User's Guide

CMU/SEI-91-UG-2
April 1991

Serpent: System Guide

User Interface Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

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

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

John Herman
SEI Joint Program Office

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

This report was prepared for the

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

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

John Herman
SEI Joint Program Office

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

1 Introduction 1

1.1 This Manual 1

1.1.1 Organization 1

1.1.2 Typographical Conventions 1

1.2 Other Serpent Documents 2

1.3 Installing Serpent 4

2 System Configuration 5

2.1 Environment Variables 5

2.2 File Types and Naming Conventions 6

2.3 Layout of Programs and Data Files 7

3 System Components 11

3.1 Saddle Compiler 11

3.2 Slang Compiler and Linker 12

3.3 Dialogue Manager 14

3.4 Toolkit Interface 15

3.4.1 Glue and the Six Interface 16

3.4.2 Other Paths for Toolkit Integration 16

3.5 User Application Program 17

3.6 Running A Dialogue/Application 17

4 Example Dialogue/Application 19

4.1 First Stage of Developing Clock 20

4.1.1 Slang Dialogue 20

4.1.2 Makefile 25

4.2 Second Stage of Developing Clock 25

4.2.1 Slang Dialogue 28

4.2.2 External C Routines 34

4.2.3 Makefile 34

4.3 Third Stage of Developing Clock 35

4.3.1 Saddle Description 35

4.3.2 Slang Dialogue 36

4.3.3 Application Program 42

4.3.4 Makefile 45

4.4 Fourth Stage of Developing Clock 45

4.4.1 Slang Dialogue 46

4.4.2 Makefile 47

List of Figures

Figure 1-1	Serpent Documents	3
Figure 3-1	– Key to Illustrations	11
Figure 3-2	– Saddle Compiler	12
Figure 3-3	– Slang Compiler and Linker	13
Figure 3-4	– Dialogue Manager	14
Figure 3-5	– Glue and Six	16
Figure 4-1	– Picture of Executing Dialogue	19
Figure 4-2	– Initial Clock Dialogue Layout	20
Figure 4-3	– Clock Dialogue, Revised Layout	25
Figure 4-4	– Revised Layout Showing Date Enabled	26
Figure 4-5	– Revised Layout Showing 12-Hour Mode Enabled	27
Figure 4-6	– Revised Layout Showing “Set” Buttons Enabled	28
Figure 4-7	– Alternative Interface for Clock	45

List of Examples

Example 4-1 Slang Dialogue for Sample Clock 25
Example 4-2 Slang Dialogue for the Second Stage in Developing the Sample Clock 34
Example 4-3 Slang Dialogue for the Third Stage in Developing the Sample Clock 42
Example 4-4 Application Program 44
Example 4-5 Slang Dialogue for the Alternative Dialogue 47

1 Introduction

Serpent is a user interface management system (UIMS) being developed at the Software Engineering Institute (SEI). Serpent 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.

1.1 This Manual

Designed as a prelude to other, more detailed Serpent documents, this manual introduces the environment variables used by Serpent, the file naming conventions and expected file types, and how to build a Serpent dialogue/application from scratch. It is not designed as a reference guide for any of the Serpent system components–these can be found in other Serpent documentation. Rather, this document outlines the layout of the Serpent system and, in general, how to use it. It should be read after the *Serpent Overview* and before other Serpent documentation.

1.1.1 Organization

This guide is organized into the following chapters:

- **Introduction**. Presents information about installing Serpent.
- **System Configuration.** Outlines the layout of the Serpent system.
- **System Components**. Describes the interaction between the various components of the Serpent system.
- **Example Dialogue/Application.** Presents a four-stage example.

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, italic statements***

1.2 Other Serpent Documents

The following documents provide information about the Serpent system.

Serpent Overview
Introduces the Serpent system.

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.

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

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

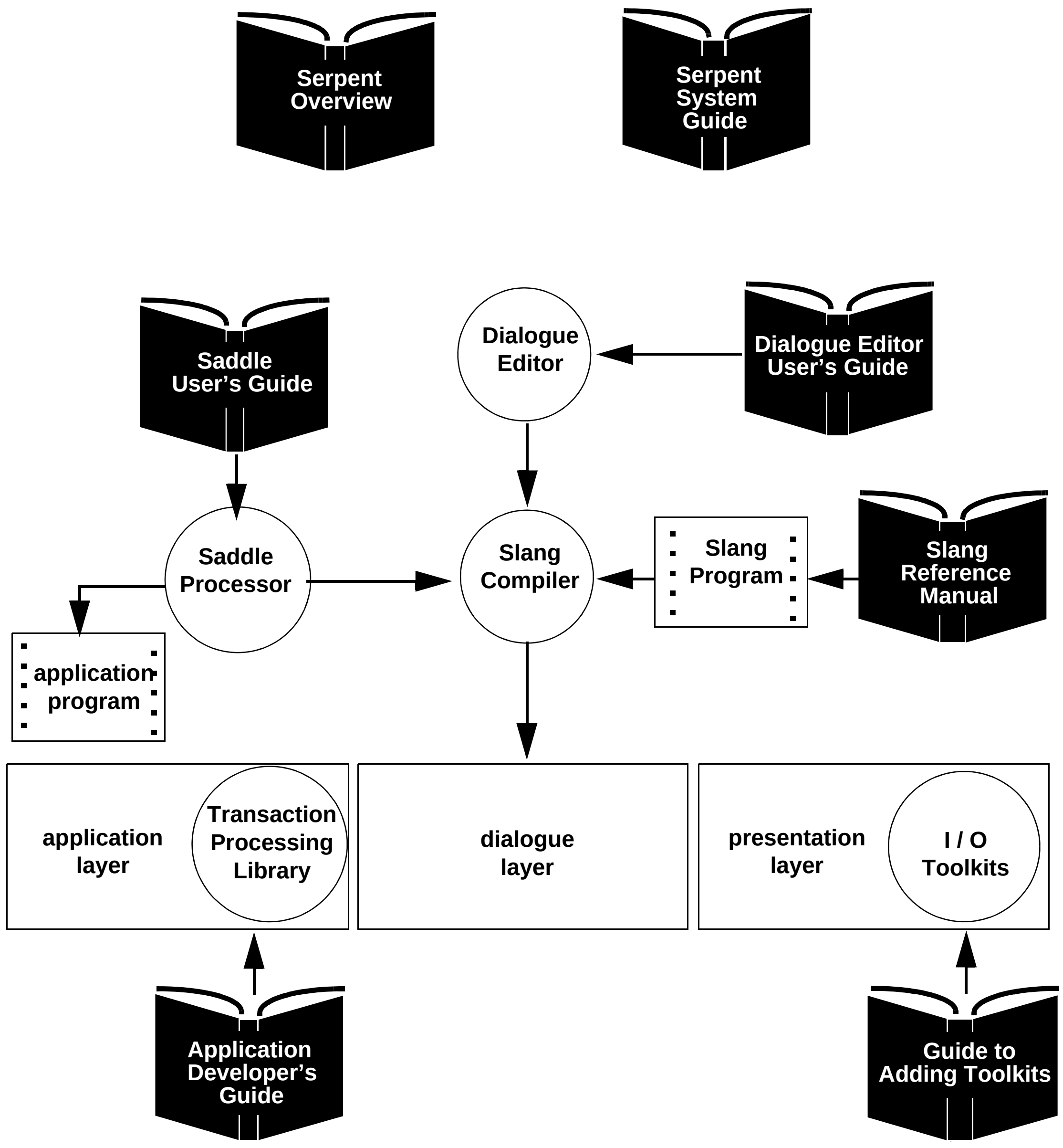

Figure 1-1 Serpent Documents

1.3 Installing Serpent

Serpent runs on most versions of Unix that support the AT&T Interprocess Communication (IPC) system. Typically, these Unix versions are based on either AT&T System VR2 (or later revisions), or BSD 4.3 (or later revisions), and include releases by Sun Microsystems, Digital Equipment Corporation, Hewlett Packard, IBM, and others.

Most examples in this document address applications written in C. Serpent also supports an Ada based interface, but for ease of description, the examples used in Chapter 4 of this document are written in C.

The Serpent system is automatically installed with the `tar` utility and the `INSTALL` C-shell script that is provided with Serpent. The installation script checks for all the required programs, and registers the installation site as a Serpent user. Once the installation procedure has completed, the directory hierarchy will exist as outlined in Section 2.3.

Other than running `tar` and `INSTALL`, usually no special installation actions need be done by the system administrator.[1]

[1] If the IPC is not configured in the Unix kernel, the system administrator may need to regenerate the kernel to include this facility.

2 System Configuration

This section outlines the layout of the Serpent system – the environment variables used by Serpent, the file formats and naming conventions, and the Serpent directory hierarchy.

2.1 Environment Variables

Serpent uses three environment variables that must be set before Serpent can be used. It is advisable to set these variables in your `.cshrc` or other shell initialization file. These variables are used by both the C-shell scripts and the executable programs within Serpent. The variables are:

`SERPENT_BASE_PATH`

This environment variable establishes the root of the Serpent hierarchy. It contains a single absolute path name, which is the location of the directory containing all of the Serpent subdirectories. The user may change this variable as necessary, but should be aware that changes to it may necessitate changes to the settings of the following two variables.

`SERPENT_DATA_PATH`

This environment variable specifies the data search path used by various Serpent components. It contains a space-separated list of the directories where referenced files may be located. The types of files that are located with this environment variable include: included files, `.ill` files, iconic bitmaps, etc. As specified in the Serpent distribution, the three components of this environment variable are: "`.`" (the current directory), `$SERPENT_BASE_PATH/data`, and `$SERPENT_BASE_PATH/include`. The user may add directories to this search path as desired.

`SERPENT_EXE_PATH`

This environment variable specifies the search path used to locate executable programs used by the Serpent system. It contains a space-separated list of directories to be searched to locate Serpent executable programs (note that this is different from the `PATH` environment variable established by the shell). As specified in the Serpent distribution, the two components of this environment variable are: "`.`" (the current directory) and `$SERPENT_BASE_PATH/bin`. The user may add directories to this search path as desired.

`IMAKEINCLUDE`

This environment variable tells the `imake` program where to find its included files (other than in whatever happens to be the current directory). As specified in the Serpent distribution, this variable is defined as the string "`-I$SERPENT_BASE_PATH/config`".

2.2 File Types and Naming Conventions

The Serpent system maintains a number of different types of files with different file name extensions. In the following, the file name `baz` is assumed.

1. Slang source file (`baz.sl`) – the source language (Slang) description of a Serpent dialogue. Slang source files describe the layout and interaction of a dialogue. The user may write directly in Slang or may use the dialogue editor to build a Slang dialogue.

   The Slang compiler will generate six or eight files from a compilation. The user should not attempt to modify any of these files (they are created with a leading period character to make them "invisible" to the `ls` command. Given a source file named `baz.sl`, these files would be named: `.baz.c`, `.baz.h`, `.baz.t`, `.baz.x`, `.baz.z`, `.baze.d`, `.bazi.d`, and `.bazt.d`. The file `.baz.o` will be produced from the C compilation of the file `.baz.c`.

