examples	`Courier typeface`
Variables, attributes, etc.	`Courier typeface`
Syntax	`Courier typeface`
Warnings and Cautions	***Bold, italics***

1.2 Serpent Documentation

The purpose of this guide is to provide you with sufficient information to integrate toolkits into Serpent. The following documents provide information about the Serpent system:

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
Serpent: Ada Application Developer's Guide
Describe how the application interacts with Serpent. These guides describe the runtime interface library, which includes routines that manage such functions as timing, notification of actions, and identification of specific instances of the data.

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

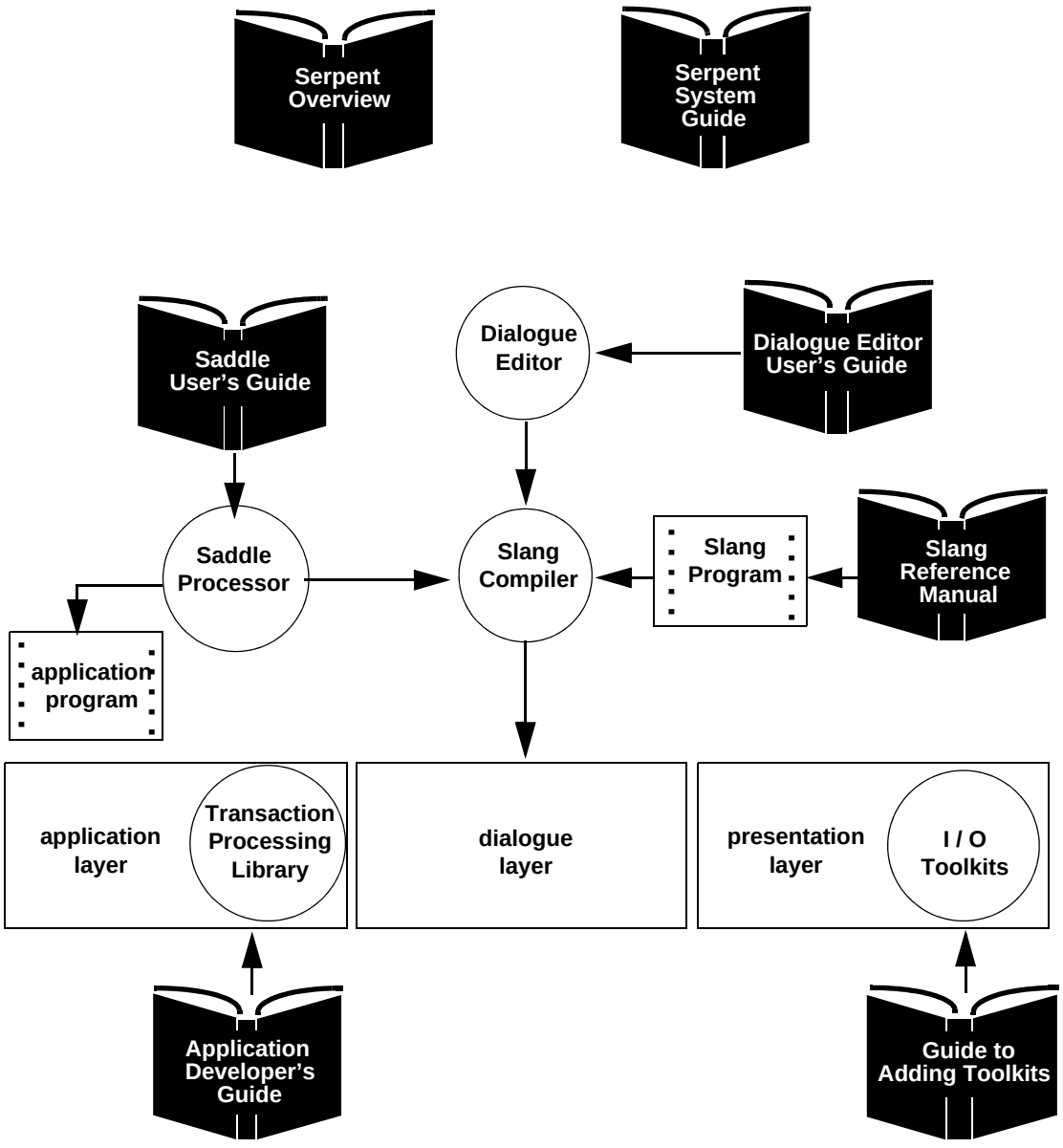

Figure 1-1 Serpent Documents

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 in order to provide the application developer with a toolkit independent interface. The application portion consists of those components of the software system that implement the "core" functionality of the 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.

Given this separation, it is necessary to provide a set of components that actually implement the user interface portion. While Serpent provides most of these, the goal of the toolkit integrator is to provide the last piece, the toolkit-specific portion. Serpent was designed so that the implementation of this component is a relatively straightforward procedure.

Throughout this document the terms *widget* and *object* mean the following:

widget — a toolkit entity

object — an instance of Serpent shared data

There is usually a one-to-one mapping between objects and widgets when a toolkit is integrated into Serpent.

2.1 The 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. Each of the layers is a separate system process. Furthermore, if more than one toolkit is used within a Serpent execution, each toolkit is a separate system process.

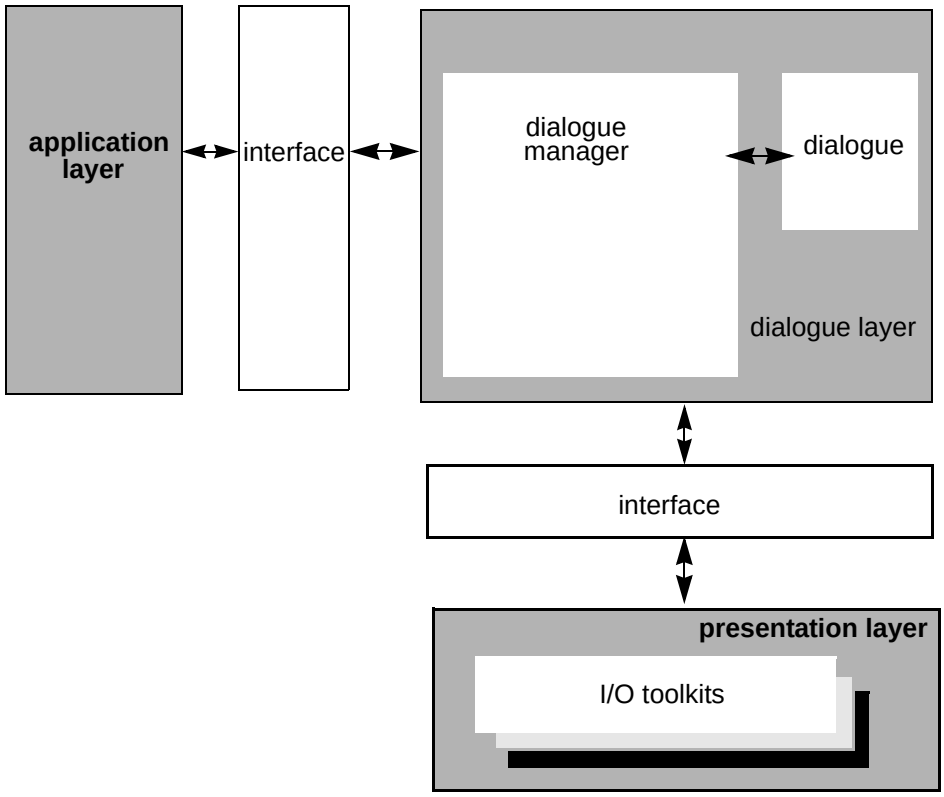

Figure 2-1 Serpent Architecture

The presentation layer consists of various input/output toolkits, referred to simply as toolkits, which have been incorporated into Serpent. A toolkit, in this sense, is an existing hardware/software system that performs some level of generalized interaction with the end user and supports a certain user interface style. Serpent is being distributed with an interface to the Athena widget set and the Motif widget set, both of which run under the X Window System (Version 11). Other toolkits that have been integrated with Serpent include a digital mapping system and a combined video/graphics processor. This guide draws on these efforts and general toolkit integration concepts to offer an approach to the integration of other toolkits with Serpent.

One way of viewing the three levels of the architecture is the level of feedback provided for user input. Roughly, the presentation layer is responsible for lexical functionality, the dialogue layer for syntactic functionality, and the application layer for semantic functionality. For example, in implementing a menu, the presentation layer is responsible for such things as cursor tracking, determining and highlighting the selected menu item, and presenting user feedback to that effect. The dialogue layer is responsible for deciding whether another menu should be presented (and, if so, presenting it) or whether the choice requires application action. The application layer is responsible for implementing the underlying command corresponding to the menu selection.

The user interface for a software system within the UIMS structure is specified formally as a dialogue, which is executed by the dialogue manager at runtime in order to provide a user interface for the 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 independent of the presentation and may be developed in C, Ada, or other programming languages. The application may be a simulation for prototyping purposes or the actual application to be contained in the delivered system. The actions of the application layer are based on knowledge of the specific problem domain.

2.2 Shared Database

Serpent provides an active database model to support the user interface portion of a system. In an active database, multiple processes are allowed to access and to update the database. Any changes made to the database by a particular process are then propagated to the other processes using the database. This active database is shared by the application and the toolkits and is managed by the dialogue manager. Both applications and toolkits can add, modify, query, or remove data from the shared database.

Information in the shared database may be updated by either the application or by a toolkit. Figure 2-2 illustrates the use of the shared database in Serpent. As this figure and Figure 2-1 show, multiple toolkits can run concurrently with Serpent.

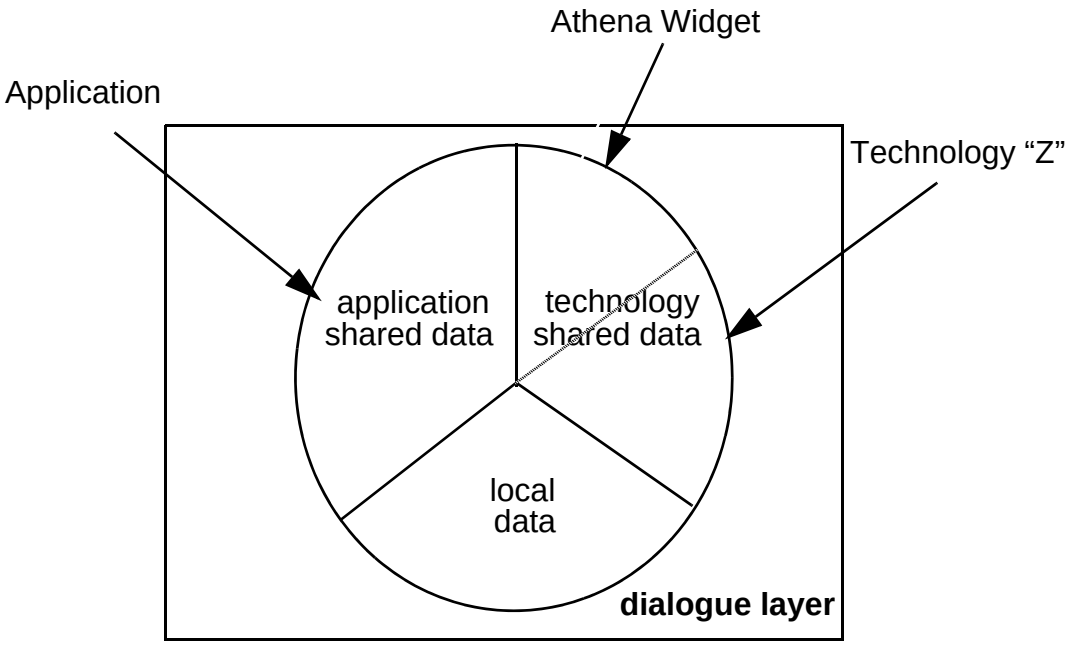

Figure 2-2 Shared Database

Serpent allows the specification, in the dialogue, of constraints between data items in the shared database. These constraints define a mapping between application data and toolkit data. The dialogue manager enforces these constraints 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 toolkits, as appropriate. (See *Serpent: Slang Reference Manual* 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 a schema. A shared data definition (sdd) file is required for each application and for each toolkit.

The shared data definition for a toolkit describes a set of data structures that are available to the user interface designer. (Within Serpent, the user interface designer is more specifically referred to as the dialogue designer or dialogue specifier, as the user interface is specified via a dialogue). Each data structure within a toolkit sdd represents an object template. The dialogue designer thus has a choice of object types from which to implement a particular user interface.

Running a toolkit sdd file through the Saddle processor generates a C header file or Ada package. Another file, called an *ill file*, is also generated and is included into a dialogue that uses the particular toolkit. These files are logically equivalent but exist in different forms for better Serpent system performance.

A shared data definition file consists of a set of aggregate (record) data structures. Any data contained within an element are known as *components* and are considered part of the shared data element. (This corresponds to the database concept of fields within records.) Serpent does not allow nesting of records. (Further details about the specification of shared data definitions may be found in *Serpent: Saddle User's Guide.)*

The record structures in the sdd file are actually templates. During a Serpent run, the templates are used to make shared data instances that are also referred to as *objects*. Each shared data instance is identified by a unique ID. IDs must be maintained by the toolkit-to-Serpent integration code in order to map between the shared data instances (objects) and the toolkit widgets. The dialogue manager communicates with the integration code in terms of objects, and the toolkit communicates with the integration code in terms of widgets. This is discussed further in Chapter 4.

2.3 Inter-Process Communication

Since the dialogue manager, the application, and any toolkits participating in a particular execution of Serpent are separate system processes which 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.

Communication between the dialogue manager and a toolkit may be accomplished synchronously or asynchronously. Asynchronous communication is usually necessary because toolkits must respond to multiple event sources, i.e., both user interactions and dialogue communication via transactions.

The actual physical interaction between processes is via the system-shared memory facility, which differs from Serpent shared data. Shared memory is merely a communication mechanism and is transient, while shared data is persistent within any execution of the Serpent system. The toolkit integrator need not worry about the physical shared memory facility.

2.4 Toolkit Integration into Serpent

Integrating a new toolkit into Serpent requires performing the following tasks:

1. *Define the toolkit capabilities that are to be visible to the dialogue specifier and cast these capabilities in terms of objects.* [**NOTE:** The term *object* in this paragraph refers to generic objects.] Many toolkits already provide a "view" of their capabilities in an object orientation; others present their abilities as a set of functions which can be performed on sets of entities; still others provide a mix of these two. For example, the Athena widget set is essentially a set of objects (widgets), which have certain attributes and predefined behaviors, whereas the SIGGRAPH CORE standard provides segments (akin to objects, although not in the classic sense) and a large set of commands to manipulate segments and other graphical entities which may or may not be contained within segments. As will be seen later, there is nothing which precludes a CORE implementation from being integrated into Serpent. Note, however, that the "object orientation" of Serpent can influence the amount of effort necessary to integrate a toolkit into Serpent. A fairly low-level toolkit that is not based on an object-oriented mechanism, such as one based on CORE, would take more effort to integrate because it would first have to be "objectified."