   If dialogues are both compiled and linked (see Section 3.2 for details), an additional file will be generated: `baz`, an executable version of the dialogue.

2. Externals definitions (`baz.ext`) – a special type of include file used by the Slang compiler. Externals files are provided by the Slang compiler as a way of defining commonly used external functions, such as the Serpent string-handling library.

3. Saddle source file (`baz.sdd`) – the source language description of the Serpent shared data description. Shared data can serve two functions:

   • Shared data descriptions are used to describe the format (but not the layout in memory) of items to be shared between components of a running Serpent application, specifically, between the dialogue and the application, and between the dialogue and the toolkit interface program. Shared data elements are dynamically created and destroyed by either the dialogue or the application.
   • Shared data descriptions are also used to describe the format of dynamically created elements used exclusively within the dialogue. Although these elements are not shared with any other component of the Serpent system, the source description and the interface model are identical. The term *dialogue shared data* is reserved for this private "shared" data.

4. Interface language layer file (`baz.ill`) – one of the files generated by the Saddle compiler; this file is the internal representation of the shared data description. It should not be edited by the user.

5. Glue file (`baz.gl`) – the source level description of the toolkit interface. It is compiled by the Glue program into a technology file. A Glue description lists all of the widgets within a technology. For example, for the Motif widget set, the Glue file describes all of the Motif widgets.

   Two other file extensions are used within the Glue source. These are used as parts of a Glue description, but since they are incomplete portions of a description, they are assigned a different extension. These file extensions are:

   - Attribute list (`baz.at`) – a list of technology specific attribute/value pairs.
   - Method description (`baz.me`) – a technology-specific method description.

6. Technology file (`baz.tx`) –one of the files generated by the Glue compiler. It is a binary representation of the technology definition and should not be modified by the user.

2.3 Layout of Programs and Data Files

As shipped, the Serpent system contains source and include files for a number of programs. These files are automatically placed by the installation/register program. The directory hierarchy is as follows (refer to Section 3 for details on each section):

`bin`

The directory containing the executable images of all of the generated Serpent programs.

`c_toolkit`

The directory containing the Zimmermann list, hash, and tree packages. The C toolkit is used internally by most Serpent applications. This directory has two subdirectories, `c_toolkit/include` and `c_toolkit/src`.

`compiler`

The directory containing the Slang compiler. The compiler translates a Slang dialogue into a set of interpretable tables, which are used by the dialogue manager. The `compiler` directory has two subdirectories, `compiler/include` and `compiler/src`. The compiler is detailed in Section 3.2, and can be invoked with the `serpent` command.

`data`

This directory contains all the technology files generated for use by the toolkit interface (Six) and the `.ill` files for the same technologies, bitmap files for the dialogue editor, include files defining technology specific constants, etc.

`de`

This directory contains the dialogue editor. It has two subdirectories, `de/include` and `de/src`. The dialogue editor is detailed in the *Serpent: Dialogue Editor User's Guide* (CMU/SEI-91-UG-4).

`demos`

This directory contains a number of demonstration programs of varying complexity that illustrate the features of Serpent. This directory contains three subdirectories, `demos/sat`, `demos/smo`, and `demos/sol`, for dialogues written for the Athena Widget set, the Motif widget set, and the OpenLook widget set, respectively. Each demo is in a further subdirectory sorted by demo name. Documentation for each demo is contained within the `README` file in the demo subdirectory.

`dm`

This directory contains the source files for the dialogue manager. The dialogue manager is used to execute a compiled dialogue and realize it on the user's display. This directory has two subdirectories, `dm/include` and `dm/src`. The dialogue manager is detailed in Section 3.3. It is not invoked directly, but as part of the overall execution of a dialogue.

`docs`

This directory contains PostScript™ images of all of the Serpent documentation. Source files for Serpent documents are not provided.

`externs`

This is the source directory for Slang-external functions. i.e., functions provided with Serpent which are referenced via the `EXTERNALS` declaration in Slang. This directory contains two subdirectories, `externs/include` and `externs/src`.

`glue`

This is the directory containing the Glue program. Glue is the tool that builds technology bindings from descriptions. This directory has two subdirectories, `glue/include` and `glue/src`. The glue system is described in greater detain in Section 3.4 and in the *Serpent: Guide to Adding Toolkits* (CMU/SEI-91-UG-8), and is invoked with the `glue` command.

`include`

This directory contains all of the globally accessible include files for building the Serpent applications. It is also used by the user building a dialogue (via the environment variable `SERPENT_DATA_PATH`) to locate the necessary `.ill` files.

`int`

This directory contains the source and include files for the inter-process communication interface between the technology, application, and dialogue manager in the Serpent system. This directory has three subdirectories: `int/include`, `int/src`, and `int/ada`. The first two contain the C source and include files. The directory `int/ada` contains the Ada type declarations for Serpent and Ada bindings to the C object files, with a subdirectory for each Ada compiler – currently `int/ada/alsys` and `int/ada/verdix`.

`lib`

This directory contains all of the generated libraries used in the Serpent system.

`man`

This directory contains all of the manual page entries for the Serpent system, using the standard `-man` package. Two subdirectories, `man/man1` and `man/cat1,` follow the Unix manual page standard.

`master_ctags`

This file is a tags file for all of the Serpent system. All tags are automatically generated by the Makefiles by using the `ctags` utility. Each source directory also maintains its own tags file, but this file encompasses all of the Serpent source. To use both the local and master tags files within `vi`, the following command must be issued:

```
:set tags tags\ $SERPENT_BASE_PATH/master_ctags
```

To use Gnu EMACS, the Makefiles must be changed to use the `tags` utility. Within Gnu EMACS, the command:

```
Meta-x visit-tags-table
```

must be used to specify the correct tags file set.

`saddle`

This directory contains the source and include files for the Saddle processor. This directory has two subdirectories, `saddle/include` and `saddle/src`. Saddle is explained in greater detail in Section 3.1 and in the *Serpent: Saddle User's Guide* (CMU/SEI-91-UG-3), and is invoked with the `sdd` command.

`six`

This source directory contains the source for the Six toolkit, as well the generated files for the Motif, OpenLook, and Athena widget sets. Six is the Serpent Interface to X, and is described in more detail in the *Serpent: Guide to Adding Toolkits* (CMU/SEI-91-UG-8). This directory has two subdirectories, `six/src` and `six/include`. The toolkit uses Glue, Saddle, and hand-tailored C code to implement a Serpent interface to a technology. The individual toolkits are not executed directly, but rather as part of a general dialogue execution.

`tools`

This directory contains the source files for various tools used with Serpent, including the preprocessor, dialogue debug tools, shared memory debug tools, and the `serpent` command. This directory has two subdirectories, `tools/include` and `tools/src`.

`utilities`

This source directory contains utility routines used throughout the Serpent system, including a general symbol table management package and assorted string and data management routines. This directory has two subdirectories, `utilities/include` and `utilities/src`.

3 System Components

This section outlines the interaction between the various components of the Serpent system. These interactions are more fully described in Section 4, where a complete example dialogue and application are constructed using all of the components of the Serpent system.

The key in Figure 3-1 shows the shapes of objects used in all of the following figures:

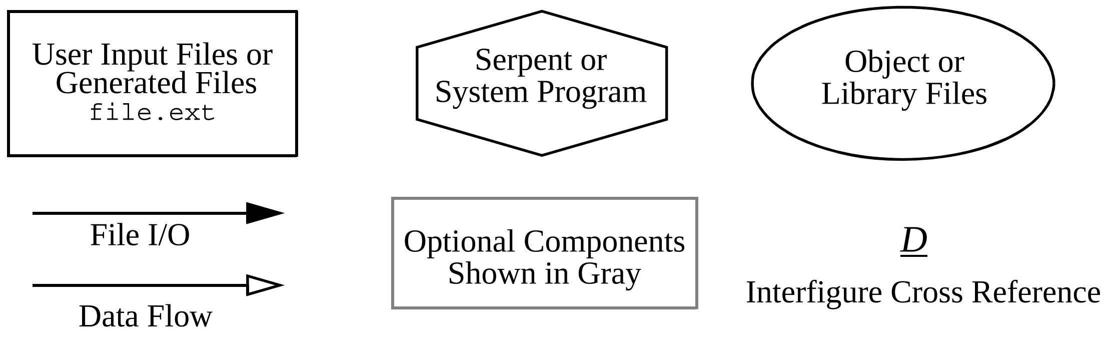
User Input Files or Generated Files
file.ext
Serpent or System Program
Object or Library Files
File I/O
Data Flow
Optional Components Shown in Gray
D
Interfigure Cross Reference

Figure 3-1 – Key to Illustrations

Interfigure cross references are used for files that are generated by one component of the Serpent system and used by another. For example, .ill files are generated by the Saddle compiler (described in Section 3.1) and used by the Slang compiler (Section 3.2) and by the dialogue manager runtime (Section 3.3). To facilitate your understanding of the relationships between the various components, the .ill files are tagged with the letter *A*.

3.1 Saddle Compiler

The Saddle compiler is used to translate an editable description of a shared data layout (a Saddle file) into an internal form (an .ill file). Saddle descriptions are used to define the components of shared data records that the dialogue or application can create and share between them.

The Saddle description also describes the way in which the associated application program is executed (this can be a toolkit interface or a user application program). When the Slang compiler reads the .ill files, it determines which programs must be run to instantiate the user-defined system.

The path from Saddle source file to .ill file is illustrated in Figure 3-2. The Saddle compiler reads the user-created Saddle source file[2] and creates two output files. The first is an .ill file (labelled *A*), which is used later by both the Slang compiler and the dialogue manager runtime. The .ill file is created in a binary format that is not designed to be edited by the user. The second file is a C include file (or an Ada spec, labelled *B*), which may be used by the application program. The include file creates a set of `typedef`s which are equivalent to the declarations made in the Saddle file. In this way, the application program can use a data format that is identical to that known by the dialogue for all shared data transactions.

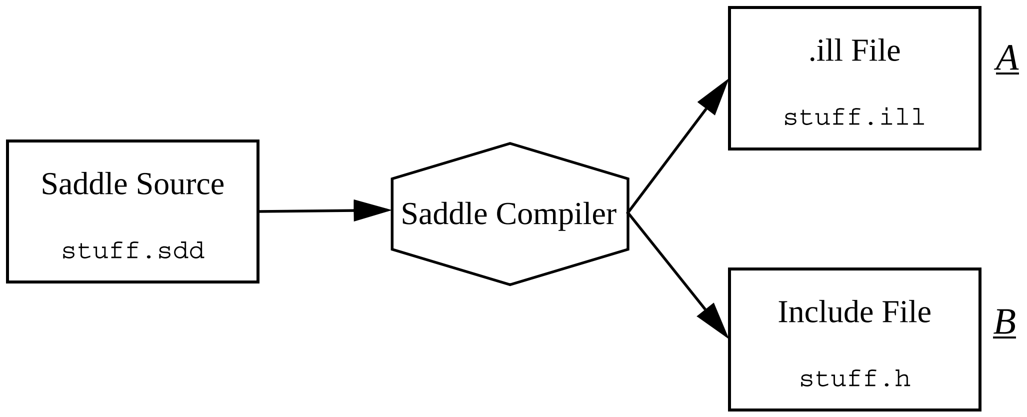

Figure 3-2 – Saddle Compiler

If an application program exists, the only way it can share data between itself and the dialogue is through shared data described in a Saddle description. However, not all dialogues need an application program. The `counter` demo is one such example. It is also important to note that the Saddle description does not declare any shared data – it merely defines the format of the various shared data *types*, which then allows either the dialogue or the application to create instances of those types at runtime.

3.2 Slang Compiler and Linker

The Slang compiler is used to translate a user-written Slang dialogue into an executable form. A Slang file describes four aspects of the dialogue:

1. Which programs must be run to instantiate the user dialogue (this information is obtained in the .ill files)

2. The layout of and relationships between objects on the screen when the dialogue is executed

[2] Saddle files can also be created by the Glue compiler (Section 3.4). The Saddle compilation process is identical irrespective of source.

3. The interactions between objects in the dialogue layer, interactions between the dialogue and the user application program, and actions the dialogue should take in response to user actions
4. Conditions under which groups of objects (i.e., view controllers) are created and destroyed

The pathway for compiling a Slang dialogue into an executable image is illustrated in Figure 3-3. The user-created dialogue (in this example, named `baz.sl`) is run through the compiler and linker to produce an executable image (labelled *D*, and in this example, automatically given the name `baz`). The compiler also generates a number of other

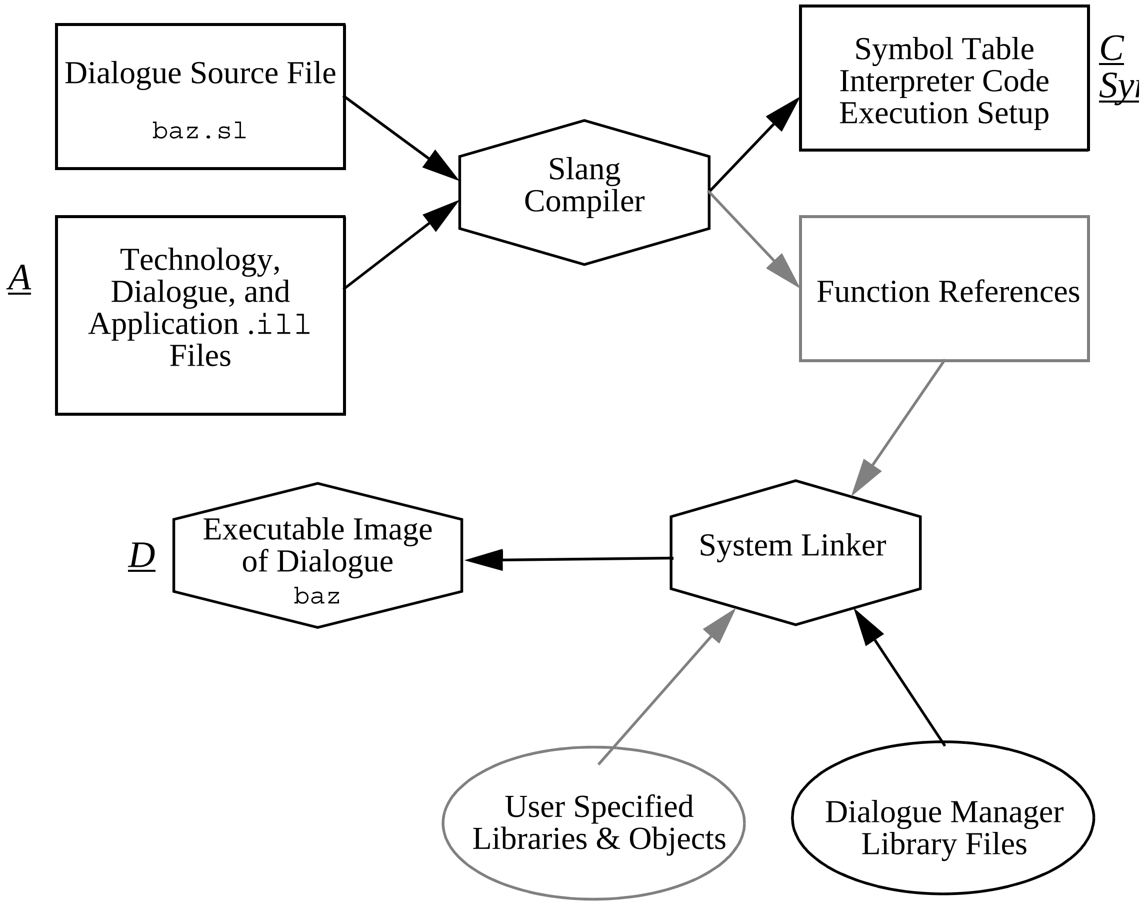
Dialogue Source File
baz.sl
Technology, Dialogue, and Application .ill Files
A
Slang Compiler
Symbol Table Interpreter Code Execution Setup
C
Sy
Function References
System Linker
Executable Image of Dialogue baz
D
User Specified Libraries & Objects
Dialogue Manager Library Files

Figure 3-3 – Slang Compiler and Linker

"invisible" files (enumerated in Section 2.2): interpreter source code files, an execution initialization file (containing the execution instructions from the .ill files), and symbol table files.[3] The compiler also reads in the .ill files (labelled *A*) needed for this compilation. The .ill files can be for a technology (e.g., `smo.ill` for the Motif widget set), for application shared data, or for dialogue shared data.

Compiling a dialogue is accomplished with the `serpent` command. The `-c` switch is used to compile a dialogue (producing the symbol table, interpreter code, execution setup, and external function reference files), while the `-l` switch is used to process an existing function reference file and link it with the dialogue manager runtime libraries to produce an executable image of the dialogue. Both the `-c` and `-l` switches may be specified together, which will cause the dialogue to be compiled and linked in the same command. The `serpent` command is described in more detail in its manual page.

3.3 Dialogue Manager

Once a dialogue is compiled, it must be executed to visualize the dialogue. This can be done either through the `serpent` command with the `-g` switch, or by simply typing in the name of the dialogue itself. When a dialogue is executed, two or more programs are started automatically. One program is run for each technology that the dialogue uses (e.g., `smo` for the Motif toolkit, `sat` for the Athena toolkit, etc.), one (or more) for the user application program(s) (if present), and one for the dialogue manager.

The dialogue manager is not a separate program *per se*, but is a collection of library routines which are linked with a compiled form of the user-specified dialogue. Figure 3-4 shows the actions of the dialogue manager during the execution of a dialogue. The executable version of the dialogue,

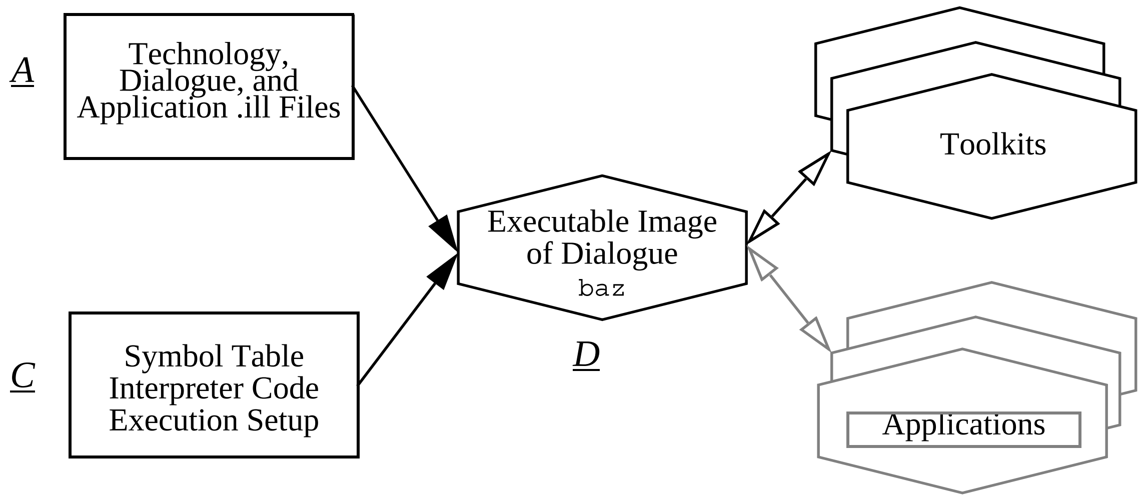
A
Technology, Dialogue, and Application .ill Files
C
Symbol Table Interpreter Code Execution Setup
Executable Image of Dialogue
baz
D
Toolkits
Applications

Figure 3-4 – Dialogue Manager

[3] The user should not change the name of the executable file directly; the names of all of the generated files are bound into the executable image. These files (labelled *C*) are later used by the dialogue manager runtime (see Figure 3-4).

generated by the Slang compiler and linker (labelled *D*), reads in the `.ill` files specified in the Slang source (labelled *A*) and the symbol table and interpreter code generated by the Slang compiler (labelled *C*). The dialogue manager does not generate any files, but communicates with the technologies (that were generated through Glue, Six, or other methods) and the application(s) (described in Section 3.5) to cause the dialogue to be executed.

3.4 Toolkit Interface

The toolkit interface is the mechanism by which a dialogue communicates with the display medium. Each toolkit interface is a separate program, a copy of which must be executed for each running dialogue. Toolkit interfaces are used, for example, to visualize and interact with toolkit widgets so that the dialogue (and dialogue writer) need not be concerned with the mechanics of the toolkit.

A toolkit interface may be written in one of two ways: either by writing it from scratch, using the Serpent shared memory interface, or by describing the toolkit with Glue, and writing a minimal amount of C code to produce a Six binding. Ordinary Serpent users do not need to concern themselves with toolkit integration. Only when a new toolkit (or a new release of an already bound toolkit) is released does a toolkit interface need to be constructed.

3.4.1 Glue and the Six Interface

If a toolkit is based on the standard X Toolkit model, it is fairly easy to build a toolkit interface with the programs Glue and Six. Figure 3-5 shows the pathway followed for toolkit integration in Serpent.The Glue program reads in a description of an Xt-based toolkit and creates a toolkit file (labeled *E*) and (optionally) some ancillary C files. The Serpent interface to X (Six), is linked with the ancillary C files to produce a toolkit-specific interface.

3.4.2 Other Paths for Toolkit Integration

Two toolkit interfaces are provided with the Serpent distribution: Motif and Athena. Because the underlying toolkits are Xt-based, each of these interfaces was created through the use of Glue and Six. However, not all toolkits are Xt-based, and other methods can be used to create a Serpent toolkit interface.