2. *Define the entities determined in the previous step as a set of Serpent shared data elements*. The combination of this task and the previous task is referred to collectively as *interface definition*, which is covered more extensively in Chapter 4.

3. *Define and code a mapping among Serpent shared data instances (objects).* This mapping is implied by the shared data definition created in the previous step and the actual toolkit entities (widgets) to be used to "implement" the Serpent objects.

4. *Code the toolkit-to-Serpent transaction mechanism interface.* This is the mechanism for communicating with the dialogue manager as defined in the *Application Developer's Guide*. Note that although Serpent provides both synchronous and asynchronous communication mechanisms, the toolkit integrator would probably use the asynchronous form because the toolkit must usually be free to respond to user input. More details about this and the previous task are given in Chapter 4.

It is assumed that the mechanisms for interaction between the toolkit and the end user exist and are well understood by the integrator; these concepts are toolkit-specific and are not addressed here.

3 Interface Definition

This chapter describes how to define the data interface between Serpent and the toolkit that is to be integrated.

3.1 Toolkit Selection

As discussed in Chapter 2, the Serpent toolkit integration paradigm requires that the dialogue manager view the binding to a toolkit as a set of objects. Each object is made up of a set of attributes that describe its physical appearance and possibly its valid user interactions. Three key points are important to consider when selecting a toolkit for integration into Serpent:

1. *The toolkit capabilities must be able to be cast as a set of objects*. All toolkits have a set of entities and functions that can be performed on those entities. However, some toolkits provide a higher abstract level of entities than others. For example, defining a command button in CORE requires describing a rectangular polyline and a bounding box with the same dimensions. The end user feedback mechanism would also have to be defined, and a segment subroutine would have to be written and invoked to handle the button selection. On the other hand, the Athena command button already provides most of these attributes.

2. *All of the toolkit's capabilities should be exposed to the dialogue specifier*, at least as a first cut at the toolkit integration process. Remember, this effort is not to provide a specific user interface, but to give the dialogue designer a pool of capabilities from which to draw. It will be much easier to do this with an object-oriented toolkit than with one that is functionally oriented.

3. As with virtually any programming activity, *tradeoffs must be made between the level of functionality and implementation ease*. While CORE is much more flexible than the Athena widget set, CORE would be much more difficult to integrate into Serpent for reasons which have already been discussed.

3.2 Serpent Object Definition

The first task of integrating a toolkit is to abstract its capabilities in terms of Serpent objects, or more specifically, Serpent shared data templates. These templates serve as the interface between the toolkit binding code and Serpent, as well as define a pool of resources available to the dialogue designer. **NOTE:** The terms *shared data*, *shared data element*, *Serpent object*, and *shared data instance* are used interchangeably in this document.17

An interaction object has two parts:

- a set of attributes, which describe the physical appearance of the object, and
- a set of methods, which describe the set of behaviors the object can exhibit.

3.2.1 Object Attributes

Attributes, as mentioned above, are used to define the physical appearance and behavior of an object. There are three classes of attributes:

1. Direct attributes. These Serpent object attributes usually map directly onto toolkit widget attributes. For example, the Athena command widget has attributes that define the x and y location of the widget; the colors of the background, foreground (text), and border; the thickness of the border; and the like. Setting one of these attributes to a particular value has an obvious effect, e.g., setting the foreground color to red causes the text label to be red.
2. Indirect attributes. There is another class of attributes whose settings may not produce obvious results. For example, setting the attribute which controls command widget sensitivity to false causes the text label in the widget to become gray or otherwise changed in color (depending on the characteristics of the screen).
3. Command attributes. Finally, there is a third class of attributes which can be used for changing the object's behavior or for otherwise downloading information about that object to the toolkit. This type of attribute produces no immediate visible results. For example, in the Athena text widget, a large set of actions is tied to specific keystroke sequences, e.g., the return key, by default, causes a carriage return code to be sent to the widget, followed by a line feed. These defaults can be changed to generate different codes, although changing one causes no visible results at all until that key sequence is entered.

Again, in any toolkit there is a set of widgets which can be manipulated in some way (see Section 3.1). In general, mapping toolkit widgets, be they Athena widgets or CORE segments, into Serpent objects is the most natural approach to defining the interface between the toolkit and Serpent.

3.2.2 Object Methods

A method, in its simplest sense, is a particular value returned by the toolkit binding code to the dialogue manager when a corresponding particular condition occurs with respect to a given widget. What this means is that if a given action is taken on a widget, such as selecting a command widget, a corresponding value is returned which signifies that the action has taken place within the scope of the Serpent object corresponding to that command widget.

It is important to note that a method is treated as a special, well-known attribute by Serpent; i.e., any object description for which operator interaction is defined must have an attribute called `method`. All information passed between methods must take the form of a string, and the defined length of the method must be sufficient to handle the longest string that may be passed.

For example, there is a Serpent object named `xawcommand` which corresponds to the Athena command widget (see Example 3-1). Among the attributes of `xawcommand` is one named `method`, defined as a string of length 50. One of the values that this method can assume is “select,” which signifies that the command widget corresponding to an `xawcommand` Serpent object has been clicked once by the system operator.

3.2.3 Object Semantics

Inherent in the task of defining an interface description is ensuring that the semantics of the data values that are passed between the dialogue manager and the toolkit binding code are well understood. Careful attention should be paid to ensure that the toolkit integrator and the dialogue designer(s) who will use the integrated toolkit agree on what happens when a particular attribute value is set.

In the command widget example given above, the semantics of setting `sensitive` to false are that the color of the label text is gray or otherwise altered. If there is a simple one-to-one mapping between the sensitive attribute of the command widget and the sensitive attribute of the `xawcommand` object, these semantics are the same whether one is using Serpent or programming directly in the Athena toolkit. Defined Serpent objects should follow the same semantics as their toolkit widget counterparts for two reasons:

1. Using the same semantics exposes all of the capabilities of the toolkit widgets, so that anyone using Serpent to manipulate those entities can count on the entities’ behavior to be the same as when directly programmed in the toolkit.

2. Using the same semantics makes the initial adoption of the widgets easier. A new set of semantics need not be invented and implemented on top of the toolkit semantics.

3.3 Shared Data Definition

Once the interface has been defined in terms of objects, their attributes, and their methods, the next step is to cast the interface in the form of a shared data definition, or simply, `sdd`. Example 3-1, taken from the sdd file for the Athena widget set, is written in Saddle and shows the Serpent object that is mapped to the Athena command widget (see *Serpent: Saddle User's Guide* for a complete description of the syntax).

```
xawcommand: record
   ...
  method                    : string[50];
  parent                    : id;
     ...
  background                : string[80];
  backgroundpixmap          : buffer;
  bitmap                    : buffer;
    ...
  font                      : buffer;
  string[80];
  height                    : integer;
  justify                   : integer;
     ...
  label                     : buffer;
     ...
  width                     : integer;
  x                         : integer;
  y                         : integer;
  bottom                    : integer;
  fromhoriz                 : id;
  fromvert                  : id;
  horizdistance             : integer;
  left                      : integer;
  resizable                 : boolean;
  right                     : integer;
  top                       : integer;
  vertdistance              : integer;
end record;
```

Example 3-1 Example Serpent Object: The Athena Command Widget Implementation

Some of the attributes of this Serpent object are direct, like x, y, width, and foreground_color, while others fall into the category of indirect, such as sensitive. Still others are command attributes, such as resizable (whether or not the widget is allowed to be resized), font, and justify (left, center, or right justification). Note the `method` attribute, which is used for passing end user indications from the command widget to the dialogue manager.

A perusal of the command widget section in the Athena widget documentation will show that, with the exception of the `method` attribute, these attributes correspond almost directly to their counterparts in the original widget description.

3.3.1 Creating an SDD File

A complete sdd file consists of three parts: the invocation command, the shared data block, and element declarations. Example 3-2 shows another excerpt from the Serpent object set that implements the Athena widget set: the excerpt contains all of the major parts of an sdd file, but most of the objects are left out for brevity.

```
<< saw >>


saw : shared data

xawcommand: record
    parent              : id;
    x                   : integer;
    y                   : integer;
    ...
    ...
    method              : string [50];
end record;
...
end shared data;
```

Example 3-2 SDD File Major Parts

Anything between double angle brackets (`<< >>`) is treated as the invocation command, i.e., the command necessary to invoke the integrated toolkit. This command is used by the dialogue manager to start the toolkit as an independent UNIX process. In this case, the name of the program that runs Athena Toolkit and the Serpent binding is `saw`.
The end of Chapter 4 goes into more detail about executing toolkits.

The `shared data` statement and the corresponding `end shared data` statement constitute the shared data block, which delineates the declarations that are relevant to this toolkit. Any name for the shared data block is valid, as long as it is comprised of less than 33 alphanumeric characters and underscore; this is also true of any other user-supplied name in Saddle.

The rest of the file comprises the element declarations. These serve as templates from which Serpent objects will be instantiated at runtime. Proceed by encoding each defined object abstraction into an sdd element declaration (see the *Serpent: Saddle User's Guide* for the complete syntax).

Each record description must be in the form of a flat record, i.e., a record cannot be contained within another record. Comments, begun with an exclamation point, and ended with an end-of-line, can appear after any semi-colon. C style comments (started with / and ended with /) are allowed anywhere.

3.3.2 Processing the SDD File

Once the sdd file has been built, execute it with the sdd command. Assuming the file to be processed is called tool.sdd (all sdd files must have the extension.sdd): if you plan to integrate the toolkit using C, use the command

```
sdd tool.sdd
```

Otherwise, if you plan to integrate the toolkit using Ada, use the command

```
sdd -a tool.sdd
```

In either case, the extension (`.sdd`) is optional.

If there are no errors, Saddle prints a success message. Otherwise, Saddle prints an error message, usually because there is a syntax error or because an element name or a component name within an element is used twice. *Serpent: Saddle User's Guide* describes the possible errors and how to correct them.

Running the Saddle processor has several outcomes. A C header file (or Ada package specification file) is created in which all the sdd element declarations are converted into their equivalents in the C (or Ada) language. This file can then be included (or WITHed) by the interface binding code (see Chapter 4). Finally, another file is created, similar to the C or Ada file, except that the declarations are written in an internal Serpent format called ILL (for Interface Layer Language).

Note: All output files are written to the directory containing the source files. Serpent looks in the environment variable `SERPENT_ILL_PATH` for the directories in which to find ILL files. Append the name of the directory containing the ILL files to this environment variable during development.

4 Interface Binding Development

This chapter describes how to code the toolkit binding program, which must provide:

- a map between Serpent objects and the particular toolkit's widgets
- an interface with the Serpent application developer's library

A detailed description of the development process for writing the mapping in C is provided. A description of the use of Ada within Serpent can be found in *Serpent: Ada Application Developer's Guide*.

4.1 C Language Development

Example 4-1 shows an excerpt from the C header file generated by the Saddle processor from the sdd file in Section 3.2. The first two lines in the example define two constants used by the Serpent runtime environment. The `typedef` corresponds to the record defined for the `form_widget` in the sdd file. Note that `method` is defined as `char` [51]; an extra byte is always added to the length of the Saddle string to provide for the null terminator ('\0') required by C strings.

```
...
#define MAIL_BOX "TEST_BOX"
#define ILL_FILE "test ill"
...
typedef struct{
          ...
          id_typeparent;      /* (no element pointer) */
          ...
          int   x;
          int   y;
          ...
          char  method [51];
          ...
          }     xawcommand;
...
```

Example 4-1 Example C Header File

This header file serves as the C description of the interface between Serpent and the toolkit binding code. It must be included in any program component intended for accessing the Serpent objects (shared data instances).

The following sections detail the code required to perform the various functions used within the context of Serpent.

4.1.1 Useful C Routines

A package of useful C routines within the Serpent system is available to the toolkit integrator. This support package is included with the Serpent system and implements some more commonly used data structures, including lists, hash tables, and stacks. The package also implements a mechanism for allocating general-purpose memory; the mechanism is used throughout the Serpent system and is more efficient than `malloc`. A complete description of this package can be found in docs/toolbox.ps within the Serpent file area.

The Serpent mechanism for sharing data also provides a set of routines for manipulating shared data and transactions. These routines are described in *Serpent: C Application Developer's Guide*.

4.1.2 Shared vs. Local Process Data

The term *shared data* is somewhat misleading. The data is not shared physically among processes, only logically. The Serpent transaction mechanism is used to pass the data between processes. If a toolkit binding requires that the shared data be persistent within the process (as it will in most cases), the binding process must allocate local process space for the data and copy the shared data from the transaction. When a data modification comes through a transaction, the toolkit binding must also have a way of associating the id of the modified data with the process local storage. The same is also true for removes via a transaction. The examples that follow use a simple hash table; see Section 4.1.1 to associate shared data ids with the corresponding process data space. A mechanism for mapping a shared data id to any dataspace handle[1] that is necessary for actual toolkit widgets would probably also be necessary but is not included here because of toolkit dependence. However, the id-to-data space hash mechanism for the Motif and Athena widget sets have been used successfully. (Two hash tables were actually used, since the mapping has to go both ways: shared data id to toolkit widget and vice versa.)

4.1.3 Initializing Toolkit to Serpent Interface

The first thing that must be done by the toolkit binding is to initialize the Serpent runtime support environment. Example 4-2 shows the code necessary to perform this operation.

[1] A dataspace handle is a pointer to a specific data structure defining the layout of the widget.

The header file `serpent.h` exposes all of the global Serpent routines and types. Assuming the name of the toolkit sdd file is tool**.**sdd, the resultant header file would be named tool.h.

```
#include "serpent.h" /* serpent interface definition */
#include "tool.h"    /* toolkit sdd */

main ()

    {
    ...
    serpent_init (MAIL_BOX, ILL_FILE); /* register this
                                          toolkit */
    /* here would go any toolkit initialization
       procedures */
    ...
    /* main loop processing, described later */

    }
```

Example 4-2 Serpent Initialization in C

Note: For detailed descriptions of all of the Serpent C routines and types, refer to the *Serpent: C Application Developer's Guide*.