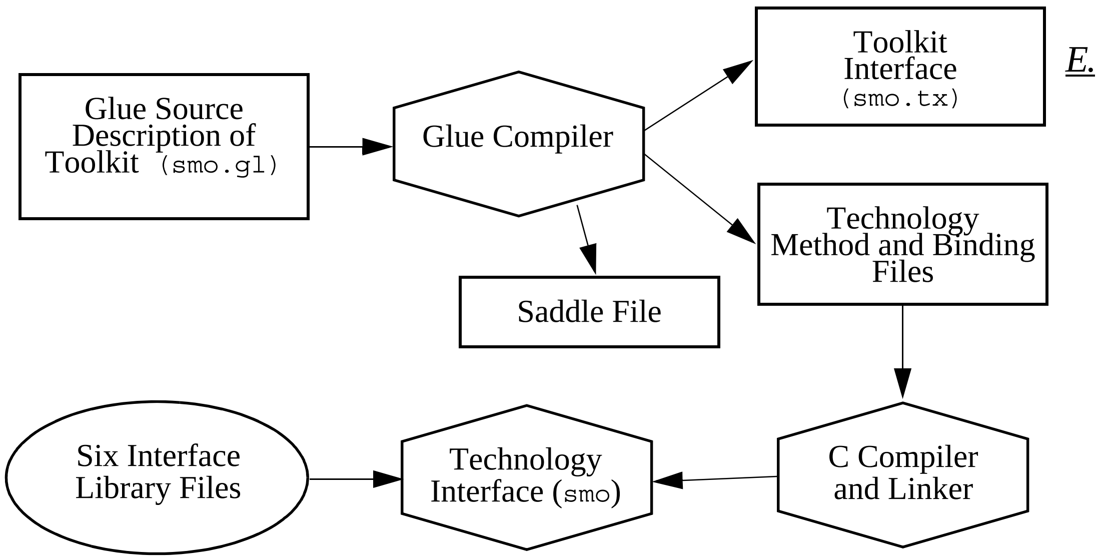
Glue Source Description of Toolkit (smo.gl)
Glue Compiler
Toolkit Interface (smo.tx)
E.
Technology Method and Binding Files
Saddle File
Six Interface Library Files
Technology Interface (smo)
C Compiler and Linker

Figure 3-5 – Glue and Six

Two examples are a digital mapping system and a gesturing system. Although both of these interfaces have been created with Serpent, they were done on an experimental basis and are not distributed with Serpent. Refer to the *Serpent: Guide to Adding Toolkits* (CMU/SEI-91-UG-8) for full details on toolkit integration.

3.5 User Application Program

Depending on the complexity of the system, a dialogue may have an application program associated with it. The application is responsible for performing those actions which are outside of the aegis of the user interface. It is certainly possible to place application code in the dialogue, just as it is possible to put user interface code in an application. Serpent was designed, however, with separation of concerns in mind. As a simple example, an application program can be a database system – the user interface can be textual, button oriented, or some other mechanism. The application is tasked with accessing the database, irrespective of the user interface characteristics.

The relationship between the application program and the dialogue is shown in Figure 3-4. The application is free to perform any manner of calculations and accesses of external files it wishes. In the current version of Serpent, application programs may be written in C or Ada, or in any language that can interface with either C or Ada. The only restriction is that the application communicate with the dialogue manager (and thus the running dialogue) through the Serpent interface routines. These routines are described in detail in the *Serpent: C Application Developer's Guide* (CMU/SEI-91-UG-6) and in the *Serpent: Ada Application Developer's Guide* (CMU/SEI-91-UG-7). Section 4.3.4 outlines the steps needed to link the interface routines with the application program.

3.6 Running A Dialogue/Application

Once the dialogue (and optional application program) have been compiled and linked, running the dialogue is a simple matter of typing the dialogue name. The user may also type `serpent -g` *`<dialogue>`*, and achieve the same result (full details can be found in the manual page for the `serpent` command). In both cases, the `serpent` command initializes the Serpent environment and executes the required programs. Although the initialization actions of the Serpent system are automatic, it is useful to understand what happens when a dialogue starts executing.

The following steps occur when running a dialogue and application. Assume that the dialogue is named `baz`:

1. The `serpent` command looks for a file named `.bazi.d` in the directories specified in the environment variable `SERPENT_EXE_PATH`. This file contains the names of `.ill` files and the associated mailboxes that are used by the dialogue. Mailboxes are the mechanism that Serpent uses to communicate between processes. Although it may not be explicitly listed, the name `DM_BOX` will be used if there is no specific application mailbox.

   The `serpent` command creates a Unix message queue for each mailbox listed in `.bazi.d` (and one for `DM_BOX`), as well as Unix shared memory segment(s) and a Unix semaphore set for controlling access to the shared memory. The Unix IPC handles for each of these resources are passed through environment variables to the programs that are executed in the following steps.

2. The `serpent` command next looks for a file named `.baze.d`. This file contains the files which must be executed to realize the dialogue. Each program name (and arguments) is preceded by the name of an environment variable. If the user has an environment variable with this name, the string contained in the environment variable will be executed instead of the command in the file named `.baze.d`. This is one way in which one of the programs in a dialogue may be debugged – by substituting a debugger execution instead of the normal command execution.

   Note that the environment variable `SERPENT_EXE_PATH` is used to find the executable image and not the variable `PATH`. This means that in order to run a debugger, there must be a symbolic link to the debugger from somewhere in `SERPENT_EXE_PATH`.

3. The programs listed in `.baze.d` are executed. Typically, this will include the dialogue manager (which for this example would be named `baz`), one or more toolkit interfaces (typically `smo`, `sat`, or `sol`), and if the user has specified any, one or more application program. Once executing, a dialogue appears as shown in Figure 3-4. Each of the running programs communicates through the message queues, semaphores, and shared memory established in Step 1.

4 Example Dialogue/Application

To best understand the interaction of the various components of a Serpent dialogue, and to better understand how a dialogue and application can be created using a stepwise refinement technique, this document concludes with a sample dialogue and application. Rather than show just the completed system, however, the example constructs the system from scratch, much as a prototype system would be developed into a fully functional system.

This particular example builds a clock which displays the current time of day and allows its user to optionally set the time. The user will also be able to optionally display the date along with the time. The example is broken into four stages:

1. A layout of the dialogue is constructed, with all objects visible and none of the object interactions constructed. There is no application program interfaced with the dialogue.
2. The object interactions are more fully specified, so that objects are made visible under certain conditions and hidden under others. Although there is no application program (that is, the date and time will be hard-wired constants in the dialogue), the user will be able to exercise the “show date” and “set time” features of the dialogue.
3. An application program is constructed. With almost no change to the dialogue, the application program will supply the current time and date to the dialogue, and the dialogue will communicate changes made by the user back to the application.
4. A completely different dialogue is constructed using the application program written for stage 3. This final stage shows one aspect of the flexibility of Serpent, where different user interfaces can be evaluated for a single application program.

A picture of the running dialogue in its final form is shown in Figure 4-1:

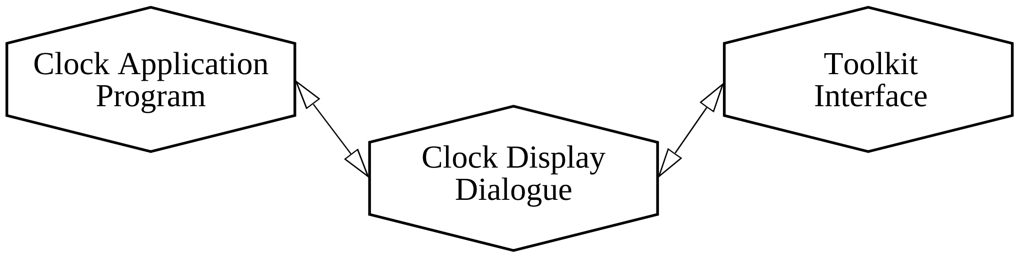

Figure 4-1 – Picture of Executing Dialogue

4.1 First Stage of Developing Clock

In its first stage of development, the sample clock is simply a layout that can be used to show the placement of objects in the dialogue. None of the controls actually do anything to affect the dialogue or layout, although the toggle buttons can be turned on and off due to the behavior of the technology (in this case, the Motif widget set).

The layout of the first stage as it appears on the screen is shown in Figure 4-2. Note that even though the "Show Date" toggle has not been selected, the date is shown. This is because the dialogue is used

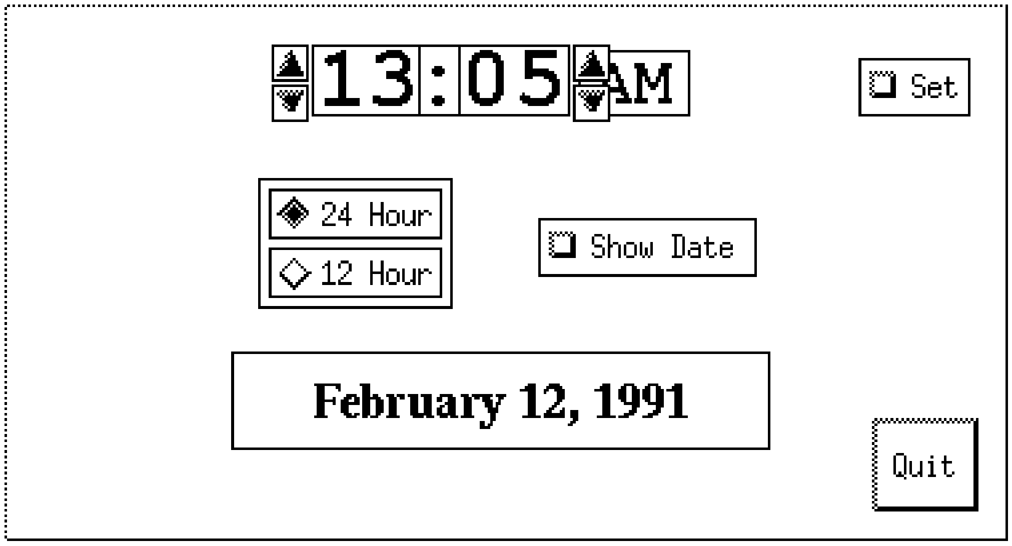

Figure 4-2 – Initial Clock Dialogue Layout

only as a layout example – the controls are just "dummied up" to appear as they will in the final dialogue. Note also that the arrow buttons to the right of the minutes indicator overlaps the AM/PM indicator. This is intentional, and is in anticipation of the AM/PM indicator being movable, depending on whether the arrow buttons are present or not.

Finally, note that all objects have been drawn with borders around them. While this is the default for some objects, for other objects (such as toggle buttons), the borders have been added to emphasize visibility and placement. These borders are removed in the final version of the dialogue.

4.1.1 Slang Dialogue

The Slang dialogue for the sample clock is shown in Example 4-1. This dialogue may also be found as the `file demos/smo/clock/one.sl` in the Slang source hierarchy.

Note that the first line of the Slang dialogue includes the file `smo.ill`, indicating to Slang that the Motif toolkit will be used throughout the dialogue. The file `glueXm.h` is also included. This file defines the constants `XmARROW_UP`, `XmPACK_COLUMN`, etc. Although a different toolkit could have been used, all widgets in this example dialogue are from the Motif toolkit.

```
#include "smo.ill"

|||

#include "glueXm.h"

OBJECTS :

    palette : XmBulletinBoard {
       ATTRIBUTES :
          height : 200;
          width : 375;
          }

    quit : XmPushButton {
       ATTRIBUTES :
          parent : palette;
          labelstring : "Quit";
          height : 35;
          width : 40;
          x : palette.width - width - 10;
          y : palette.height - height - 10;
       METHODS :
          notify : { exit(); }
          }


    hours : XmLabel {
       ATTRIBUTES :
          parent : palette;
          fontlist : "*-courier-bold-r-*-*-34-*";
          labelstring : "13";
          recomputesize : false;
          borderwidth : 1;
          height : 25;
          width : 40;
          x : 115;
          y : 15;
          }
```

```
colon : XmLabel {
   ATTRIBUTES :
      parent : palette;
      fontlist : “*-courier-bold-r-*-*-34-*”;
      labelstring : “:”;
      recomputesize : false;
      borderwidth : 1;
      height : 25;
      width : 15;
      x : 155;
      y : 15;
      }

minutes : XmLabel {
   ATTRIBUTES :
      parent : palette;
      fontlist : “*-courier-bold-r-*-*-34-*”;
      labelstring : “05”;
      recomputesize : false;
      borderwidth : 1;
      height : 25;
      width : 40;
      x : 170;
      y : 15;
      }

AM_PM : XmLabel {
   ATTRIBUTES :
      parent : palette;
      fontlist : “*-courier-bold-r-*-*-24-*”;
      labelstring : “AM”;
      recomputesize : false;
      borderwidth : 1;
      height : 23;
      width : 40;
      x : 215;
      y : 17;
      }

hrs_up : XmArrowButton {
   ATTRIBUTES :
      parent : palette;
      arrowdirection : XmARROW_UP;
      shadowthickness : 0;
      borderwidth : 1;
      height : 12;
      width : 12;
      x : 100;
      y : 15;
      }
```

```
hrs_down : XmArrowButton {
   ATTRIBUTES :
      parent : palette;
      arrowdirection : XmARROW_DOWN;
      shadowthickness : 0;
      borderwidth : 1;
      height : 12;
      width : 12;
      x : 100;
      y : 30;
      }

mins_up : XmArrowButton {
   ATTRIBUTES :
      parent : palette;
      arrowdirection : XmARROW_UP;
      shadowthickness : 0;
      borderwidth : 1;
      height : 12;
      width : 12;
      x : 213;
      y : 15;
      }

mins_down : XmArrowButton {
   ATTRIBUTES :
      parent : palette;
      arrowdirection : XmARROW_DOWN;
      shadowthickness : 0;
      borderwidth : 1;
      height : 12;
      width : 12;
      x : 213;
      y : 30;
      }

date : XmLabel {
   ATTRIBUTES :
      parent : palette;
      fontlist : “*-times-bold-r-*-*-20-*”;
      labelstring : “February 12, 1991”;
      recomputesize : false;
      borderwidth : 1;
      height : 35;
      width : 200;
      x : 85;
      y : 130;
      }
```

```
FormatSelector : XmRowColumn {
   ATTRIBUTES :
      parent : palette;
      height : 100;
      width : 200;
      x : 95;
      y : 65;
      packing : XmPACK_COLUMN;
      numcolumns : 1;
      orientation : XmVERTICAL;
      radiobehavior : true;
      radioalwaysone : true;
      }

Button1 : XmToggleButton {
   ATTRIBUTES :
      parent : FormatSelector;
      labelstring : “12 Hour”;
      indicatortype : XmONE_OF_MANY;
      borderwidth : 1;
      width : 75;
      height : 30;
      set : false;
      }

Button2 : XmToggleButton {
   ATTRIBUTES :
      parent : FormatSelector;
      labelString : “24 Hour”;
      indicatortype : XmONE_OF_MANY;
      borderwidth : 1;
      width : 75;
      height : 30;
      set : true;
      }

Show_Date : XmToggleButton {
   ATTRIBUTES :
      parent : palette;
      labelString : “Show Date”;
      borderwidth : 1;
      x : 200;
      y : 80;
      width : 80;
      height : 20;
      set : false;
      }
```

```
            Set_Time : XmToggleButton {
               ATTRIBUTES :
                  parent : palette;
                  labelString : “Set”;
                  borderwidth : 1;
                  x : 320;
                  y : 20;
                  width : 40;
                  height : 20;
                  set : false;
                  }
```

Example 4-1 Slang Dialogue for Sample Clock

4.1.2 Makefile

The Makefile for this dialogue is very simple. It needs only to compile the dialogue file with the Slang compiler, and needs no application program or external C routines.[4]

```
one:      one.sl
    serpent -cl one
```

4.2 Second Stage of Developing Clock

In the second stage of development, the dialogue for the clock still exists without an application (that is, the time and date are still hardwired), but the interactions between objects have been defined, and the controls now actually do something . Figure 4-3 shows

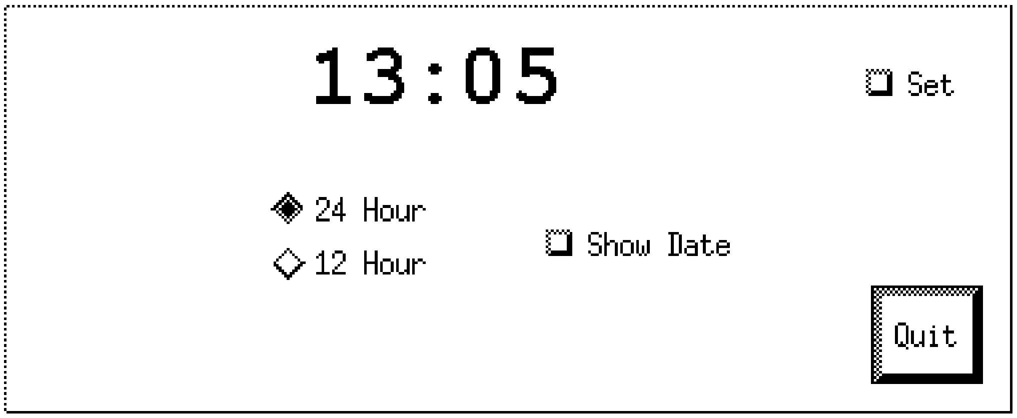

Figure 4-3 – Clock Dialogue, Revised Layout

the revised layout of the clock program as it appears at program start-up.

[4] The actual Makefile for the dialogue is more complicated than this, due to `imake` complexities and the fact that the Makefile contains instructions for all four examples shown in this document. However, for this single dialogue, these are the only requirements.

There are a few important things to note about the appearance of the dialogue as it is initially presented. The first is that the borders have been removed from most of the objects, and that the shadow border around the “Quit” button has been enhanced to improve its visibility. One advantage of Serpent is that it allows incremental refinement of the appearance or behavior of a dialogue during the specification or development stage of a project.

The second thing to note is that the visibility of the “set” buttons (the arrows to the left and right of the time fields), the AM/PM indicator, and the date have all been tied to the actions of the corresponding controls. This is done by placing them in Serpent view controllers whose creation conditions (and implicitly, destruction conditions) are based on the values of variables or of attributes within other objects.

Figure 4-4 shows the display that the user sees when the button labelled “Show Date” (the ToggleButton object named `Show_Date`) is pressed. The `toggle` method associated with the object

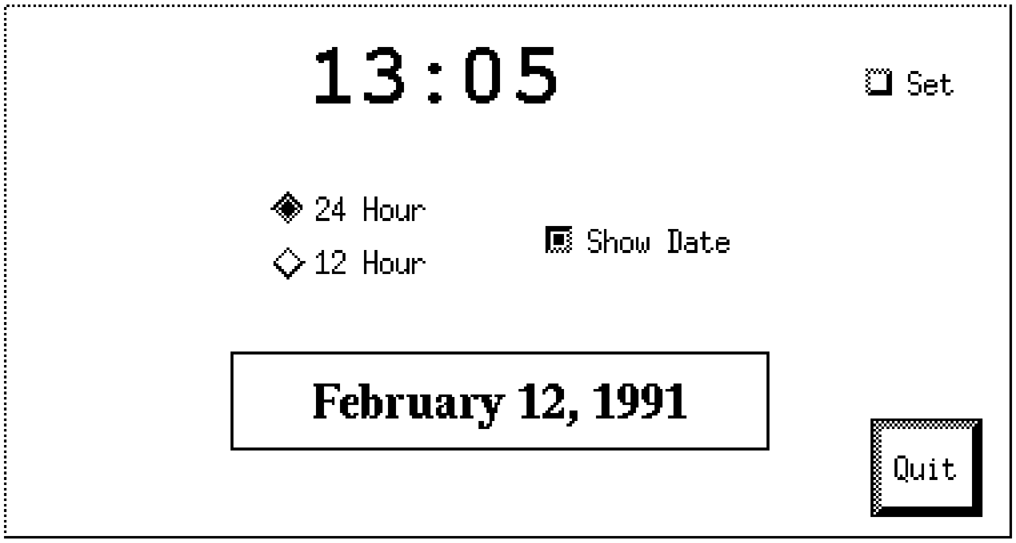

Figure 4-4 – Revised Layout Showing Date Enabled

is activated, which simply sets the variable `doShowDate` to the value of the `set` attribute within the object. The date (still hardwired in the dialogue) appears in its specified location. Note that the BulletinBoard widget (called `palette`) which encloses all of the objects, grows to allow the date to fit. This is because the height of `palette` depends upon the variable `DatePlus` in the dialogue, which in turn depends upon the value of the `doShow_Date` variable. Note also that the “Quit” push-button shifts downwards to accommodate the change in `palette`’s size. This is because the “Quit” button’s `x` and `y` attributes depend on the `height` and `width` attributes of `palette` (that is, the location of “Quit” is specified relative to the bottom right corner of the BulletinBoard widget to which it is parented).

Although the ToggleButton turns black to indicate that it is selected, this action is not directly specified in the dialogue; rather, this action is provided by the toolkit interface, in this case, `smo`, the Serpent/Motif interface. When the “Show Date” button is pressed a second time, the button will once more turn white (and the `toggle` method will be activated to reflect the new value of the `set` attribute).