4.1.4 Main Loop

Once the Serpent support environment and the toolkit have been initialized, the toolkit binding code will typically begin an infinite loop that performs two main actions: getting shared data `transaction` variables from the dialogue manager and reacting to any events from the toolkit proper. The events and possible reactions depend upon the toolkit, and therefore are not described here. (See the appropriate toolkit documentation for this implementation.)

Data retrievals can occur synchronously or asynchronously. The more common, asynchronous transactions, are exemplified in Example 4-3:

```
#include "serpent.h"
#include "tool.h"

 ...
    transaction_type transaction;
    id_type id;
    ...
```

```
while (true)
    {
    transaction = get_transaction_no_wait ();
        /* get the handle asynchronously */
   if (transaction != not_available) /* Serpent constant */
    {
    id = get_first_changed_element (transaction);
    while (id != null_id)
      {
      /* process the element */
      ...
      id = get_next_changed_element (transaction);
          }
    purge_transaction (transaction);
    }
    /* do any necessary toolkit specific processing */
    ...
    }
```

Example 4-3 Serpent Asynchronous Shared Data Retrieval in C

In Example 4-3, the `transaction` variable is used as the handle to the transaction, while the `id` variable is used to access each of the elements in the transaction, one at a time. As mentioned previously, the toolkit binding views the Serpent dialogue manager as an active database manager. Thus, any time toolkit shared data is created, modified, or deleted by the dialogue manager, the toolkit binding is automatically informed via a database transaction. The `get_transaction_no_wait` routine returns immediately, regardless of whether or not there is a transaction. If there is no transaction, it returns `not_available`, allowing the toolkit binding program to perform necessary functionality, e.g., responding to toolkit specific stimuli (mouse clicks, etc.) through whatever mechanism is appropriate for the toolkit. If there is a transaction, `get_transaction_no_wait` returns the next unprocessed transaction.

The routines `get_first_changed_element` and `get_next_changed_element` return the value `null_id` if the end of the transaction is reached. After processing the entire transaction, `purge_transaction` should be called to free resources. Finally, as is normal in event processing systems, an infinite loop is placed around the entire process.

Note: This looping mechanism tends to use valuable computing cycles; see Section 4.1.8 for a better mechanism.

4.1.5 Using Retrieved Shared Data

After retrieval, data can be processed to determine its element type and the change type. The previous section discusses one method of determining whether or not an element in a transaction is of change type `new`. The other method is to examine `change_type` to determine whether the element is newly created, modified, or deleted. Example 4-4 shows how to obtain the element name and change type. (This example does not include the computation of the variables from the previous example; it essentially replaces the comment labeled `/* process the changes */`]).

```
change_type change;  /* Serpent enumeration */
string element_name; /* name (type) of each element */
int size;            /* size of the element */
caddr_t addr;        /* pointer to local element data */
...
change = get_change_type (transaction, id);
element_name = get_element_name (transaction, id);
...
```

Example 4-4 Getting the Element Name and Change Type in C

The type `change_type` is an enumeration of whether the element is new, modified, or deleted (with values `create`, `modify`, and `remove`, respectively). The type `element_name` is a string indicating the type of the element. In example Example 4-4, if the string `xawcommand` were sent in a transaction, the value returned from `get_element_name` would be `xawcommand`.

It is usually necessary to maintain a correlation between an element's id and the current data in local process memory associated with the id. In this example, it is assumed that the local process data address of each element is maintained as a local hash table indexed by the element id. If the element is new, space must be allocated for a local copy. The routine `get_length` returns the amount of space to allocate, and `make_node` actually allocates the space. The call to the routine `add_to_hashtable` is used to correlate the element id with the allocated space that will hold the element data. Other routines remove entries from hash tables and remove the memory allocated by `make_node` to handle shared data remove actions. The routine `get_from_hashtable` returns a pointer to the local copy of the element that corresponds to the id.

Note: The routine `make_node` and the hash table routines are part of the support package mentioned in Section 4.1.1. The routine `get_length` is part of the Serpent interface package for sharing data.

The next three subsections illustrate how to process the data based on whether the element is new, modified, or deleted.

4.1.5.1 Element Creation

Example 4-5 shows how to process a newly created element. Bear in mind that there may be processing specific to the toolkit that must be performed when creating a toolkit widget corresponding to the specific Serpent object.

```
LIST change_list;   /* list of changed element
                    values */
LIST component_list;/* current component
                    (attribute) */
caddr_t addr;   /* ptr to local copy of data */
string component_name;
...
switch (change)
   {
   case create: /* Serpent constant of type
                       change_type */
   /* any toolkit specific processing */
   size = get_length (element_name);
   addr = (caddr_t) make_node (size);
   add_to_hashtable (id_table, id, addr);
   incorporate_changes (transaction, id, addr);
   change_list =
         create_changed_component_list (id);
   component_name =
         get_list_next (change_list, NULL);
   while (component_name != NULL) /*loop through the
                                              components */
         {
         /* any component specific processing */
         component_name = get_list_next
(change_list, component_name);
         }
    /* any toolkit specific processing */
    destroy_changed_component_list (change_list);
    break;
    ...
    }
```

Example 4-5 Processing New Elements in C

The `LIST` type is a pointer to the list structure provided in the previously mentioned support package for data structures. Space for the shared data is first allocated in process memory and a mapping between the id and the address of that space is added to a hash table. The `incorporate_changes` routine updates the local copy of the element associated with the id. The routine `create_changed_component_list` returns a pointer to a list of all components in the element that have been changed.

Note: Attributes that are not set specifically in the dialogue have a default setting of `undefined`. (See Section 4.1.8 for further discussion.)

The routine `get_list_next` returns a pointer to the next item in the list. (When called with an initial value of `Null, get_list_next` returns the first item on the list). In this case, as the name of each component is retrieved from the list, the component is processed (usually in some toolkit-specific way) to set a toolkit value. When `get_list_next` returns a value of `NULL`, it is because the end of the list has been reached.

After processing each component, further toolkit-specific processing (such as the actual creation of the toolkit widget corresponding to the Serpent object being processed) may occur, after which `destroy_changed_component_list` should be called to free resources.

4.1.5.2 Element Modification

Processing modified elements, depicted in the following example, is similar to processing new elements in terms of the Serpent mechanisms used. The major difference between creating an element and modifying one is that it is not necessary to allocate space for the data nor to set up a mapping between that space and the element id. When an element is modified, the space previously allocated for it is available through the id. Any toolkit-specific processing would probably be to reflect the changed Serpent object attributes in the corresponding toolkit widget attributes.

```
LIST change_list;   /* list of changed element values */
LIST component_list; /* current component (attribute) */
caddr_t addr;   /* ptr to local copy of data */
string component_name;
...
switch (change)
     {
     case modify: /* Serpent constant of type change_type
*/

          /* any toolkit specific processing */
```

```
        addr = (caddr_t) get_from_hashtable (id_table, id);
        incorporate_changes (transaction, id, addr);
        change_list =
            create_changed_component_list (id);
        component_name =
            get_list_next (change_list, NULL);
        while (component_name != NULL) /* loop through the
                                          components */
           {
           /* any component specific processing */
           c omponent_name = get_list_next (change_list,
               component_name);
           }
         /* any toolkit specific processing */
         destroy_changed_component_list (change_list);
         break;
    ...
    }
```

Example 4-6 Processing Modified Elements in C

4.1.5.3 Element Deletion

Example 4-7 illustrates the processing of deleted elements. The routine `free_node` frees space created for the local copy of the object and `remove_from_hashtable` deletes the entry associated with `id` from the hash table. Toolkit-specific processing might include deleting the toolkit widget corresponding to the Serpent object/element being deleted.

```
caddr_t addr;
      ...
      case remove:
        addr = (caddr_t) get_from_hashtable
                    (id_table, id);
        free_node (addr);
        remove_from_hashtable (id_table, id);

        /* toolkit specific processing */

        break;
      ...
```

Example 4-7 Processing Deleted Elements in C

4.1.6 Modifying Shared Data

Conceptually the opposite of retrieval, updating shared data communicates information from the toolkit binding to the dialogue manager. Recall that the dialogue manager is viewed by the toolkit binding as an active database manager. In keeping with that model, Example 4-8 details the process of starting a transaction, adding data, and committing the transaction.

```
transaction_type transaction;
id_type id, id_a, id_b;
string element_name,
        component_name;
caddr_t data;
...
transaction = start_transaction ();
id = add_shared_data (transaction, element_name,
     component_name, data);
...
put_shared_data (transaction, id_b, element_name,
         component_name, data);
...
remove_shared_data (transaction, element_name, id_a);
...
commit_transaction (transaction);
```

Example 4-8 Updating Shared Data in C

The routine `start_transaction` starts the transaction, returning a handle to the newly opened transaction. The `add_shared_data` routine creates new shared data, allocates space for it in the transaction, sets the attribute named by `component_name` to the value pointed to by `data`, and returns the id of the new shared data instance. If `component_name` is set to `NULL`, `data` is assumed to point to an entire element structure. If `component_name` and `data` are both set to `NULL`, `add_shared_data` simply allocates the space without setting any values.

The routine `put_shared_data` works similarly to `add_shared_data`, except that it assumes that `id` already points to an allocated element. This routine is always used for modifying one or more attributes in an existing element, so it is especially useful for setting a method.

Toolkits need to be able to remove an object from shared data directly. The routine `remove_shared_data` simply deletes the object associated with the `id` of type `element_name` from the shared database.

Finally, the routine `commit_transaction` sends all updates to the dialogue manager, which atomically performs the actual actions.

Note: Do NOT purge the transaction because the routine `purge_transaction` frees the space in shared memory where the transaction temporarily resides while it waits to fetch. The function of the `routine purge_transaction` is used only by transaction receivers. Note that a transaction can have any combination of adds, modifies, and removes.

4.1.7 Terminating Toolkit-to-Serpent Interface

Before ending toolkit processing, the toolkit should terminate the toolkit-to-Serpent interface so that no further transaction activity between Serpent and the toolkit can occur. Termination of this interface is illustrated in Example 4-9:

```
main ()
    {
    ...
    serpent_cleanup ();
    ...
    }
```

Example 4-9 Serpent Termination in C

4.1.8 Helpful Hints

This section offers some hints for improving toolkit integration.

4.1.8.1 Efficient Looping Through Events

Section 4.1.4 describes a looping mechanism by which Serpent transactions can be interleaved with toolkit-specific events. Including a sleep statement in the main loop improves performance in the execution of Serpent dialogues. However, the sleep statement should be no longer than 100 msec; longer sleep causes the toolkit and Serpent to act sluggishly.

4.1.8.2 Undefined Values

All attributes that are not explicitly set by either the dialogue manager or the toolkit (see Section 4.1.5.1) are given default values. The following table presents these default values and the names (contained in a file named serpent.h) for use within any C code.

TYPE	DEFAULT VALUE	C TYPE
boolean	UNDEFINED_BOOLEAN	boolean
integer	UNDEFINED_INTEGER	int
real	UNDEFINED_REAL	double
string	UNDEFINED_STRING	string
id	UNDEFINED_ID	id_type
buffer	UNDEFINED_BUFFER_BODY	caddr_t
	UNDEFINED_BUFFER_LENGTH	int

Note:

1. `boolean, string, id_type,` and `buffer` are all defined Serpent types.
2. When checking for a default buffer value, it is best to check the value of its length.
3. For more information about Serpent component types, see the *Saddle User's Guide*.

4.1.8.3 Linking

Linking the toolkit binding requires the use of various libraries, some of which may be toolkit-specific (see the appropriate toolkit documentation). The Serpent libraries that are needed are `libint.a, liblist.a`, and `libutl.a`, all of which are in the lib directory of the Serpent file area.

4.2 Execution

To execute the toolkit under Serpent, regardless of the implementation language:

1. Make sure that the invocation command in the sdd file is correct.
2. Append the location (directory) of the resultant.ill file (output from Saddle) to the `SERPENT_DATA_PATH` environment variable.
3. Append the location of the executable binding program to the `SERPENT_EXE_PATH` environment variable.

Once these actions have been performed, the new toolkit can be accessed with the statement: `#include "tool.ill"`.

Appendix A Glue

Two tools, Glue and Six, are available to help with the integration of Xt-based toolkits. Glue is a language for defining toolkit widgets so that the addition of widgets to Serpent is simplified. Six is a generic Serpent-to-Xt driver that converts the runtime tables produced by Glue into calls to Xt. These tables can also be used by the Serpent dialogue editor to automate the process of adding widgets to a toolkit-to-Serpent binding.

A.1 What Is Glue?

Serpent provides facilities for interactively constructing user interfaces and then dynamically managing the resulting interface at runtime. A requirement of Serpent is that instances of widget classes must be described in enough detail to be visualized and modified during the construction of the interface both through direct manipulation and property sheets.

To make the repetitive process of adding new Xt-based widget classes easier, Glue uses a widget description to create tables and routines that control the runtime execution of the widget. The tables are read by Six, an Xt-based driver program, which controls the runtime creation, modification, and deletion of widgets. (See Appendix D for a description of Six. The generated tables can also drive other applications, including the dialogue editor (used to create dialogues that specify widgets and their interactions).

Because Six and the dialogue editor are table driven, they can capitalize on the large amounts of redundancy in a widget set. Consequently, adding a widget that uses existing data types to Serpent takes approximately half an hour. If new data types are needed, the integration time is longer, but still fairly short.

In addition, Glue and Six allow limited subclassing of existing widgets and specification of runtime defaults. This allows different users to have different default sets for a particular widget set. Glue also allows for widget-specific “escape” routines and for defining special actions or error checking.

A.2 Glue Syntax

This section describes the syntax and semantics of the Glue language, and provides examples of its use.

The Glue language consists of four types of components, all of which may be freely intermingled in a Glue description:

1. String definitions
2. Global variables
3. Equivalence types
4. Widget descriptions

Example A-1 shows a simple Glue description for an imaginary toolkit. Explanations of the components in this description are given in the remainder of this section.

```
#include "glueXt.h"

Saddle_command = "six test";/* Global variables */
description = "This is a simple example";

Boolean  :     boolean;     /* Equivalence types */
Position:integer;
Pixel    :     string[80];

thing_widget {              /* Widget descriptions */
    description = "This is a thing widget description";
    class = XtThing;
    include_file = "Xt/thing.h";
    attributes = {
                "allow_user_move" : {
                    x_type = Boolean;
                    serpent_default = false;
                    access = !technology;
                    }
                XtNx : {
                    x_type = Position;
                    serpent_default = 10;
                    description = "X location";
                    }
                XtNforeground : {
                    serpent_name = "foreground_color";
                    x_type = Pixel;
                    }
                };
    methods = {
              move : {
```

```
                parameters = XtNx, "allow_user_move";
                }
            };
};
```

Example A-1 Simple Glue Description

Processing a description file generates at least two output files, the Saddle and toolkit files. Two optional output files, the methods and binding files, are also generated.