Figure 4-5 shows the same dialogue with the “Show Date” button deselected, but with the radio button for “12 Hour” mode selected. Because the variable `DatePlus` now has a value of `0`, the `palette` shrinks back to its original size, and the “Quit” push-button moves upwards again. When the “12 Hour” radio button is pushed, the `toggle` method associated

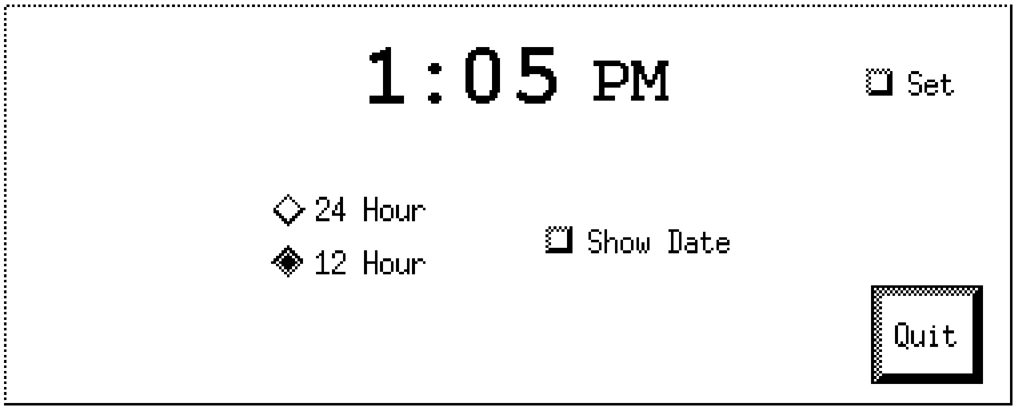

Figure 4-5 – Revised Layout Showing 12-Hour Mode Enabled

with the button being pressed (*and* the button being “unpressed”) are called. In this case, both actions set the variable `MilitaryTime` to false. The double action is redundant here, but does not harm anything in the dialogue.

Because the creation condition for the `AM_PM` view controller is specified as `not MilitaryTime, when MilitaryTime` becomes false, the view controller is created. Since the value of `hours_text` is greater than 12, the `labelstring` associated with the `AM_PM` object within the view controller of the same name becomes “PM”. Note that the definition and assignment of the `labelstring` attribute is only performed after the view controller is created; no actions are performed in the object if the creation condition for the view controller is not satisfied. Note also that if the “24 Hour” button is pressed (which will cause `MilitaryTime` to become true), the view controller and all of its associated objects will be deleted.

Figure 4-6 shows the same dialogue with the “Set” ToggleButton depressed. When the `Set_Time` object is clicked in the first time, the variable `doSet_Time` is set to true. This causes the `set_buttons` view controller to be created, which causes the four arrow buttons to appear. Note that the AM_PM object shifts to the right to make room for the buttons. (The

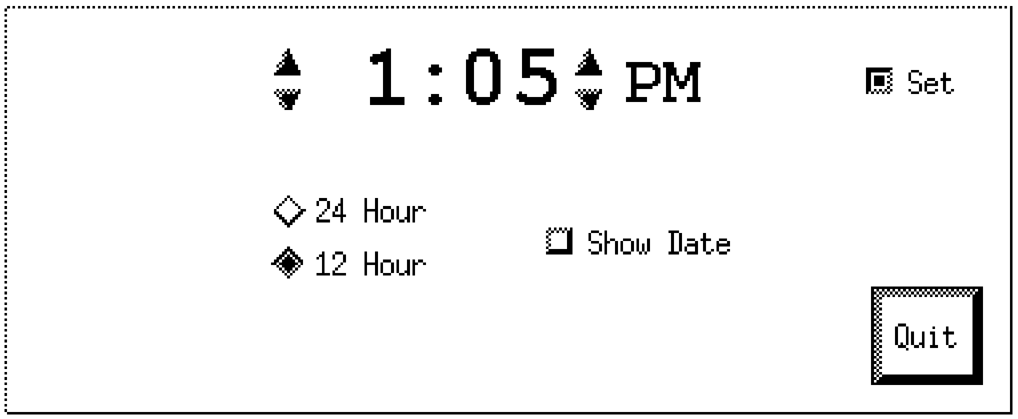

Figure 4-6 – Revised Layout Showing "Set" Buttons Enabled

x attribute of the object is dependent on the variable SetPlus, which is in turn dependent on the variable doSet_Time.)

When the up and down buttons are pressed by the user, the value of hours_text or minutes_text are incremented or decremented in the notify methods associated with each of the ArrowButton objects. The method code is responsible for keeping the hours within the range of 0 and 23, and for keeping minutes between 0 and 59. The code snippet associated with the hours object is responsible for displaying the hour value in the proper (12 or 24 hour) format.

4.2.1 Slang Dialogue

The Slang dialogue for the sample clock is shown in Example 4-2. This dialogue may also be found as the file demos/smo/clock/two.sl in the Slang source hierarchy.

One predominant change from the last example is that the AM/PM indicator, the "set" buttons, and the date display have all been moved into view controllers. This Serpent paradigm allows objects to be created and destroyed, based on conditions within the dialogue (so that, for example, the date may appear when the value of the variable doShow_Date is true). Another major change is that the controls are now live, in that the method code has been filled in and that the values of certain attributes (the labelstring attribute of the hours object) are now dependent on complex code snippets instead of static text.

The AM/PM indicator now has its location based on the presence of the set buttons (which is to say, it shifts over to make room for the buttons). Additionally, a shell widget was added to allow the display to grow when the date is shown.

Another change worth noting is that the dialogue now uses an external C routine, called `i2str`. This routine is described in greater detail in Section 4.2.2. Finally, compare the way in which the `toggle` method in the `Button1` and `Button2` objects is used to accomplish actions similar to that using the `set` attribute in the `Show_Date` and `Set_Time` objects.

Note: change bars are used to show where the code in this example differs from the previous example.

```
#include “smo.ill”

|||

#include “glueXm.h”

EXTERNALS :
    string i2str(integer);

VARIABLES :
    hours_text : “13”;
    minutes_text : “05”;
    DatePlus : {
       IF (Show_Date.set) THEN
          DatePlus := 50;
       ELSE
          DatePlus := 0;
       ENDIF;
       }
    SetPlus : {
       IF (Set_Time.set) THEN
          SetPlus := 12;
       ELSE
          SetPlus := 0;
       ENDIF;
       }
    MilitaryTime : true;

OBJECTS :

    shell : XmTopLevelShell {
       ATTRIBUTES :
          height : palette.height;
          width : palette.width;
          allowShellResize : true;
          }

    palette : XmBulletinBoard {
       ATTRIBUTES :
          parent : shell;
          height : 150 + DatePlus;
          width : 375;
          }
```

```
quit : XmPushButton {
   ATTRIBUTES :
      parent : palette;
      labelstring : “Quit”;
      borderwidth : 1;
      shadowthickness : 3;
      height : 35;
      width : 40;
      x : palette.width - width - 10;
      y : palette.height - height - 10;
   METHODS :
      notify : { exit(); }
      }

FormatSelector : XmRowColumn {
   ATTRIBUTES :
      parent : palette;
      height : 100;
      width : 200;
      x : 95;
      y : 65;
      packing : XmPACK_COLUMN;
      numcolumns : 1;
      orientation : XmVERTICAL;
      borderwidth : 0;
      radiobehavior : true;
      radioalwaysone : true;
      }

Button1 : XmToggleButton {
   ATTRIBUTES :
      parent : FormatSelector;
      labelString : “12 Hour”;
      indicatortype : XmONE_OF_MANY;
      width : 75;
      height : 30;
      set : false;
   METHODS:
      toggle : { MilitaryTime := not set; }
      }

Button2 : XmToggleButton {
   ATTRIBUTES :
      parent : FormatSelector;
      labelString : “24 Hour”;
      indicatortype : XmONE_OF_MANY;
      width : 75;
      height : 30;
      set : true;
   METHODS:
      toggle : { MilitaryTime := set; }
      }
```

```
Show_Date : XmToggleButton {
   ATTRIBUTES :
      parent : palette;
      labelString : “Show Date”;
      x : 200;
      y : 80;
      width : 80;
      height : 20;
      set : false;
      }

Set_Time : XmToggleButton {
   ATTRIBUTES :
      parent : palette;
      labelString : “Set”;
      x : 320;
      y : 20;
      width : 40;
      height : 20;
      set : false;
      }

hours : XmLabel {
   ATTRIBUTES :
      parent : palette;
      fontlist : “*-courier-bold-r-*-*-34-*”;
      labelstring : {
         IF (MilitaryTime) THEN
            labelstring := i2str(hours_text);
         ELSIF (hours_text = 0) THEN
            labelstring := 12;
         ELSIF (hours_text <= 12) THEN
            labelstring := hours_text;
         ELSE
            labelstring := hours_text - 12;
         ENDIF;
         }
      recomputesize : false;
      alignment : XmALIGNMENT_END;
      height : 25;
      width : 40;
      x : 115;
      y : 15;
      }

colon : XmLabel {
   ATTRIBUTES :
      parent : palette;
      fontlist : “*-courier-bold-r-*-*-34-*”;
      labelstring : “:”;
      recomputesize : false;
      height : 25;
      width : 15;
      x : 155;
      y : 15;
      }
```

```
minutes : XmLabel {
   ATTRIBUTES :
      parent : palette;
      fontlist : “*-courier-bold-r-*-*-34-*”;
      labelstring : i2str(minutes_text);
      recomputesize : false;
      height : 25;
      width : 40;
      x : 170;
      y : 15;
      }

VC : AM_PM
   CREATION CONDITION : (not MilitaryTime)

   OBJECTS :
      AM_PM : XmLabel {
         ATTRIBUTES :
            parent : palette;
            fontlist : “*-courier-bold-r-*-*-24-*”;
            labelstring : {
               IF (hours_text >= 12) THEN
                  labelstring := “PM”;
               ELSE
                  labelstring := “AM”;
               ENDIF;
               }
            recomputesize : false;
            height : 23;
            width : 40;
            x : 215 + SetPlus;
            y : 17;
            }
   ENDVC AM_PM

VC : today
   CREATION CONDITION : (Show_Date.set)

   OBJECTS:
      date : XmLabel {
         ATTRIBUTES :
            parent : palette;
            fontlist : “*-times-bold-r-*-*-20-*”;
            labelstring : “February 12, 1991”;
            recomputesize : false;
            borderwidth : 1;
            height : 35;
            width : 200;
            x : 85;
            y : 130;
            }
ENDVC today
```

```
VC : set_buttons
   CREATION CONDITION : (Set_Time.set)

   OBJECTS :
      hrs_up : XmArrowButton {
         ATTRIBUTES :
            parent : palette;
            arrowdirection : XmARROW_UP;
            shadowthickness : 0;
            height : 12;
            width : 12;
            x : 100;
            y : 15;
         METHODS :
            notify : {
               hours_text := hours_text + 1;
               IF (hours_text > 23) THEN
                  hours_text := 0;
               ENDIF;
               }
            }

      hrs_down : XmArrowButton {
         ATTRIBUTES :
            parent : palette;
            arrowdirection : XmARROW_DOWN;
            shadowthickness : 0;
            height : 12;
            width : 12;
            x : 100;
            y : 30;
         METHODS :
            notify : {
               hours_text := hours_text - 1;
               IF (hours_text < 0) THEN
                  hours_text := "23";
               ENDIF;
               }
            }

      mins_up : XmArrowButton {
         ATTRIBUTES :
            parent : palette;
            arrowdirection : XmARROW_UP;
            shadowthickness : 0;
            height : 12;
            width : 12;
            x : 213;
            y : 15;
         METHODS :
            notify : {
               minutes_text := minutes_text + 1;
               IF (minutes_text > 59) THEN
                  minutes_text := 0;
               ENDIF;
               }
            }
            mins_down : XmArrowButton {
               ATTRIBUTES :
                  parent : palette;
                  arrowdirection : XmARROW_DOWN;
                  shadowthickness : 0;
                  height : 12;
                  width : 12;
                  x : 213;
                  y : 30;
               METHODS :
                  notify : {
                     minutes_text := minutes_text - 1;
                     IF (minutes_text < 0) THEN
                        minutes_text := “59”;
                     ENDIF;
                     }
                  }
         ENDVC set_buttons
```