1. *Saddle file.* This file contains the Saddle description of the toolkit. The file must be compiled into an `.ill` file for use by the Serpent compiler before the toolkit can be used in any dialogue specification.
2. *Toolkit file.* This file is used by both Six and the dialogue editor. The toolkit file contains all of the information of the Glue specification, but is in a binary format for quick scanning.
3. *Methods file*. This optionally generated file contains the C routine stubs necessary for implementing the Slang method routines.
4. *Binding file*. This optionally generated file contains the C language bindings to X that allow the methods code to be executed. It also contains special routine handles for check routines and X callback routines specified by the toolkit integrator.

The default file name extension for a Glue description file is .gl; for the generated Saddle file, .sdd; for the generated toolkit file, .tx. Thus, for a Glue description file named motif.gl, the Saddle file would be `motif.sdd`, and the toolkit file would be `motif.tx`. If the toolkit integrator specified that the methods and binding files should also be generated, they would be called `motif_meth.c` and `motif_bind.c` respectively. See Sections A.3 and A.4 for more details on the command-line switches and generated output files.

A.2.1 Compiler Basics

Most of the basics of the Glue compiler will be familiar to users of the C programming language. With very few exceptions, the fundamentals of Glue are common to most programming languages.

A.2.1.1 Comments

Comments in Glue follow the standard C syntax. Specifically, comments start with the characters /* and end with the characters */. You cannot nest comments. For example, the following is treated as one comment: `/* This is /* a single comment */ .`

A.2.1.2 Preprocessor

Before the Glue compiler begins parsing, the Glue source code is passed through the standard C preprocessor (`/lib/cpp`). This means that the Glue source file may contain the standard preprocessor directives, namely:

```
#define   #undef
#ifndef   #ifndef
#else     #endif
#if
```

A.2.1.3 CaseInsensitivity

Unlike C, the Glue compiler is mostly case insensitive. Specifically, keywords to Glue may be specified in upper, lower, or mixed case. Where values need to be passed to the toolkit or the generated C files, the case in the Glue file will be preserved (e.g., quoted text has the case of the text preserved).

A.2.1.4 DataTypes

Glue recognizes basic data types:

- integer
- boolean
- real
- string
- C identifier

These types are assigned to various components of a Glue description. The following sections outline the representations of these data types.

A.2.1.5 Integers

Integer values in Glue are represented as the sequence of the digits `0` through `9`. No special recognition is made for octal numbers (in this respect, Glue differs from C syntax). However, any number preceded by the characters 0x or 0X is considered to be a hexadecimal constant.

A.2.1.6 Boolean

Boolean values in Glue are represented by the two special tokens `true` and `false`. In keeping with Glue convention, the two special tokens may be entered with any capitalization. Thus, the tokens `true, True, TRUE`, and `tRuE` are considered to be equal.

A.2.1.7 Reals

Real values are represented in Glue as any sequence of digits that include a decimal. The use of an exponent value is *not* supported by the Glue compiler. The following are all examples of real values in Glue:

2.71828 .625 98.

A.2.1.8 Strings

Strings in Glue are represented as a sequence of characters enclosed in quotation marks. The Glue compiler allows special characters to be included in a string, provided they are “escaped” in a manner similar to the C compiler. The following escape characters are recognized by the Glue compiler:

`\b`	Backspace
`\f`	Formfeed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\"`	Quotation mark
`\'`	Apostrophe (for compatibility)
`\0`	Null (Using this is probably an error)
`\nnn`	Any 1, 2, or 3 octal characters are interpreted as a single character whose ascii code is *nnn*. Note that the strings `"\0508", "\508"`, and”(`8`” are equivalent.
`\<newline>`	A backslash at the end of a line indicates that the end of line is to be ignored. The character immediately after the end of line is read as if it occurred at the location of the line-terminating backslash.

A.2.1.9 C Identifiers

In some cases, the Glue compiler needs to pass the name of a C identifier to the generated toolkit description. C identifiers are passed as unquoted sequences of upper and lower case letters, digits, and the underscore character (that is, the characters [`_a-zA-Z0-9`]). The case of the characters representing a C identifier is preserved in the generated toolkit description file.

A.2.2 String Definitions

Although technically only seen by the preprocessor, string definitions are an essential component of a Glue description. In Example A-1, the string definitions are contained in the file `glueXt.h` and define the names `XtNx`, `XtNforeground`, etc. The name-to-string mapping, which depends upon the toolkit, is defined by each widget set in a special C include file.

Normally, an include file for a widget set resides in a toolkit-specific directory. This include file often contains declarations of C external variables, typedefs, etc., which are unknown by the Glue purser. To ameliorate this problem, a subset of the toolkit include file is usually generated by the toolkit integrator with a simple Unix script. For the example in Example A-1, the include file might be generated with the command:

```
grep "^#" /usr/include/X11/Xt.h > glueXt.h
```

The resulting file contains only preprocessor commands (especially those defining string equivalences) that can be safely used by Glue.

A.2.3 Global Variables

Global variables are optional variables that allow the toolkit integrator to modify general parameters of Glue. A global variable takes a string as a modifier. The syntax of a global variable declaration is:

```
global_variable: variable_name '=' string_value ';'
```

There are currently two global variables that can be set in Glue. They are:

`description` — If set, this variable contains a description of the toolkit to which this Glue file corresponds. An example of the use of this variable is:

```
description = "Motif Glue Description";
```

`saddle_command` — If set, this variable contains the string that is passed into the generated Saddle file as the command to be executed when the toolkit is run. If, for example, the generated Saddle file were to be used with Six (the Xt interface driver), the use of this variable would be:

```
Saddle_command = "six motif";
```

A.2.4 Equivalence Types

Equivalence types enable mapping various toolkit data types (i.e., X11 attribute and widget types) to Serpent data types. The purpose is to allow the toolkit integrator to map toolkit types into Serpent types so that Serpent will know how to interpret toolkit data. Equivalence types are used in the widget description portion of a Glue description file (see Section A.2.5 for details).

The syntax for an equivalence type specification is:

```
type_equiv: toolkit ":" serpent_type
serpent_type: "boolean"
              | "buffer"
              | "integer"
              | "integer" "+" "procedure"
              | "real"
              | "string" "[" INTEGER "]"
              | "id"
```

The six Serpent types that may be used on the right-hand side of an equivalence type are:[1]

- boolean
- buffer
- integer
- real
- string
- id

[1] Although `adi` and `undefined` are legal Serpent types, they are not allowed in Glue.

The `string` type is special in that it is mandatory to specify a string size; string size is the number of characters reserved on the Serpent side of the interface to store the string. The size of a string is represented as an integer enclosed in square braces following the word `string`.

The `integer` type is special in that it is used to maintain addresses. Under X, many widgets have associated attributes that contain the address of a callback procedures list (e.g., `XmNdestroyCallback`). Although there is no address type in Serpent, the address of a callback list may be stored in a Serpent `integer`. To indicate to Glue that the integer is used for a callback address (instead of simply a plain integer value), the annotation `+procedure` is placed after the word `integer`, which changes the way in which Glue generates the binding file. See Section A.3.4 for details.

Some examples of equivalence type specifications can be seen in Figure A-2.

```
Boolean         : boolean;
caddr_t         : buffer;
Cardinal        : integer;
char            : string[1];
Screen          : integer;
Widget          : id;
WidgetList      : BUFFER;
XtCallbackList  : Integer+Procedure;
MethodName      : string[50];
```

Figure A-2 Equivalence Type Examples

Note that the mapping is many-to-1; that is, more than one toolkit type may be equated to a single Serpent type.

A.2.5 Widget Descriptions

This portion of a Glue description allows the toolkit integrator to enumerate the widgets associated with a toolkit, along with the characteristics, attributes, and methods associated with each widget.

A widget description consists of the name of the widget, followed by a list of widget description components. The list of components should be enclosed in braces and the components should be separated by semicolons; see Example A-3 for an example. The name of the widget is the name that the toolkit integrator wants to be accessed in the Serpent domain. In the toolkit domain, a widget is accessed by its handle (i.e., a pointer to a specific data structure defining the layout of the widget). In the Serpent domain, however, a widget is accessed by its name--the name used in the Glue widget description.

The syntax of a widget description component is:

```
component : component_name "=" value
```

The following is a list of all possible components of a widget description. The list of components should be enclosed in braces and the components should be separated by semicolons; see Example A-3. All components are optional:

`attributes` This component is a list of the attributes of a widget; the attributes should be separated by semicolons and enclosed in braces. More details on this widget description component are covered in detail in Section A.2.5.1.

`check_routine` This component can be used to test the value of a widget attribute to be sure that it is reasonable. For instance, the X Toolkit allows the user to create a widget with a height and width of zero. While this is legal from the Serpent standpoint, it does not make any sense from a user interface standpoint. The `check_routine` can test for this condition.
The value of the `check_routine` widget description component is a C token, which is assumed to be the name of a routine. References to this routine are contained in the generated binding file, and the routine is automatically called by Six when a widget of this type is created or modified. (The toolkit integrator must supply the actual routine body, however, before linking a new version of Six.)

`class` The value of `class` is the widget class; this value is specified in the documentation for the particular widget type. For example, in the Motif ‘‘Bulletin Board’’ widget, the class is `xmBulletinBoardWidgetClass`.
The value of the class widget description component is a C token. The name of this C token is placed into a special array in the generated binding file, so that Six can access the actual value of the widget class through the `WIDG` structure described in Appendix A.5.1.3.
When this component is not specified, a warning statement will be issued by the Glue compiler. Not specifying this component is usually an error, so the compiler will call attention to its absence.

`description` This component describes the widget; its value is a quoted string. This widget description component is optional, and is typically used by the dialogue editor to provide help information to the editor user. Glue does not interpret the contents of this component at all--the toolkit integrator may make the description as simple or complex as desired.

`include_file` The widget include file is the `.h` file that describes the contents of the widget. This file will be `#included` in the generated methods and binding files that Glue generates from the description file.

The value of the `include_file` widget description component is a quoted string containing the name of a file that can be used in an `#include` statement.

When this component is not specified, a warning statement will be issued by the Glue compiler. Not specifying this component is usually an error, so the compiler will call attention to its absence.

`methods` This component lists the methods of a widget; see Appendix A.2.5.2 for details.

`widget_type` This component takes a token as its value. If this component is not present in a widget description, the value `widget` is assumed.

The following are possible values:

- `shell` Indicates that the widget is a shell widget (which may contain other widgets). One example of this is the Motif ‘‘TopLevelShell’’ widget. For more details on what a shell widget is, consult the appropriate toolkit documentation.
- `override` Indicates that the widget is an override shell widget (which may contain other widgets, but which bypasses the window manager). One example of this is the Motif ‘‘MenuShell’’ widget. For more details on what an override shell widget is, consult your toolkit documentation.
- `widget` Indicates that this is an ordinary widget. This is the default value for this field.

- `none` — Indicates that this is not a true toolkit widget. This mechanism is provided for creating a new ‘‘widget’’ that functions outside or in addition to the toolkit. One example of this is the screen object in the Serpent-Motif binding `(smo)`, which allows the Slang user to query the characteristics of the display screen.

`userdef` — This component is us to allow the specifier to communicate with any application that reads the output of Glue. Its value is a quoted string. This component is optional and is present to provide some simple extensibility to the Glue interface. Application programs may read the characters of this string and interpret them ‘‘at whim’’ through the `WIDG` structure described in Appendix A.5.1.3. Glue does not interpret the contents of this component at all; thus, the toolkit integrator may make its contents as simple or complex as desired.

A simple example of a widget description is shown in Example A-3.

```
Specimen {
    include_file = "Xx/Sample.h";
    class = xmSampleWidgetClass;
    description = "This is a sample widget";
    attributes = {
                "parent" : {
                    x_type = Widget;
                    access = !technology, !set;
                    description = "This is poppa bear";
                    }
                XmNhorizontalSpacing : {
                    x_type = int;
                    serpent_default = 2;
                    userdef = "IeH3err";
                    }
                XmNrubberPositioning : {
                    x_type = Boolean;
                   serpent_name = "boing";
                   technology_default = true;
                    }
#               include "Part_and.at"
#               include "Parcel.at"
                }
    methods = {
                select : {
```

```
                                parameters = XmNhorizontalSpacing,
    "boing";
                                description = "Click me";
                                c_routine = select_me;
                                }
    #               include "move.me"
    #               include "resize.me"
                    }
        };
```

Example A-3 Simple Example of a Widget Description

In this example, the name of the widget is `Specimen`; note that this name does not have to be the same as the widget designer's name for the widget (in this case, `Sample`). No `shell_widget` component is included, so the widget is assumed to not be a shell widget. The details of the `attributes` and `methods` components are covered more fully in the following sections.

A.2.5.1 Attributes

Each widget comprises a list of attributes, which may originate in the toolkit (e.g., the position on the screen) or in Serpent (e.g., whether the user is allowed to move the widget around on the screen). The attribute component of a widget description lists all of the attributes of the widget along with their relevant characteristics. As can be seen in Example A-3, the attributes are given as a list of optional components, separated by commas. Each component consists of the name of the attribute (in the form of a quoted string), followed by the characteristics of the attribute. In cases where a quoted string does not appear (as with `XmNhorizontalSpacing`), a preprocessor macro has been defined that equates the name to a string; for example:

```
#define XmNhorizontalSpacing "horizontalSpacing"
```

The characteristics of each attribute are specified as a list enclosed in braces and terminated with a semicolon. When there are attributes common to multiple widgets, the use of `#include` files is supported and encouraged. The complete list of attribute characteristics is:

`access` Defines how the toolkit interface (Six) and the Slang dialogues access the attribute. The value assigned to this component is a list of access classes (separated by commas). Possible access classes are:

- `all` Specifies that all of the above accesses are available to the attribute; `all` is the default value if the access component is not included.

- `create` Allows the attribute to be set at widget creation.
- `get` Allows the value of this attribute to be retrieved at any time.
- `serpent` Allows the Slang dialogue to access the attribute. This access should be present for all attributes that the toolkit integrator wishes to export to the dialogue. By not specifying `serpent` access, the toolkit integrator can hide some of the details of a widget from the Serpent user.
- `set` Allows the value of this attribute to be written at any time.
- `technology` Allows the toolkit to access the attribute. This access should be present for all attributes defined by the toolkit. (Note that in addition to those in the toolkit widget, some attributes are provided by Serpent. One example of this is the `allow_user_move` attribute. Attributes provided by Serpent should not be accessed by the toolkit.)