Example 4-2 Slang Dialogue for the Second Stage in Developing the Sample Clock

4.2.2 External C Routines

When Serpent converts integers to strings, it does so in the most compact form possible. This means that the number “5” will be displayed as “5” in a Label object. While this is often desirable, people are used to seeing clocks display their time as “13:05” (with a leading 0), and not as “13: 5”. To get Serpent to display the time fields in the desired format, an external routine is used to explicitly convert the integer into a string, instead of using the built-in mechanism Serpent provides. The routine is declared through the `EXTERNALS` section in the dialogue, and referenced by using it in-line. The code for the routine is shown below, and may be found in the file `demo/smo/clock/threeE.c`.

```
#include “serpent.h”

string i2str (in)
int in;
{
    static char out[3];

    (void) sprintf (out, “%02d”, in);
    return out;
}
```

4.2.3 Makefile

The Makefile for the augmented dialogue is slightly more complicated than for the simple display dialogue. This is necessitated by the presence of the external routine, which has been placed in the file `twoE.c`.

```
two:      two.sl twoE.o
    serpent -cl -L twoE.o two

twoE.o:   twoE.c
    cc -c $(INCS) $<
```

4.3 Third Stage of Developing Clock

Once the dialogue and its behavior have been developed to some degree of satisfaction, the next logical step is to attach an application program to provide real functionality. In this case, the application is the program that will feed the dialogue the current time of day. Before an application can be developed, a format for communication between the dialogue and application must be defined. This is done through a Saddle description, which specifies the format of instances of shared data. Note that Saddle does not create any instances of shared data, it only defines the layout of instances that can be created by the dialogue or by the application program.

Once this is done, the application program and the dialogue can communicate by reading and writing the instances of shared data that are created at runtime. All communication is done through transactions (explicit in the application program and implicit within the Slang dialogue). Figure 4-1 (page 19) shows the interactions between the components of the running dialogue. More details on using the shared data interface can be found in the *Serpent: Ada Application Developer's Guide* (CMU/SEI-91-UG-7) and in the *Serpent: C Application Developer's Guide* (CMU/SEI-91-UG-6).

In the clock example, the application program will send the current time of day to the user, and allow the user to change the time of day using the "set" buttons of the dialogue. Because only the super-user can change the system time, the application program maintains a "user-delta," which is the amount of time that the clock is to be fast or slow. This way the user is given the appearance of changing the clock without actually doing so.

4.3.1 Saddle Description

The Saddle description which follows (and which can also be found in the file `demos/smo/clock/threeA.sdd`) is used to define the format of the data shared between (in this case) the dialogue and the application. Because this is a rather simple example, only one shared data type is defined for this interface. Naturally, more complex interfaces could have just as easily been defined.

The Saddle description starts with the command to be used to start up the application program. In this case, the application is called `threeA`, and the command line takes no special flags or arguments. The first line of the description states this. The remainder of the Saddle description defines the shared data types. In this example, a single shared data type called `sd_time` is defined. It contains two integer fields, `hrs` and `mins`, and a 20-character string called `date`. It should be noted again that the Saddle file only defines the layout of instances of shared data records (it is somewhat similar to the typedef declarations in a C include file). A Saddle description does not actually declare or create any of the instances. This latter task is left to the dialogue or the application.

```
<< threeA >>

whatever : shared data

    sd_time : record
       hrs: integer;
       mins: integer;
       date: string[20];
    end record;

end shared data;
```

In this example (and as shown later in Section 4.3.3), only a single instance of this shared data record is created. This is not a fundamental restriction of Serpent – any number of instances of a shared data record may be created.

4.3.2 Slang Dialogue

The Slang dialogue for the third stage in development of the sample clock is shown in Example 4-3. This dialogue may also be found as the file `demos/smo/clock/three.sl` in the Slang source hierarchy.

The differences between this example and the previous one are fairly small, although they are pervasive. First, the current time has been moved into a view controller that is bound to an instance of the shared data type `sd_time`. Rather than using the hardwired variables `hours_text` and `minutes_text`, the objects in the `current_time` view controller all refer to the components of the shared data element, namely `sd_time.hrs` and `sd_time.mins`. Reading from these components references the current value in shared data; writing to them automatically causes a transaction to be created (which is then processed by the application, as shown in Section 4.3.3).

The only other significant change to the dialogue is that it now includes the file `threeA.ill`, a file which defines the layout of the shared data used by both the dialogue and the application.

Note: change bars are used to show where the code in this example differs from the previous example.

```
#include “smo.ill”
#include “threeA.ill”

|||

#include “glueXm.h”

EXTERNALS :
    string i2str(integer);

VARIABLES :
    DatePlus : {
       IF (Show_Date.set) THEN
          DatePlus := 50;
       ELSE
          DatePlus := 0;
       ENDIF;
       }
    SetPlus : {
       IF (Set_Time.set) THEN
          SetPlus := 12;
       ELSE
          SetPlus := 0;
       ENDIF;
       }
    MilitaryTime : true;

OBJECTS :

    shell : XmTopLevelShell {
       ATTRIBUTES :
          height : palette.height;
          width : palette.width;
          allowShellResize : true;
          }

    palette : XmBulletinBoard {
       ATTRIBUTES :
          parent : shell;
          height : 150 + DatePlus;
          width : 375;
          }
```

```
quit : XmPushButton {
   ATTRIBUTES :
      parent : palette;
      labelstring : “Quit”;
      borderwidth : 1;
      shadowthickness : 3;
      height : 35;
      width : 40;
      x : palette.width - width - 10;
      y : palette.height - height - 10;
   METHODS :
      notify : { exit(); }
      }

FormatSelector : XmRowColumn {
   ATTRIBUTES :
      parent : palette;
      height : 100;
      width : 200;
      x : 95;
      y : 65;
      packing : XmPACK_COLUMN;
      numcolumns : 1;
      orientation : XmVERTICAL;
      borderwidth : 0;
      radiobehavior : true;
      radioalwaysone : true;
      }

Button1 : XmToggleButton {
   ATTRIBUTES :
      parent : FormatSelector;
      labelString : “12 Hour”;
      indicatortype : XmONE_OF_MANY;
      width : 75;
      height : 30;
      set : false;
   METHODS:
      toggle : { MilitaryTime := not set; }
      }

Button2 : XmToggleButton {
   ATTRIBUTES :
      parent : FormatSelector;
      labelString : “24 Hour”;
      indicatortype : XmONE_OF_MANY;
      width : 75;
      height : 30;
      set : true;
   METHODS:
      toggle : { MilitaryTime := set; }
      }
```

```
Show_Date : XmToggleButton {
   ATTRIBUTES :
      parent : palette;
      labelString : "Show Date";
      x : 200;
      y : 80;
      width : 80;
      height : 20;
      set : false;
      }

Set_Time : XmToggleButton {
   ATTRIBUTES :
      parent : palette;
      labelString : "Set";
      x : 320;
      y : 20;
      width : 40;
      height : 20;
      set : false;
      }

VC : current_time
   CREATION CONDITION : ( new("sd_time") )

   OBJECTS:
      hours : XmLabel {
         ATTRIBUTES :
            parent : palette;
            fontlist : "*-courier-bold-r-*-*-34-*";
            labelstring : {
               IF (MilitaryTime) THEN
                  labelstring := i2str(sd_time.hrs);
               ELSIF (sd_time.hrs = 0) THEN
                  labelstring := 12;
               ELSIF (sd_time.hrs <= 12) THEN
                  labelstring := sd_time.hrs;
               ELSE
                  labelstring := sd_time.hrs - 12;
               ENDIF;
               }
            recomputesize : false;
            alignment : XmALIGNMENT_END;
            height : 25;
            width : 40;
            x : 115;
            y : 15;
            }
```

```
colon : XmLabel {
   ATTRIBUTES :
      parent : palette;
      fontlist : "*-courier-bold-r-*-*-34-*";
      labelstring : ":";
      recomputesize : false;
      height : 25;
      width : 15;
      x : 155;
      y : 15;
      }

minutes : XmLabel {
   ATTRIBUTES :
      parent : palette;
      fontlist : "*-courier-bold-r-*-*-34-*";
      labelstring : i2str(sd_time.mins);
      recomputesize : false;
      height : 25;
      width : 40;
      x : 170;
      y : 15;
      }

VC : AM_PM
   CREATION CONDITION : (not MilitaryTime)

   OBJECTS :
      AM_PM : XmLabel {
         ATTRIBUTES :
            parent : palette;
            fontlist : "*-courier-bold-r-*-*-24-*";
            labelstring : {
               IF (sd_time.hrs >= 12) THEN
                  labelstring := "PM";
               ELSE
                  labelstring := "AM";
               ENDIF;
               }
            recomputesize : false;
            height : 23;
            width : 40;
            x : 215 + SetPlus;
            y : 17;
            }
   ENDVC AM_PM
```

```
VC : today
   CREATION CONDITION : (Show_Date.set)

   OBJECTS:
      today : XmLabel {
         ATTRIBUTES :
            parent : palette;
            fontlist : “*-times-bold-r-*-*-20-*”;
            labelstring : sd_time.date;
            recomputesize : false;
            borderwidth : 1;
            height : 35;
            width : 200;
            x : 85;
            y : 130;
            }
   ENDVC today

VC : set_buttons
   CREATION CONDITION : (Set_Time.set)

   OBJECTS :
      hrs_up : XmArrowButton {
         ATTRIBUTES :
            parent : palette;
            arrowdirection : XmARROW_UP;
            shadowthickness : 0;
            height : 12;
            width : 12;
            x : 100;
            y : 15;
         METHODS :
            notify : {
               sd_time.hrs := sd_time.hrs + 1;
               IF (sd_time.hrs > 23) THEN
                  sd_time.hrs := 0;
               ENDIF;
               }
            }

      hrs_down : XmArrowButton {
         ATTRIBUTES :
            parent : palette;
            arrowdirection : XmARROW_DOWN;
            shadowthickness : 0;
            height : 12;
            width : 12;
            x : 100;
            y : 30;
         METHODS :
            notify : {
               sd_time.hrs := sd_time.hrs - 1;
               IF (sd_time.hrs < 0) THEN
                  sd_time.hrs := “23”;
               ENDIF;
               }
            }
```

```
                    mins_up : XmArrowButton {
                       ATTRIBUTES :
                          parent : palette;
                          arrowdirection : XmARROW_UP;
                          shadowthickness : 0;
                          height : 12;
                          width : 12;
                          x : 213;
                          y : 15;
                       METHODS :
                          notify : {
                             sd_time.mins := sd_time.mins + 1;
                             IF (sd_time.mins > 59) THEN
                                sd_time.mins := 0;
                             ENDIF;
                             }
                          }

                    mins_down : XmArrowButton {
                       ATTRIBUTES :
                          parent : palette;
                          arrowdirection : XmARROW_DOWN;
                          shadowthickness : 0;
                          height : 12;
                          width : 12;
                          x : 213;
                          y : 30;
                       METHODS :
                          notify : {
                             sd_time.mins := sd_time.mins - 1;
                             IF (sd_time.mins < 0) THEN
                                sd_time.mins := "59";
                             ENDIF;
                             }
                          }
                 ENDVC set_buttons

           ENDVC current_time
```

Example 4-3 Slang Dialogue for the Third Stage in Developing the Sample Clock

4.3.3 Application Program

The newly created application program (found in the file `demo/smo/clock/threeA.c` in the Slang source hierarchy) is responsible for feeding the dialogue the current time of day (perhaps offset by a user-induced delta). It is also tasked with creating the shared data element that is used to communicate this information. The inclusion of the file `serpent.h` defines the routines and data types used by Serpent, while the inclusion of the file `threeA.h` defines the shared data types specified in the Saddle file.