Any of the components may be negated by preceding it with an exclamation mark. If only positive access assertions are listed, Glue assumes that *only* those accesses are allowed. If only negative access assertions are listed, Glue assumes that all accesses are allowed *except* those listed. If both positive and negative assertions are listed, all access types must be specified in either positive or negative form. The following three access lists are equivalent:

```
access = !technology, !set;
access = serpent, get, create;
access = serpent, !technology, get, !set, create;
```

One or both of the `technology` and `serpent` access types, and at least one of the `create, set,` and `get` access types must be specified if the default value of `all` is not used.

`description` Describes the attribute in a quoted string; this attribute is optional, and is typically used by the dialogue editor to provide help information to the editor user. Glue does not interpret the contents of this component at all–the toolkit integrator may make the description as simple or complex as desired.

`serpent_default` Specifies a Serpent-specific default value that overrides the default value provided by the toolkit. For example, a site may wish to change the default background and foreground color for all widgets in order to adhere to a project or company standard.

The type of this attribute must match the implicitly declared Serpent type of the attribute (see the x_type attribute component). This means that a string attribute must have a string as its default value, etc. Serpent_default overrides technology_default. If neither a serpent_default nor technology_default is specified, neither will be available to the applications, but widgets will still be created with the toolkit default values.

`serpent_name` Specifies a Serpent-specific name (a string value) for an attribute that overrides the name specified by the toolkit. If `serpent_name` is not specified, Serpent uses the name used in the toolkit.

`technology_default` Although the toolkit automatically provides a default value for each attribute, some applications interfacing with Glue need to know what these default values are. The `technology_default` field provides a means to explicitly note the technology default.

The type of this attribute must match the implicitly declared Serpent type of the attribute (see the x_type attribute component). This means that a string attribute must have a string as its default value, etc. If serpent_default is also specified, it overrides technology_default. If neither serpent_default nor technology_default is specified, neither will be available to the applications, but widgets will still be created with the toolkit default values.

Note: This field does not actually change the technology default values, but simply provides a means to access it. This means that when a new release of the toolkit is acquired, the toolkit integrator must verify that the values specified in the Glue description actually match that of the toolkit.

`userdef` This optional component allows the specifier to communicate with any application that needs the output of Glue. Application programs may read the contents of this field and interpret them ‘‘at whim’’ through the `ATTR` structure described in Appendix A.5.1.4. Glue does not interpret the contents of this component at all--the toolkit integrator may make its contents as simple or complex as desired.

`x_type` Declares the toolkit type of the attribute. This type is used to access the attribute within the widget in the toolkit interface. It is mandatory that the toolkit type of the variable be specified. Declaring a toolkit type also implicitly declares the Serpent type of the attribute via the Equivalence Types section of the Glue description (Section A.2.4).

A.2.5.2 Methods

Each widget optionally has one or more methods associated with it. These methods are reflected in routines written by the toolkit integrator, which are called when the Serpent user performs some action (e.g., clicking in a command widget or moving a widget on the screen). The methods component of a widget description lists all of the methods of the widget along with their relevant (from the standpoint of the interface) components. The actual contents of the C routine are given in Section A.3.3 and must be written separately from the Glue description. As can be seen in Example A-3, the methods are given as a list of components, optionally separated by commas. Each component consists of a method name, followed by the components of the method. The name of each method is given as a C token (i.e., not as a quoted string). The method name is exported to Serpent as the name used in Slang dialogues.

The list of method components is terminated with a semicolon and enclosed in braces. As with attributes, the use of `#include` files is supported and encouraged. The complete list of method components is:

`c_routine` — Names the routine to be generated by Glue in the methods file (see Section A.3.3 for more details). By default, the name of the routine will be the name of the method followed by `_method` (i.e., the "move" method will be accessed by a routine named `move_method`). If the default value is not desired, an explicit name may be declared with the optional `c_routine` component.

This feature is most often used when two or more widgets have methods with identical names but different functions. One example is using widgets from two technologies in one Serpent dialogue: the "move" method may be valid in both technologies, but the actions may be radically different. Similarly, two widgets may share one C routine, even though the names of the methods may be different.

`description` — Describes the method in a quoted string. This method description component is optional, and is typically used by the dialogue editor to provide help information to the editor user. Glue does not interpret the contents of this component at all; hence, the toolkit integrator may make the description as simple or complex as desired.

`parameters` — Optionally lists the parameters that are to be passed between the method and the interface. The parameters are supplied as a list of the Serpent names of the parameters, separated by commas. (Note that when a Serpent name is not explicitly specified with the `serpent_name` attribute component, the Serpent name is the same as the toolkit name.)

For each method that shares any given method routine, Glue ensures that the names and number of parameters match; that is, any other method in any other widget that uses the method routine select_me must also have the parameters “horizontalSpacing” and “boing”, and no others. Note that the sharing of routines is determined either explicitly via the c_routine component, or implicitly via the method name.

`userdef` — Optionally provides some simple extensibility to the Glue interface. Its value is a quoted string. Application programs may read the contents of this field and interpret them ‘‘at whim’’ through the `METH` structure described in Section A.5.1.4.1. Glue does not interpret the contents of this component at all; hence, the toolkit integrator may make its contents as simple or complex as desired.

A.3 Files Generated by Glue

The Glue compiler generates two or four files from a single Glue description, depending on whether the command line option `-M` is present (see Section A.4 for more details). In the following section, it is assumed that the Glue description file is the file shown in Example A-1, and is called `test.gl`.

A.3.1 Toolkit File

The generated toolkit file, which has the file extension `.tx` (`test.tx`, in the preceding example), contains all of the original Glue description in a machine-dependent, binary representation. Use the routines and data structures described in Section A.5 to read the generated toolkit file.

A.3.2 Saddle File

The generated Saddle file contains the shared data description necessary for the Serpent dialogue manager to communicate with the toolkit manager. From a Glue description file named `test.gl`, the Glue compiler creates a Saddle file called `test.sdd`. This Saddle file, shown in Example A-4, has to be processed with the Saddle compiler before using it with the Slang compiler.

```
<<six test>>
test : shared data
  thing_widget : record
    allow_user_move        : boolean;
    x                      : integer;
    foreground_color       : string[80];
  end record;
end shared data;
```

Example A-4 Generated Saddle File

The first line of the Saddle file contains either a default value--the name of the toolkit (in this case `test`)--or the Saddle command specified by the toolkit integrator in the Glue description (in this case `six test`). Only one value appears in the Saddle command: the value specified in the Glue description overrides the default value.

The remainder of the Saddle file contains the shared data descriptions. Each widget from the Glue description has its own shared data record. The names of the shared data elements correspond to the Serpent names of the widget attributes from the Glue description. Where not explicitly defined (as in `allow_user_move` and `foreground_color`), the shared data element names are the same as the toolkit names.

The type of each shared data element is the Serpent type that was equated to the X type named in the Glue description. For example, the attribute XtNx[1] has an X type of `Position`. The toolkit type `Position` was equated to Serpent type `integer` in the Equivalence types section, so in the `thing_widget` shared data record, the shared data element `x` is of type `integer`.

A.3.3 Methods File

The generated methods file contains the skeleton of the C code needed to allow Six to communicate with the Serpent dialogue manager to execute Slang methods. From a Glue description file named `test.gl`, the Glue compiler creates a methods file called `test_meth.c`. This methods file, shown in Example A-5, has to be compiled and linked with Six before it can be used. Note that the methods file will be generated only if the `-M` switch is specified when the Glue compiler is run. Note also that if the toolkit integrator has edited the skeleton methods file, the edited file will be overwritten the next time Glue is run with the `-M` option (see Section A.4 for more details). Consequently, the generated files should never be edited; only the `#included` files (such as `move.c`) should be edited.

---

[1] The name `XtNx` is simply a preprocessor constant. The file `glueXt.h` has a line containing the command:

```
#define XtNx "x"
```

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include "isd.h"
#include "glue_intf.h"
#include "glue_inst.h"
/* Include files generated by Glue from test.gl */
#include <Xt/thing.h>
/***********************************************
 * Method name: move
 * Routine name: move_method
 * Parameters:
 *       x                  : integer
 *       allow_user_move: boolean
 */
void move_method (w, event, params, num_params)
Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
{
#include "move.c"
}
```

Example A-5 Generated Methods File

The methods file begins with a number of `include` directives. Note that the `include_file` specified in the widget description (that is, `Xt/thing.h`) is listed here. For every include file listed in the Glue description, a corresponding `include` directive will appear.

Following this is a C routine. The name of the C routine is generated from the name of the method specified in the widget. In this case, the name of the routine is `move_method`, since the name given to the method in the `thing_widget` was move. If the toolkit integrator wishes to use a different name (because of slightly differing behaviors or because of name conflict), an alternative name may be specified by the `c_routine` method component. A separate routine will be created in the methods file for each unique method or routine name in the Glue description.

The body of the routine is simply an `#include` directive for code generated by the interface designer. The body of this code usually contains declarations of local variables and actions which interface with the Serpent runtime. A sample of the body of a method routine is shown in Example A-6. The toolkit integrator does not need to follow this template; it is merely provided as an example.

```
isd_trans trans;
static char *w_name = "thing_widget";
INST *inst = glue_get_by_widget (w);
ATTR *x = glue_get_attr_from_instance
                    (inst, "x", SERP_NAMES);
ATTR *allow_user_move = glue_get_attr_from_instance
                    (inst, "allow_user_move", SERP_NAMES);
/*
 * Your method specific code goes here, presumably doing
 * something intelligent with the method parameters...
 */
/*
 * Send the transactions to the interface
 */
trans = isd_start_transaction ();
isd_put_shared_data (trans, inst->id, w_name, "method",
                     "move");
isd_put_shared_data (trans, inst->id, w_name,
                            "x",
                            &x->value.i);
isd_put_shared_data (trans, inst->id, w_name,
                            "allow_user_move",
                            &allow_user_move->value.i);
isd_commit_transaction (trans);
```

Example A-6 Sample Method Internals

For each attribute which is potentially modified by a method (specified by the `parameters` method modifier), a variable of type `ATTR *` is declared whose name is the same as the Serpent name of the parameter attribute. In this case, two `ATTR *` variables are declared and initialized, namely `x` and `allow_user_move`. Note that the names of the variables are the Serpent names (which in this case happen to be the same as the toolkit names). If the toolkit integrator had explicitly specified a Serpent name (such as `boing` in Example A-3), this is the name that would be used in the methods code.

After the variables are declared, a space is left for the toolkit integrator to write method-specific C code. It is anticipated that the C routine will use the variables that have been previously initialized, although the toolkit integrator is free to add any needed variable declarations and actions.

Once the method-specific code has completed execution, the Serpent dialogue manager must be informed of any changes made to the method parameter variables. The generated methods code concludes with a transaction to send the parameter variables back across the Serpent/toolkit interface with calls to `isd_put_shared_data`. One call will be generated for each of the parameter variables.

A.3.4 Bindings File

The generated bindings file contains the C code needed to interface a particular widget set (e.g., Athena or Motif) to the Xt-based driver program Six. From a Glue description file named `test.gl`, the Glue compiler would create a bindings file called `test_bind.c`. This bindings file, shown in Example A-7, would have to be compiled and linked with Six before it could be used. Note that the bindings file will only be generated if the -M switch is specified when the Glue compiler is run (see Appendix A.4 for more details).

Note: None of the entries in the generated bindings file should be altered by the toolkit integrator. Doing so may result in erroneous behavior by the Glue interface routines.

```
#include <X11/Intrinsic.h>

/* Include files generated by Glue from test.gl */
#include <Xt/thing.h>

extern void move_method();

int glue_actions_count = 1;

XtActionsRec glue_actions[] = {
         { "move",  move_method },
};

int glueClassCount = 1;

WidgetClass *glueClassList[1] = {
    &XtThing,
};
int glueSpecialProcCount = 0;

typedef void (*PFV)();
PFV glueSpecialProcList[0] = {
};
```

Example A-7 Generated Bindings File

The bindings file begins with a number of `include` directives. Note that the `include_file` specified in the widget description is listed here. For every include file listed in the Glue description, a corresponding `include` directive will appear.

Following this, a list of external declarations for all the routines implicitly or explicitly declared in the Glue description is given. These routines include the methods routines, check routines, and special callback routines (all of which are explained below).

The variable glue_actions_count is used internally by the routine glue_get_technology (see Appendix A.5.2.1) to count the number of method routines listed in the array glue_action. This latter array provides the hook needed by the X Toolkit to use the method routines.

The variable glueClassCount is also used by glue_get_technology to count the elements in the glueClassList array. The array is used to initialize the class component of the WIDG structure (see Appendix A.5.1.3 for more details).

Finally, the variable glueSpecialProcCount counts the elements in the array glueSpecialProcList. This array is used to list all of the special routines declared in the Glue description, namely the check routines and the routines declared with the +procedure modifier in the equivalence type specifications (see Appendix A.2.4 for more details).

A.4 Running Glue

The general format for running the Glue compiler on a Glue description is:

```
glue [switches] file[.gl]
```

When this command is issued, the named file will be processed by Glue and the appropriate files will be generated. The file to be processed is assumed to have the extension .gl, so it is not necessary to specify a file extension. If an extension is specified, it will be used (so that although the default extension is .gl, other extensions may be used by the toolkit integrator). The list of flags that Glue recognizes are:

-C
: This flag is passed to the Glue preprocessor (/lib/cpp), and causes it to keep comments in the preprocessed output. Ordinarily, comments are stripped by the preprocessor, since they are ignored by Glue.

-D
: This flag is passed to the Glue preprocessor (/lib/cpp), and allows a preprocessor constant to be defined on the command line. An example of the use of this flag is:

  -DTHING

  which is the same as putting #define THING in the Glue source file, and:

  -DOTHER=5

  which is the same as putting #define OTHER 5 in the Glue source file.

`-E` This flag causes the Glue preprocessor (`/lib/cpp`) to be run without subsequently invoking the Glue compiler. This flag is useful for manually checking preprocessor definitions and expansion of macros.

`-I` This flag is passed to the Glue preprocessor (`/lib/cpp`), and allows the user to specify an alternate path to be searched when looking for include files. An example of the use of this flag is:

```
-I../include -I../../data
```

By default, the Glue preprocessor will look in the directories listed in the environment variable SERPENT_DATA_PATH, in the current directory, and in /usr/include.

`-k` By default, the Glue compiler creates a file in `/tmp` to contain the intermediate file produced by the preprocessor. Once the Glue compiler is finished, this intermediate file is automatically deleted. The `-k` flag causes the intermediate file to be retained. The name of the file is printed on the toolkit integrator's terminal.

`-M` By default, the Glue compiler will generate only two output files: the toolkit file and the Saddle file (see Appendix A.3 for details on the Glue-generated files). This flag causes the Glue compiler to generate two additional files: the methods file and the bindings file.

WARNING: The methods file is a C skeleton which `#includes` files that are intended to be filled in by the toolkit integrator. The toolkit integrator should not edit the generated methods file, only the `#included` files. **If the generated files are modified by the toolkit integrator, running Glue with the -M flag will overwrite these files.**

`-U` This flag is passed to the Glue preprocessor (`/lib/cpp`), causing it to delete any initial definition of a preprocessor variable. An example of the use of this flag is:

```
-UTHING
```

which would delete any initial definition of the preprocessor variable THING.

`-v` This flag sets verbose mode, which prints status information as the Glue compiler does its work.

`-w` This flag causes some classes of warning messages to be suppressed (specifically, messages warning about widget description components that are absent, but which Glue feels should be present). If this flag is used, the Glue compiler will not print individual messages, but will only print a summary of the number of messages that were suppressed at the end of the compilation.

A.5 Interfacing to Glue

One of the files generated by Glue is toolkit description. This description file can be used by an application to determine what features are available in a toolkit. Currently, the two application programs that make use of this description are Six and the Serpent dialogue editor. The format of the toolkit file will *not* be discussed here. Rather, what is provided is a collection of C routines that can be linked into the application program to read the toolkit. A brief description of each of the routines and data structures is provided here.

The reader is assumed to have a knowledge of the basics of the structure of the Serpent libraries, including the `ALIST` package. For full details on the workings of these routines, refer to the code in the Glue source directory.

A.5.1 Data Structures

This section describes the data structures provided for in the Glue toolkit interface.

A.5.1.1 NameType

The `NameType` enumeration, shown in Example A-8, provides a switch to distinguish whether to use toolkit names or Serpent names in the routines `glue_get_attribute`, `glue_alist_all_attributes`, and `glue_alist_defaulted_attributes`. Both name types are provided in the Glue description file; the user must distinguish between the two sets of names.

```
typedef enum {
    TECH_NAMES  = 0,
    SERP_NAMES  = 1
    } NameType;
```

Example A-8 The `name_type` Enumerated Type

A.5.1.2 TECH

The `TECH` structure, shown in Example A-9, is the main mechanism to get information about the various toolkits in the Serpent suite. A copy of this structure for a particular toolkit is acquired via the `glue_get_technology` routine (see Appendix A.5.2.1). The caller may read (but not modify) the list of widgets contained in the `TECH` structure with impunity; however, it is expected that the caller will use the faster routines provided in the later sections rather than using a linear search algorithm.

Warning: **The `hook` structure is for Glue internal use only.**

```
typedef struct {
    char *description;
    char *mailbox;
    char *ill_file;
    GT_TYPE gt_last;
    ALIST widgets;
    struct hook *hook;
    } TECH;
```

Example A-9 The TECH Structure

The components of the TECH structure are:

description	From global variable in the Glue file, the description of the toolkit.
mailbox	The name of mailbox for the toolkit.
ill_file	The name of .ill file for the toolkit.
gt_last	The value of the last entry in the GT_TYPE enumerated type when this file was written.
widgets	An ALIST (of type WIDG *) of all the widgets in the toolkit.
hook	**Warning: For internal use only**.

A.5.1.3 WIDG

WIDG structure, shown in Example A-10, defines the way in which widgets are stored by Glue access routines. The caller may read (not write) the attributes and methods ALISTs here with impunity; however, it is expected that the caller will use the faster routines provided in the later sections rather than using a linear search algorithm.

```
Typedef struct {
    char *name;
    WidgetClass class;
    ALIST attributes;
    ALIST methods;
    char *description;
    char *userdef;
    PFV check_rtn;
    WidgetType widget_type;
    TECH *technology;
    short idx;
    } WIDG;
```

Example A-10 The WIDG Structure

The contents of the WIDG structure are:

name	The name of the widget.
class	Widget class for X access.
attributes	An ALIST (of structure ATTR *) of attributes for the widget.
methods	ALIST (structure METH *) of methods for the widget.
description	Description of the widget.
userdef	From same parameter with the same name in the Glue description.
check_rtn	The routine to be used to check the initial values of the widget's attributes. The type PFV is ''pointer to function returning void.''
widget_type	The type of the widget for X access.
technology	A pointer back to the technology/toolkit structure.
idx	**Warning: For internal use only.**

A.5.1.4 ATTR

The ATTR structure, shown in Example A-11, defines the way in which attributes are stored by Glue access routines.

```
typedef struct {
    char *t_name;
    char *s_name;
    GT_TYPE x_type;
    idd_data_types s_type;
    ValueType val_type;
    union {
        int i; double r; string s; caddr_t c;
        } value;
    short access;
    char *description;
    char *userdef;
    short idx;
    } ATTR;
```

Example A-11 The ATTR Structure

The contents of the ATTR structure are:

t_name	The name of the attribute in the toolkit domain.
s_name	The name of the attribute in the Serpent domain.
x_type	C type of the attribute (i.e., the X type).
s_type	Serpent type of the attribute.

`val_type`	The type of value this attribute contains (i.e., none, toolkit default, or Serpent default).
`value`	Default value of the attribute; its type is specified by `val_type`.
`access`	A bitmap of the types of access allowed on attribute.
`description`	Description of the attribute.
`userdef`	From same parameter with the same name in the Glue description.
`idx`	**Warning: For internal use only**.

A.5.1.4.1 METH

The `METH` structure, shown in Example A-12, defines the way in which methods are stored by Glue access routines.

```
typedef struct {
    char *s_name;
    char *c_name;
    int c_index;
    ALIST parms;
    char *description;
    char *userdef;
    } METH;
```

Example A-12 The METH Structure

The contents of the `METH` structure are:

`s_name`	The name of method in Serpent domain.
`c_name`	The name of the associated C routine.
`c_index`	The index of the address of the associated C routine in the `XtActionsRec` array (contains 0 if toolkit is not linked with generated methods file).
`parms`	An `ALIST` (of structure `ATTR` *) of parameters to the method.
`description`	A description of the method.
`userdef`	From same parameter with the same name in the Glue description.

A.5.2 Interface Routines

This section describes the routine names and parameters provided in the Glue toolkit interface. Although the data structures can often be read directly by the application program using the toolkit interface, it is recommended that these routines be used because they have been optimized for speed of access.

A.5.2.1 glue_get_technology

Given a toolkit name, opens the appropriate toolkit data file, initializes a `TECH` structure, and returns a pointer to the structure (similar to the way `fopen` returns a `FILE` *).

The name of the toolkit will typically be an unqualified file name. If an extension is given, it will be used; otherwise, the file extension is assumed to be `.tx`. If the filename is given as an absolute path name, that file will be used; otherwise, Serpent looks for the file in the directories specified in the environment variable `SERPENT_DATA_PATH`.

```
TECH *glue_get_technology(name)
char *name;
```

A.5.2.2 glue_alist_widgets

Given a toolkit, this routine returns an `ALIST` of all of the names of widgets associated with it. The widgets themselves are stored using the `WIDG` structure (shown in Section A.5.1.3), and can be retrieved using other routines in this package.

```
ALIST glue_alist_widgets(technology)
TECH *technology;
```

A.5.2.3 glue_get_widget

Given a toolkit and a widget name, returns a pointer to the `WIDG` structure associated with it. The caller should use this routine to access the class and description components, and should use the `glue_get_method` and `glue_get_attribute` routines to inspect the contents of the `METH` and `ATTR ALIST`s.

```
WIDG *glue_get_widget(technology, widget_name)
TECH *technology;
char *widget_name;
```

A.5.2.4 glue_alist_all_attributes

Given a toolkit and a widget name, returns an `ALIST` of all of the names of attributes associated with the widget. The attributes themselves are stored using the `ATTR` structure (shown in Appendix A.5.1.4) and can be retrieved using other routines in this package.

```
ALIST glue_alist_all_attributes(technology,widget_name, t_or_s)
TECH *technology;
char *widget_name;
NameType t_or_s;
```

A.5.2.5 glue_alist_defaulted_attributes

Given a toolkit and a widget name, returns an `ALIST` of all of the names of attributes associated with the widget that have a Serpent-specified default value.

```
ALIST glue_alist_defaulted_attributes(technology, widget_name,
                                      t_or_s)
TECH *technology;
char *widget_name;
NameType t_or_s;
```

A.5.2.6 glue_get_attribute

Given a widget and an attribute name, returns a pointer to the `ATTR` structure that corresponds to the named attribute in the specified widget.

```
ATTR *glue_get_attribute(widget, attr_name, t_or_s)
WIDG *widget;
char *attr_name;
NameType t_or_s;
```

A.5.2.7 glue_alist_methods

Given a toolkit and a widget name, returns an `ALIST` of all of the names of methods associated with the widget. The methods themselves are stored using the `METH` structure (described in Appendix A.5.1.4.1) and can be gotten using other routines in this package.

```
ALIST glue_ package.
_methods(technology, widget_name)
TECH *technology;
char *widget_name;
```

A.5.2.8 glue_get_method

Given a widget and a method name, returns a pointer to the `METH` structure that corresponds to the named method in the specified widget.

```
METH *glue_get_method(widget, meth_name)
WIDG *widget;
char *meth_name;
```

A.5.2.9 glue_get_attr_default

Given a widget and attribute name, returns the default value associated with the attribute. Recall from Appendix A.2.5.1, only one of the Serpent or toolkit default values can be present.

```
idd_buffer_type *glue_get_attr_default (widget, attr_name,
t_or_s)
WIDG *widget;
char *attr_name;
NameType t_or_s;
```

Appendix B BNF of Glue

The BNF used in this document is a modified Backus-Naur form. The differences between it and normal BNF are:

- All literal tokens are surrounded by double quotes. Although literal token are shown in lower case, Glue is case insensitive.
- The notation [`thing`] indicates an optional instance of `thing`.
- The notation {`thing`}`sep` indicates one or more instances of `thing`, with each instance terminated by the character `sep`. If `sep` is not specified, then the separator is white space.

```
glue            : {mapping};

mapping         : global_assign
                | type_equiv
                | widget_defn

global_assign: C_TOKEN "=" C_STRING

type_equiv: C_TOKEN ":" serpent_type

serpent_type: C_TOKEN [range] [procedure]

range           : "[" INTEGER "]"

procedure: "+" "procedure"

widget_defn: C_TOKEN "{" {widget_part}; "}"

widget_part: "class" "=" C_TOKEN
                | "include_file" "=" C_STRING
                | "check_routine" "=" C_TOKEN
                | "description" "=" C_STRING
                | "userdef" "=" C_STRING
                | "shell_widget" "=" BOOLEAN
                | "attributes" "=" "{" {attribute}; "}"
                | "methods" "=" "{" {method}; "}"

attribute: C_STRING ":" "{" {attr_part}; "}"

attr_part: "serpent_name" "=" C_STRING
                | "description" "=" C_STRING
                | "userdef" "=" C_STRING
                | "x_type" "=" C_TOKEN
```

```
                | "serpent_default" "=" default_value
                | "technology_default" "=" default_value
                | "access" "=" {access},

default_value: C_STRING
                | C_TOKEN
                | REAL
                | INTEGER
                | BOOLEAN

access          : C_TOKEN
                | "!" C_TOKEN

method          : C_TOKEN ":" "{" {method_part}; "}"