The routine `find_time` gets the current time of day, converts it to local time, and places the current time in the `hrs` and `mins` field of the passed-in `sd_time` structure. It also uses the `sprintf` library routine and an array of month names to format the date in the standard form shown in the figures earlier in this chapter. The routine `adjust_time` is used to add (or subtract) the difference between the "real" time of day and the user-perceived version. This application program does not actually change the time of day – only the super-user can do that. Instead, the application program remembers how fast or slow the clock should be, and adjusts the visualized time accordingly.

```
#include "serpent.h"
#include "threeA.h"
#include <sys/time.h>

#define NO_CHANGE 999

void find_time (cur_time)
sd_time *cur_time;
{
    struct timeval tv;
    struct timezone tz;
    struct tm *now;
    static char *month[] = {
       "January", "February", "March", "April", "May",
       "June", "July", "August", "September", "October",
       "November", "December" };
    gettimeofday (&tv, &tz);
    now = localtime (&tv.tv_sec);
    cur_time->hrs = now->tm_hour;
    cur_time->mins = now->tm_min;
    (void) sprintf (cur_time->date, "%s %d, %d",
       month[now->tm_mon], now->tm_mday, now->tm_year+1900);
}

void adjust_time (cur_time, delta)
sd_time *cur_time, *delta;
{
    cur_time->hrs += delta->hrs;
    cur_time->mins += delta->mins;
}

main ()
{
    sd_time cur_time, from_dlg, delta;
    transaction_type trans;
    id_type id;

    serpent_init (MAIL_BOX, .ill_FILE);
    trans = start_transaction ();
    find_time (&cur_time);
    id = add_shared_data (trans, "sd_time", NULL, &cur_time);
    commit_transaction (trans);

    delta.hrs = delta.mins = 0;
```

```
        while (1) {
           sleep (5);
           find_time (&cur_time);
           from_dlg.hrs = from_dlg.mins = NO_CHANGE;
           while (trans = get_transaction_no_wait ()) {
              id = get_first_changed_element (trans);
              incorporate_changes (trans, id, &from_dlg);
              if (from_dlg.hrs != NO_CHANGE)
                 delta.hrs = from_dlg.hrs - cur_time.hrs;
              if (from_dlg.mins != NO_CHANGE)
                 delta.mins = from_dlg.mins - cur_time.mins;
              purge_transaction (trans);
              }
           adjust_time (&cur_time, &delta);
           trans = start_transaction ();
           put_shared_data (trans, id, "sd_time", NULL, &cur_time);
           commit_transaction (trans);
           }
  }
```

Example 4-4 Application Program

Finally, the `main` routine is responsible for transmitting the current time to the dialogue and reading the changes to the time made by the user. It first creates a shared data record of type `sd_time` in a transaction and references the particular shared data instance with `id`. The shared data element is initialized with the values contained in `cur_time`, which is in turn initialized with the routine `find_time`.

Once this is done, the application cycles indefinitely, waking up every five seconds to examine transactions from the dialogue and to send the current time-of-day transaction to the dialogue. The period of five seconds is a comfortable compromise between absolute, to-the-second accuracy and program efficiency.

Within the loop, the application checks to see if any transactions have been sent from the dialogue.[5] If there have been any, the application processes them by successively calling `get_transaction_no_wait` and other interface routines. Once the user-induced delta to the current time of day has been adjusted, the dialogue transmits the current (potentially adjusted) time of day back to the dialogue in its own transaction. The dialogue automatically reads the transactions, and the (potentially updated) time of day is displayed on the screen.

[5] A transaction from the dialogue happens automatically whenever the dialogue changes a value in the shared data element bound to the `current_time` view controller.

4.3.4 Makefile

The Makefile for the dialogue with an associated application program is only slightly more complicated than before. The dialogue portion of the Makefile is essentially the same (in fact, the external C routine used is identical to the second stage). All that is new is the instructions needed to compile the Saddle description and the application program.

```
LIBS= $(LIBDIR)/libint.a $(LIBDIR)/libutl.a \
       $(LIBDIR)/liblist.a -lm

.SUFFIXES:.ill .sdd

.sdd.ill .sdd.h:
    sdd $<

three:    three.sl threeA.ill twoE.o
    serpent -cl -L twoE.o three

twoE.o:   twoE.c
    cc $(CFLAGS) -c $(INCS) $<

threeA:   threeA.o
    cc $(CFLAGS) -o $@ $? $(LIBS)

threeA.o: threeA.c threeA.h
    cc $(CFLAGS) -c $(INCS) $<
```

4.4 Fourth Stage of Developing Clock

One of the great strengths of Serpent is the ability to test different user interfaces (i.e., dialogues) without changing the application program. The fourth stage in the development of the clock example demonstrates this ability quite nicely. Figure 4-7 shows a completely

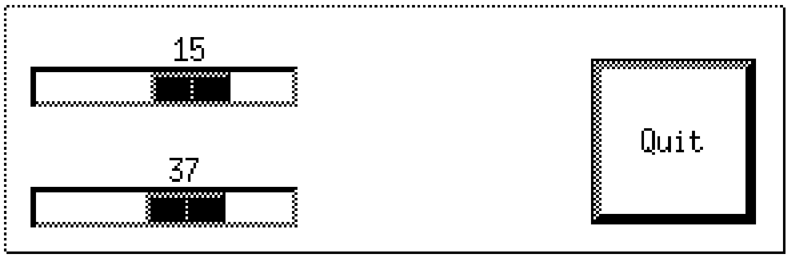

Figure 4-7 – Alternative Interface for Clock

different clock interface. The time of day is represented as a pair of sliders, with the top slider being hours, and the bottom slider being minutes. The time shown, therefore, is 15:37. Moving either of the sliders to the left or right will adjust the current time of day in the same way as the “Set” buttons did in the previous example.

It is important to note that *no changes* were made to the application program shown in Section 4.3.3 (page 42). So long as the interface is unchanged, the dialogue may take any form at all.

4.4.1 Slang Dialogue

The slang dialogue for the alternative dialogue is shown in Example 4-5 (it may also be found in `demo/smo/clock/four.sl` in the Slang source hierarchy). This example quickly and simply shows a different user interface to the clock application program.

The creation of a new `sd_time` shared data instance causes a view controller to be created, as before. In this case, however, to-scale widgets are used to represent the values of `hrs` and `mins` within the instance. Changing either value by moving the slider causes the `value_changed` method to be invoked, which causes the value in the shared data instance to be changed. This creates a transaction that is picked up by the application program.

```
#include “smo.ill”
#include “threeA.ill”

|||

#include “glueXm.h”

OBJECTS :

    palette : XmBulletinBoard {
       ATTRIBUTES :
          height : 100;
          width : 300;
          }

    quit : XmPushButton {
       ATTRIBUTES :
          parent : palette;
          labelstring : “Quit”;
          borderwidth : 1;
          shadowthickness : 3;
          height : palette.height - 40;
          width : height;
          x : palette.width - (width + 20);
          y : palette.height - (height + 20);
       METHODS :
          notify : { exit(); }
          }
```

```
VC : current_time
   CREATION CONDITION : ( new(“sd_time”) )

   OBJECTS:
      hours : XmScale {
         ATTRIBUTES :
            parent : palette;
            minimum : 0;
            maximum : 23;
            value : sd_time.hrs;
            showvalue : true;
            orientation : XmHORIZONTAL;
            processingDirection : XmMAX_ON_RIGHT;
            height : 15;
            width : 200;
            x : 10;
            y : 5;
         METHODS :
            value_changed : { sd_time.hrs := value; }
            }

      minutes : XmScale {
         ATTRIBUTES :
            parent : palette;
            minimum : 0;
            maximum : 59;
            value : sd_time.mins;
            showvalue : true;
            orientation : XmHORIZONTAL;
            processingDirection : XmMAX_ON_RIGHT;
            height : 15;
            width : 200;
            x : 10;
            y : 55;
         METHODS :
            value_changed : { sd_time.mins := value; }
            }

   ENDVC current_time
```

Example 4-5 Slang Dialogue for the Alternative Dialogue

4.4.2 Makefile

All that is different in the Makefile for the fourth and final stage in the clock development are the dialogue building instructions. The application program does not change, so the Makefile does not change here. Also, since the dialogue no longer needs an external C routine to do formatting, the file `twoE.c` is no longer referenced.

```
.SUFFIXES:.ill .sdd

.sdd.ill .sdd.h:
    sdd $<

four:  four.sl threeA.ill
    serpent -cl four
```

Index

A
Ada 9, 17
Application program 17
Athena 8
Attribute list 7

C
C routines 34, 37
C tags 9
Creation condition 27, 32, 33, 39, 40, 41, 47

D
Debugging 10, 18
Dialogue editor 8
Dialogue manager 8, 14
Directory hierarchy 7
Documentation 2, 3, 8

E
Environment variables 5, 18
External C routines 34, 37
External definitions file 6

F
File names 6

G
Glue 16
Glue compiler 8, 16
Glue file 7

I
ILL file 7, 12, 14, 18
IMAKEINCLUDE 6
Installing Serpent 4
Interprocess communication 4, 18
Invisible files 6

L
Linker 12

M
Message queues 18
Method description 7
Motif 8

O
OpenLook 8

S
Saddle compiler 10, 11
Saddle source file 6
Sat 8
Semaphores 18
Serpent
  documents 2
Serpent command 18
SERPENT_BASE_PATH 5
SERPENT_DATA_PATH 5, 9
SERPENT_EXE_PATH 5, 18
Shared data descriptions 6
Shared memory 18
Six 8, 10, 15
Slang compiler 7, 12
Slang source file 6
Smo 8
Sol 8

T
Tags 9
Technology file 7
Technology interface 9

X
Xt-based toolkits 16

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

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-2	5. MONITORING ORGANIZATION REPORT NUMBER(S) CMU/SEI-91-UG-2

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: System Guide

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	April 1991	

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

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

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 document introduces the environment variables used by Serpent, the file naming conventions and expected file types, and how to build a Serpent dialogue/application from scratch.

(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)

ABSTRACT —continued from page one, block 19

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