method_part: "parameters" "=" {C_STRING},
                | "description" "=" C_STRING
                | "userdef" "=" C_STRING
                | "c_routine" "=" C_TOKEN
```

Appendix C Glue Error Messages

This section lists in alphabetical order all of the error messages that can be generated by Glue, along with a detailed explanation of the message.

ACCESS denies all access!
The specified access type does not allow the attribute to be accessed. You must allow at least one of `create, get,` or `set`, and at least one of `serpent` and `technology`.

Access type '!all' is meaningless
The access type '!all' indicates that the attribute cannot be accessed by either the toolkit or Serpent. This makes no sense from the perspective of toolkit integration.

Arg list too long
The number of command line arguments passed to `Glue` is too large. Either specify the command line again, or contact the Glue compiler maintainer.

Attribute '%s' not a "string" (missing #define?)
All attribute names must be specified as a quoted string. Usually, these strings are `#define`d in an include file. Either you have forgotten to quote the attribute name specified, or the `#define` that matches the name you used is missing.

Attribute '%s' of type id cannot have a default value
It is impossible for the Glue compiler to assign a default value to an attribute of type `id`, since the values are not known until dialogue execution time.

Attribute '%s' must have a X_TYPE specified
The Glue compiler did not find an X type specification for the named attribute. All attributes must specify the X type of the attribute for referencing by Six and other applications.

Cannot find source file "%s"
The Glue source file name should be the last argument in the command line. The Glue compiler could not find the specified file name to process it.

Cannot open output file '%s'
The named output file could not be created by the Glue compiler. Either you do not have write permission to the current directory, or a file with the specified name already exists and you do not have write access to it.

Consistency error in gen_technology()!
This is an internal consistency error. Please contact the `Glue` maintainer.

?Could not delete temp file "%s"!
The Glue compiler could not delete one of its temporary files. This is a warning only, although the cause should be investigated with the Glue maintainer.

Duplicate ACCESS in attribute '%s' in widget '%s'
The Glue compiler encountered a duplicate `access` in the named attribute/widget pair. Only a single instance of the `access` attribute part is allowed in an attribute.

Duplicate attr '%s' in widget '%s'
The Glue compiler encountered two attributes with the same name in one widget. Only one attribute with any name may appear in a widget.

Duplicate C_ROUTINE in method '%s' in widget '%s'
The Glue compiler encountered two C routine specifications in the named method. Only one C routine may be associated with a method.

Duplicate CHECK_ROUTINE in widget '%s'
The Glue compiler encountered a duplicate `check_routine` in the named widget. Only a single instance of the `check_routine` widget part is allowed in a widget.

Duplicate CLASS in widget '%s'
The Glue compiler encountered a duplicate `class` in the named widget. Only a single instance of the `class` widget part is allowed in a widget.

Duplicate DESCRIPTION in attribute '%s' in widget '%s'
The Glue compiler encountered a duplicate `description` in the named attribute/widget pair. Only a single instance of the `description` attribute part is allowed in an attribute.

Duplicate DESCRIPTION in method '%s' in widget '%s'
The Glue compiler encountered two description specifications in the named method. Only one description may be associated with a method.

Duplicate DESCRIPTION in widget '%s'
The Glue compiler encountered two `description` specifications in the named method. Only one `description` may be associated with a method.

Duplicate INCLUDE_NAME in widget '%s'
The Glue compiler encountered a duplicate `include_name` in the named widget. Only a single instance of the `include_name` widget part is allowed in a widget.

Duplicate method '%s' in widget '%s'
The Glue compiler encountered two methods with the same name in one widget. Only one method with any name may appear in a widget.

Duplicate PARAMETER list in method '%s' in widget '%s'
The Glue compiler encountered two parameter list specifications in the named method. Only one parameter list may be associated with a method.

Duplicate Saddle command
The Glue compiler encountered two Saddle commands in the same Glue description. Only a single Saddle command is permitted in a Glue file.

Duplicate SERPENT_DEFAULT in attribute '%s' in widget '%s'
The Glue compiler encountered a duplicate `serpent_default` in the named attribute/widget pair. Only a single instance of the `serpent_default` attribute part is allowed in an attribute.

Duplicate SERPENT_NAME in attribute '%s' in widget '%s'
The Glue compiler encountered a duplicate `serpent_name` in the named attribute/widget pair. Only a single instance of the `serpent_name` attribute part is allowed in an attribute.

Duplicate WIDGET_TYPE in widget '%s'
The Glue compiler encountered a duplicate `widget_type` in the named widget. Only a single instance of the `widget_type` widget part is allowed in a widget.

Duplicate TECHNOLOGY_DEFAULT in attribute '%s' in widget '%s'
The Glue compiler encountered a duplicate `technology_default` in the named attribute/ widget pair. Only a single instance of the `technology_default` attribute part is allowed in an attribute.

Duplicate toolkit description
The Glue compiler detected two toolkit descriptions in a Glue file. Only a single toolkit description is permitted in a Glue file.

Duplicate use of ACCESS type %s
The Glue compiler detected a duplicate use of the named `access` type. Each access type may be specified only once for a given access specification.

Duplicate use of ACCESS type %s (use "all" by itself)
The Glue compiler encountered the access type "all" (which means all access types) in conjunction with other access types. Using "all" in conjunction with any other access type is redundant.

Duplicate USERDEF in attribute '%s' in widget '%s'
The Glue compiler encountered a duplicate `userdef` in the named widget pair. Only a single instance of the `userdef` widget part is allowed in an widget.

Duplicate USERDEF in method '%s' in widget '%s'
The Glue compiler encountered two user definition specifications in the named method. Only one user definition may be associated with a method.

Duplicate USERDEF in widget '%s'
The Glue compiler encountered two user definition specifications in the named method. Only one user definition may be associated with a method.

Duplicate widget type '%s'
The Glue compiler encountered two widgets with the same name. Only one widget with any name may appear in a Glue description.

Duplicate X_TYPE in attribute '%s' in widget '%s'
The Glue compiler encountered a duplicate `x_type` in the named attribute/widget pair. Only a single instance of the `x_type` attribute part is allowed in an attribute.

EOF encountered in string begun on line %d
The Glue compiler encountered the end of the input file before it found the close of the string begun on the cited line. A closing quote may have been omitted, or the closing quote had a backslash in front of it.

Equivalence type '%s' cannot have an associated procedure
Only equivalence types of type `integer` can have an associated procedure. For all other types, specifying an associated procedure is an error.

Equivalence type '%s' needs a size modifier
The named equivalence type needs to have a specified size. Serpent type `string` must have a size declared so the Glue compiler will know how many bytes to allocate to the string.

Equivalence type '%s' not allowed. Using 'integer'
An illegal equivalence type was detected on the right-hand side of an equivalence declaration. The legal equivalence types are essentially the Serpent data types. These are `boolean, buffer, id, integer, real`, and `string`. No other types (including undefined) may be used as the right-hand side of an equivalence declaration.

ERROR: Input file will be overwritten by output file
The name of the input file is the same as one of the files that will be generated by the Glue compiler. The compiler does not accept duplication; you must rename your input file.

ERROR: Number of parameters for routine '%s' for method '%s' in widget '%s' differs from previous uses of that method name (initially defined in method '%s' of widget '%s').
The number of parameters that is passed into a method routine must be the same, regardless of the widget from which the routine is invoked. You have declared a method routine with a different number of parameters than was specified elsewhere in the Glue description. Either change the parameter count or use the `c_routine` method component to specify a different routine name.

ERROR: Parameter %d ('%s') for routine '%s' for method '%s' in widget '%s' is not used in previous use of that routine (initially defined in method '%s' of widget '%s')
The names of parameters that are passed into a method routine must be the same, regardless of the widget from which the routine is invoked. You have declared a method routine with different parameter names than those specified elsewhere in the Glue description. Either change the parameter names or use the `c_routine` method component to specify a different routine name.

ERROR: Parameter %d ('%s') of method '%s' in widget '%s' is not the name of any attribute in the widget.
All parameters to methods must be the Serpent names of attributes of the widget for which the method is associated. You have either specified a non-attribute name or (if you get the additional message **HINT: Use the Serpent attribute name ('%s'), not the X name**) you have specified the X name of the attribute.

?execve() for /lib/cpp failed
The Glue compiler could not find the C preprocessor. Try executing the command again; if it fails, contact the Glue maintainer.

Expecting one of:
The Glue compiler detected a syntax error in the Glue description file. The list of legal tokens, one of which Glue expects in the input stream (but which was not encountered), are listed.

Expecting '}' - possible missing separator (',' or ';')
The Glue compiler encountered an unexpected token when it anticipated finding a close brace. This syntax error is usually caused by a missing separator (i.e., a comma or semicolon) at the point indicated by the error message.

Extra args after filename "%s"
The Glue source file name should be the last argument on the command line. The Glue compiler detected additional arguments after the Glue source file name.

?fork() for /lib/cpp failed
The Glue compiler could not execute the C preprocessor. Try executing the command again; if it fails, contact the Glue maintainer.

Illegal character in text:
The Glue compiler detected an illegal character in the Glue description. The legal characters are upper and lower case letters, numbers, spaces, tabs, newlines, comma, and the characters ".", ";", ":", ",", "[", "]", "{", "}", "(", ")", "=", "!", "+", and "#".

Invalid widget_type '%s' in widget '%s'
The legal widget types are `shell, override, widget,` and `none`. Any other type is illegal.

Missing close quote
The Glue compiler encountered the end of the input line before a string was terminated. If you wish to have a multi-line string, you must include the newline character as the digraph \n. If you have a string that you wish to be stored as a single line, but which may exceed a single line in your Glue source code, end each line that is in the middle of the string with the backslash character.

Missing Glue filename
The Glue source file name should be the last argument on the command line. The Glue compiler did not find a file name to process.

Mixed mode ACCESS must list all types!
When an access list is specified that lists both access allowances and denials (i.e., the list contains both names and negated names, as in: `get, !set`), all types must be listed. If access allowances are specified, only those accesses are permitted; if access denials are specified, only those accesses are disallowed; if both types are specified, you must list all access types.

Size modifier illegal for equivalence type '%s'
Only equivalence types of type `string` can have an associated size. For all other types, specifying an associated size is an error.

Note: Suppressed %d 'no class value' warning messages
When the Glue compiler encounters a widget without a class, it issues a warning message. If, however, the toolkit integrator specifies the `-w` command line argument, these messages will be suppressed. This note is printed to indicate how many of these messages were not printed.

Note: Suppressed %d 'no default value' warning messages
When the Glue compiler encounters a non-toolkit attribute, it issues a warning message. If, however, the toolkit integrator specifies the `-w` command line argument, these messages are suppressed. This note is printed to indicate how many of these messages were not printed.

Note: Suppressed %d 'no include file value' warning messages
When the Glue compiler encounters a widget without an include file, it will issue a warning message. If, however, the toolkit integrator specifies the -w command line argument, these messages will be suppressed. This note is printed to indicate how many of these messages were not printed.

Syntax error - is this a legal statement?
This error message indicates that the Glue compiler is confused by the syntax that it has scanned. Check the Glue source file for illegal syntax.

Too many includes in SERPENT_DATA_PATH
The number of components in the environment variable `SERPENT_DATA_PATH` is too large. Either specify the environment variable again, or contact the Glue maintainer.

Type '%s' has no Serpent equivalent
The toolkit type used for an attribute does not have a specified Serpent equivalent. This means that there is no equivalence statement earlier in the Glue description that declares an equivalence type for this toolkit type.

Unknown ACCESS type
The Glue compiler encountered an illegal access name. The legal names are: `all`, `create`, `get`, `serpent`, `set`, and `technology`. All access names (except `all`) may be complemented (i.e., `!create`, etc.).

Unknown attribute component
The Glue compiler encountered an unknown attribute component. The legal attribute components are: `access`, `description`, `serpent_default`, `serpent_name`, `userdef`, and `x_type`.

Unknown equivalence modifier
Currently, the only legal equivalence modifier is `+procedure`. Any other modifier is illegal.

Unknown equivalence type '%s'. Using 'integer'.
An unknown equivalence type was detected on the right-hand side of an equivalence declaration. The legal equivalence types are essentially the Serpent data types. These are `boolean`, `buffer`, `id`, `integer`, `real`, and `string`. No other types may be used as the right-hand side of an equivalence declaration.

Unknown flag %s
An unknown command line argument was given to the Glue compiler. The legal flags are: -C, -D, -E, -k, -I, -M, -U, -v, and -w. See Appendix A.4 for more details.

?Unknown idd_type in gen_Saddle()
This is an internal consistency error. Please contact the Glue maintainer.

?Unknown idd_type in gen_technology()
This is an internal consistency error. Please contact the Glue maintainer.

?Unknown idd_type in print_method_names()
This is an internal consistency error. Please contact the Glue maintainer.

Unknown escape character ignored
The list of legal escape characters in the Glue compiler is presented in Appendix A.2.1.8. Any other character is illegal.

Unknown keyword '%s'
The named global variable is not currently known to the Glue compiler. Check for a typographical error.

Unknown method component
The Glue compiler encountered an unknown method component. The legal method components are: `c_routine`, `description, parameters`, and `userdef`.

Unknown widget component
The Glue compiler encountered an unknown widget component. The legal component names are: `attributes, check_routine, class, description, include_file, methods, widget_type`, and `userdef`.

Usage: %s {<cpp args>} -E -M -k -v <file>
If the toolkit integrator incorrectly specifies a command line, this message is issued as a reminder of the correct command line syntax.

Warning: String contains the null char
In Glue and the rest of Serpent, strings are terminated with the null character. Your string contains the null character, so it will be terminated early. Having a null character in a string is almost certainly a mistake.

Warning: Redeclaration of equivalence type '%s'
The toolkit name specified on the left-hand side of an equivalence statement was previously declared in the current Glue source file. This message is only a warning–that is, the redeclaration will take place as requested. Only those attributes of the specified type after the redeclaration will be affected. It is probably better to declare a new toolkit type than to overload a toolkit name by redeclaration.

Warning: String too long
The internal limit of Glue string lengths has been exceeded. Either shorten the string, or contact the Glue compiler maintainer. The default maximum string length is currently 3072 characters, so it is unlikely that you will see this message with a correct Glue source file.

Warning: Technology default for '%s' in '%s' will be overwritten by Serpent default
When an attribute has specified both a Serpent default value and a toolkit default value, the Serpent default takes precedence. This message warns you that the toolkit default value is being ignored.

Warning: Technology type '%s' is unknown. You may need to rebuild Glue
The toolkit type specified on the left-hand side of an equivalence statement is not known by the current version of the Glue compiler. Chances are that you have made a typographical error in the name. If, however, the name is correct, then the Glue compiler needs to be updated to be advised about the new toolkit name.

WARNING! The names of X classes used in %s.gl are different from that in %s.tx. You need to relink the technology driver or you will have erroneous behavior.
The previous version of the toolkit file that Glue generated has a different set of classes than the newly generated version. The toolkit driver program (Six) expects some consistency between versions, and must be relinked if correct behavior is to be expected.

WARNING! The names of X procedures used in %s.gl are different from that in %s.tx. You need to relink the toolkit driver or you will have erroneous behavior.
The previous version of the toolkit file that Glue generated has a different set of names of procedures than the newly generated version. The toolkit driver program (Six) expects some consistency between versions, and must be relinked if correct behavior is to be expected.

WARNING! The number of X classes used in %s.gl is different from that in %s.tx. You need to relink the toolkit driver or you will have erroneous behavior.
The previous version of the toolkit file that Glue generated has a different number of classes than the newly generated version. The toolkit driver program (Six) expects some consistency between versions, and must be relinked if correct behavior is to be expected.

WARNING! The number of X procedures used in %s.gl is different from that in %s.tx. You need to relink the toolkit driver or you will have erroneous behavior.
The previous version of the toolkit file that Glue generated has a different number of classes than does the newly generated version. The toolkit driver program (Six) expects some consistency between versions and must be relinked if correct behavior is to be expected.

WARNING: Widget '%s' should have a CLASS listed
The Glue compiler did not find an class specification for the named widget. All widgets should specify a class that identifies the widget to the X server. Failure to do so may result in erroneous behavior by the toolkit driver program (Six).

WARNING: Widget '%s' should have a INCLUDE_FILE listed
The Glue compiler did not find an include file specification for the named widget. All widgets should specify an include file that defines the C structure layout of the widget.

X_TYPE and default value type mismatch for attribute '%s'
The type of the specified default value does not match the type defined by the X_TYPE attribute part. The types must either match, or the default value must be elided for, the Glue compilation to succeed.

Appendix D Six Overview

Together with Glue, Six helps with the integration of Xt-based toolkits into Serpent. Glue is a language for defining toolkit widgets; Six is a generic Serpent-to-Xt driver that converts the runtime tables produced by Glue into calls to Xt. These tables can also be used by the Serpent dialogue editor to automate the process of adding widgets to a toolkit-to-Serpent binding.

D.1 What is Six?

Six stands for Serpent Interface to X. It is an attempt to create an interface between Serpent and any X Toolkit-based widget set (such as the Motif and Athena widget sets). The aim of this interface is to be general, customizable, and easy to build. Widgets are described to Six through the Glue language (see Appendix A for details).

Due to the ability to describe the characteristics of widgets independently of any implementation through Glue, Six is not tied to a particular widget set implementation. It is tied to the X Toolkit as a basis for widget sets, however.

Six treats widgets as a set of attributes, each of which has a well-known type. Six, then, is primarily concerned with attribute types rather than particular attributes. All attributes of a given type are treated exactly alike. This generalization is the source of Six's power–it allows new widgets or widget sets to be integrated with a minimum of effort, since most widgets contain attributes of the same types: Position, Dimension, Boolean, Pixel, Pixmap and so on. These attributes will be handled in standard ways by existing code within Six. Since most widgets can be described in terms of a fairly small set of attributes, the process of adding new widgets is, in most cases, simply a process of describing these widgets to Six through the Glue language.

The widget integrator only has to modify Six if he or she wants to add a widget which contains new types. The bulk of this section will be devoted to explaining how a user can add new widgets and especially new attribute types to Six.

Because it is impossible to characterize all of a user's interactions with widgets as attribute setting, "hooks" were added to Six that allow the user to write special-purpose code, which is executed when a widget is created or modified. The combination of a general purpose widget integration tool and the ability to introduce arbitrary computation has been, in our experience, sufficiently powerful to integrate any widget set.

D.2 Using Six

Six is a general-purpose X Toolkit-based toolkit driver. A toolkit driver is defined as a program that reads a toolkit description (in the form of a compiled Glue file) and then accepts and dispatches interrupts from the X Toolkit and Serpent. Six's main purpose is to stand between the X Toolkit and Serpent events, translating between them. Widget set interfaces that come with Serpent are executed automatically by the Serpent command.

Using Six to integrate new widgets or a new widget set, however, requires an understanding of the inner workings of Six, particularly how and why Six transforms Serpent data types into X Toolkit data types. *Why* is easy: this translation is necessary because X-based toolkits make use of a much greater assortment of data types than Serpent supports, and so, for instance, an integer received by Six may be mapped into Cardinal, int, Dimension, short, etc. before being sent to X. *How* Serpent handles the transformation of data types and how users can add their own transformations is more complex. The answer involves Glue include files, changes to Glue data files, changes to Glue, and changes to Six.

D.3 Adding New Types to Six

Adding new attribute types to Six requires defining them first in Glue. This is because Glue checks input files to ensure that they contain only validly specified attributes and then converts these attribute specifications into an internal format that is more efficient for computation. This internal format is stored in files with a .tx suffix, hereinafter referred to simply as `TX` files. (These files are described in detail in Appendix A.)

The following descriptions are intended to be fairly specific. They refer to particular files in the Six and Glue subdirectories of Serpent, and to particular contents of those files. In addition, an explanation of the contents is given, so that an interested user could conceivably create new toolkit drivers with new files, following the reasoning given here.

D.3.1 Changes to Glue Include Files

Associated with the Glue parser is a list of all valid attribute types. This list is a way of assigning each attribute type a unique identifier; the list is available for use by the Glue parser, Six, and other programs (such as the dialogue editor) and must be updated if a new attribute type is added.

In the current implementation, this list is kept in the file `gt_type.h`, in the glue/include directory. Each attribute type is assigned a name which consists of the data type preceded by `GT_`, so, for instance, XtTranslations becomes `GT_XtTranslations`, and is #define'd to some unique integer value, as exemplified in the following:

```
#define GT_XtTranslations 27
```

In addition, this file contains version numbers to ensure that only the correct version of this file is used to build Glue and the toolkit drivers. Glue, `check_intf` (a program that checks the format of a TX file), and Six all check these version numbers to ensure that the program is not being run with an out-of-date include file. Version numbers are of the form:

```
#define  GT_LAST_V6 46
```

When new attribute types are added to Glue, the version number must be updated in order for the consistency checking to work. Details of how to do this can be found in `the file gt_type.h`.

D.3.2 Changes to Glue Data Files

The Glue parser is built to be independent of the particulars of toolkit implementation details. In particular, it does not contain any information about the data types being used. This information must be specified by the user, in the form of equivalence types (more information about equivalence types may be found in Appendix A). These types equate Serpent data types with toolkit-specific types. An example of an equivalence type is:

```
Accelerators    : buffer;
```

This means that the Xt attribute type `Accelerators` is implemented in Serpent as the data type `buffer`.

In the current implementation, the equivalence types are stored in the file `glue_eqv`.gl, in the six/src directory. If new types are needed to describe new widget attributes, they must be added to this file. Exactly how these types are converted from Serpent types to Xt types and back to Serpent types is discussed in Section D.3.4.

D.3.3 Changes to Glue

The Glue parser checks attribute definitions to ensure that only valid attributes are specified by the user. When a new attribute is added, the parser must be updated to reflect this.

Glue is currently implemented with YACC/Lex. In order to add a new type to the parser, the YACC file `gay.y` (found in the glue/src directory) must be edited, and the new attribute type must be added to the `tech_map` structure. `Tech_map` is actually an array of structures, where each array element contains the name of the new attribute type, as a string, and the equivalent internal type definition, as defined in `gt_type.h` (see the section on changes to Glue include files [Appendix D.3.1] for more information on the structure of this file). An example of an entry in the `tech_map` array is:

```
{ "Dimension", GT_Dimension }
```

Furthermore, if a new attribute type has been added to Glue, there must have been a new version number added to the include file gt_type.h. If this is the case, the version number check in `gay.y` must also be updated to indicate that the two files are synchronized. For instance, the check in Glue for V6 would be in the form:

```
if (GT_LAST > (GT_LAST_V6 + 1)) {  fputs ("Warning! ....",
stderr) ;
```

When a new version number is added to `gt_type.h`, this check will have to compare `GT_LAST` to `GT_LAST_V7`.

Finally, `check_intf` is a utility included with Glue. This utility is used to print the contents of a TX file (which is stored in a format that is not readable by humans) in order to inspect or validate the results of the parsing which Glue has performed. To be able to accurately print out the contents of a Glue-produced TX file, `check_intf` must be updated to reflect the new type. All of the valid types are contained in a switch. Each case of the switch simply prints out, in a form that is readable by humans, the type of the current attribute, for example:

```
case GT_Widget:  fputs ("Widget;\n", stdout);       break;
```

If a new type is added, this switch statement should be similarly updated. It is not necessary for the proper working of the system to keep `check_intf` up-to-date, but experience shows that it is a useful tool and users are strongly encouraged to keep it current if they modify Glue. In fact, if the file `gt_type.h` changes, `check_intf` must be updated or it will no longer work because it checks version numbers, as described above for gay.y, which must be similarly updated.

D.3.4 Changes to Six

Six consists of a core function that communicates with the X Toolkit and Serpent, reads Glue TX files, and creates, modifies, and deletes widgets. In order to be independent of widget sets, all widget-set dependent code resides in other routines, which are linked together with the core module, `six.c`, to create executable programs that are specific to widget sets. Currently, `six.c` is linked with special-purpose code for the Athena and Motif widget sets to produce the executables sat (Six AThena) and smo (Six MOtif).

When adding attributes to Six, the routine `Six_set` must be modified. `Six_set` builds argument lists for widget creation and modification and so must know the type of every attribute. As with `check_intf`, described earlier, all of the existing types are contained in a switch statement in `Six_set`, and any new types must be added to this switch.

Attributes come in many flavors. Some are quite simple, and may be directly entered into the argument list. Others need a significant amount of pre-processing before being acceptable to a widget. Three flavors of attributes will be discussed here: those that require virtually no transformation; those that require simple, readily available transformations; and those that require some ingenuity on the part of the widget integrator.

For example, boolean attributes are quite simple:

```
case GT_Boolean:
if (cur_attr->access & AXS_TECH) {  XtSetArg(args[*num_args],
cur_attr->t_name, *((boolean *) data_ptr));  (*num_args)++; }

if (cur_attr->access & AXS_SERPENT) {  cur_attr->value.i =
*((boolean *) data_ptr) ;  cur_attr->val_type = VAL_CURRENT ; }
break ;
```

What this case does is check to see whether the attribute being currently examined (called `cur_attr`) is accessible to the technology, as indicated by the contents of `cur_attr->access`. If so, the argument list is set to point at `data_ptr` (the location, in memory, of the boolean attribute being currently referenced), appropriately cast. Next, if `cur_attr` is accessible to Serpent, the current value is stored, again appropriately cast, in the attribute's data structure. Finally, the `val_type` field is set to `VAL_CURRENT`, indicating that a current value has been set for this attribute (as compared, for instance, with a default value).

Sometimes, however, the data being passed from Serpent cannot simply be cast to the appropriate type. For instance, X Toolkit translation tables are represented as ascii strings, and must be parsed before being used by a widget. This is accomplished as follows:

```
case GT_XtTranslations:
        ret_string = (char *) uc2_convert(
                  idd_string,
                  idd_buffer,
                  (idd_buffer_type *) data_ptr) ;
        if (cur_attr->access & AXS_TECH) {
             XtSetArg(
                  args[*num_args],
                  cur_attr->t_name,
                  XtParseTranslationTable(ret_string)) ;
             . . .
```

First, the translation table is converted from a buffer (its Serpent-internal data type) to a string, via a call to `uc2_convert`. Then it is set in the argument list via a call to `XtParseTranslationTable`. Since `XtParseTranslationTable` returns the type `XtTranslations`, the result of this routine call may be passed directly to the argument list without further intervention.

Finally, Pixmaps are a very inconvenient data type to manipulate and are not readily amenable to the dialogue model used in Serpent. To allow the dialogue writer some distance from Pixmaps, the Pixmap attribute is treated as the name of a bitmap file. This file is read, stored temporarily, and then converted into Pixmap form, which is the form suitable for a widget's argument list. The code to accomplish this is as follows:

```
case GT_Pixmap:
        ret_string = (char *) uc2_convert(
                  idd_string,
                  idd_buffer,
                  (idd_buffer_type *) data_ptr) ;
        if (cur_attr->access & AXS_TECH) {
             bitmap = sbm_load_bitmap(
                  ret_string,
                  &bm_width,
                  &bm_height,
                  &bm_size);
        if (bitmap) {
             XtSetArg(
                  args[*num_args],
                  cur_attr->t_name,
                  XCreateBitmapFromData(
                         display,
                         RootWindow(display, screen),
                         bitmap,
                         bm_width,
                         bm_height)) ;
             . . .
```

Finally, if a new attribute type has been added to Six, there must have been a new version number added to the include file gt_type.h. If this is the case, the version number check in `six.c` must also be updated, to indicate that the two files are synchronized, as described in Section D.3.3).

D.3.4.1 CheckRoutines

Six provides check routines to accommodate the need of users to write arbitrary code to manipulate widgets. (Not all widget interactions are mediated by argument lists.) These check routines are specified in the toolkit's Glue file. In the current implementation, the routines for the Athena widget set are kept in the file `sac.c` (Six Athena Check) in the six/src directory and the routines for the Motif widget set are kept in the file `smc.c` (Six Motif Check) in the six/src directory. These files are compiled and linked with `six.c` to produce the executables sat and smo.

If new check routines are needed to manipulate widgets added to Six, they should be placed into the appropriate file `(sac.c, smc.c,` or some other file if a new widget set is being integrated).

Each check routine has the following parameters:

```
widget_inst, change, change_list
```

where:

`widget_inst` is a pointer to the current `INST` structure (an `INST` is an internal data structure used to describe an instance of a widget) that contains, among other things, the widget handle from X.

`change` is the type of operation: create or modify (check routines are not called for deletes).

`change_list` is the list of changes to shared data for this widget.

Once you have access to a widget's `INST` structure, you can find out everything you need to know about the widget: its attributes, its parent, its widget type, etc.

Note, however, that the check routine, if it exists, is called both before and after a widget is created.

D.3.4.2 Methods

Glue provides the ability to define Methods for each widget. Methods are used to return information to the dialogue in response to user events: when the user clicks a mouse button on a widget, for example. These events are often reported to Six through the use of Action routines, discussed below. Methods are communicated to the dialogue through the use of shared data. For example, the notify method is sent to the dialogue by the following code (for details on the procedure calls used in this example, see *Serpent: C Application Developer's Guide* and *Serpent: Ada Application Developer's Guide*.)

```
/* Send the translation to the interface */ trans =
isd_start_transaction ();
/* Send the method name */ isd_put_shared_data (trans, inst->id,
inst->template->name,        "method", "notify");
/* Commit the transaction */ isd_commit_transaction (trans);
```

In some cases, a Method is sent to the dialogue in response to an event which directly affects the state of the dialogue (changing the size of one of the widgets in the dialogue, for instance). In these cases, the changed data–the x and y location of the widget–must also be sent to the dialogue:

```
/* Send the transactions to the interface */ trans
=isd_start_transaction ();

/* Send the method name */ isd_put_shared_data (trans, inst->id,
inst->template->name,       "method", "move");

         /* Send the changed attributes */
isd_put_shared_data (trans, inst->id,

inst->template->name,    "x",     &x->value.i);
isd_put_shared_data (trans, inst->id,

inst->template->name,     "y",    &y->value.i);
isd_put_shared_data (trans, inst->id,
inst->template->name,  "horizdistance", &x->value.i);
isd_put_shared_data (trans, inst->id,

inst->template->name, "vertdistance", &y->value.i);

/* Commit the transaction */ isd_commit_transaction (trans);
```

To add a new Method to Six, the Method must first be defined through Glue. This will produce a method definition in the file `<tech>_meth.c`, where `<tech>` is one of sat, smo, or, possibly, a new toolkit which is being integrated using this mechanism.

This file contains definitions for Action, routines that can be called from a widget's translation table in response to some user routines. `Six.c` registers the routines that the user has defined through Glue with the X Toolkit.

Index

U

W

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S
CMU/SEI-91-UG-8

5. MONITORING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-91-UG-8

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	PROJECT NO.	TASK NO	WORK UNIT NO.
63752F	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Serpent: Guide to Adding Toolkits

12. PERSONAL AUTHOR(S)
SEI User Interface Project

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM TO	May 1991	~95

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, toolkits, Glue, Six

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 add toolkits to Serpent. A generic description of how to integrate any toolkit into Serpent is followed by descriptions of two tools. These tools are Glue, a generalized widget integration facility, and Six, a generic Serpent-to-Xt binding driver. Readers of this guide are assumed to have programming experience in C or Ada, and to have read and understood the concepts described in the *Serpent*

(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 JPO)

Overview and *Serpent: System Guide*.

CMU/SEI-91-UG-1	Serpent Overview
CMU/SEI-91-UG-2	Serpent: System Guide
CMU/SEI-91-UG-3	Serpent: Saddle User’s Guide
CMU/SEI-91-UG-4	Serpent: Dialogue Editor User’s Guide
CMU/SEI-91-UG-5	Serpent: Slang Reference Manual
CMU/SEI-91-UG-6	Serpent: C Application Developer’s Guide
CMU/SEI-91-UG-7	Serpent: Ada Application Developer’s Guide
CMU/SEI-91-UG-8	Serpent: Guide to Adding Toolkits
CMU/SEI-91-UG-9	Dialogue Editor User’s Guide