Chapter

13

Using the Telelogic Tau Public Interface

This chapter contains a number of examples of how the use the features provided by the Telelogic Tau Public Interface.

For a reference to the topics that are exemplified in this chapter, see *chapter 12, The Telelogic Tau Public Interface*.

Introduction

The first example shows a simple application, the *Service Encapsulator*, by which it is possible to request Telelogic Tau tools services from the Operating Systems command line. The example shows how to use the PostMaster interface.

The second example uses the Service Encapsulator and shows the usage of some SDL suite Services. Since the "DOS" command prompt is rather weak in its scripting capabilities, no example script file is included **for Windows** in this example.

The third example uses the Service Encapsulator and shows the usage of some TTCN Suite Services.

The final example exemplifies how to integrate an SDL simulator and a separate user interface **(UNIX only)**. How to design such a user interface is described in a detailed example. It is assumed that the reader has experience of creating and running SDL simulators.

The Service Encapsulator

Introduction to the Service Encapsulator

This section shows an application, the Service Encapsulator, implementing a command line service request encapsulation. That is, Telelogic Tau services as made available by the Telelogic Tau Public Interface, could be obtained from the command line.

The purpose of this example is to show:

- How the PostMaster interface is used when working with Telelogic Tau services.
- By using the Service Encapsulator, show the usage of a number of services.

The application is general in the sense that it does not require a Telelogic Tau tool to be running (only the PostMaster is required to be running), but in this context it exemplifies a number of Telelogic Tau services.

From the outside world the application behaves like a tool providing *remote procedure call* functionality. That is, when the service is request-

ed, the application synchronously waits for the service reply, before returning (the application exits).

The example is found in the Telelogic Tau distribution in the subdirectory `examples` and then `public_interface`.

The Service Encapsulator is also available as a tool in the Telelogic Tau environment, see *"The Service Encapsulator" on page 537 in chapter 12, The Telelogic Tau Public Interface*.

Design

In this section is a few important aspects in the design and implementation of the *Service Encapsulator* given. The source file is named `serverpc.c` and the executable is named `serverpc`.

The tool connects to the PostMaster by issuing:

```
SPInit(SET_EXTERN, argv[0], spMList) (in SDL suite)
SPInit(IET_EXTERN, argv[0], ipMList) (in TTCN suite)
```

The first parameter gives the type of application. In this case `SET_EXTERN` or `IET_EXTERN` is used. This application type is a predefined tool type for external applications connecting to Telelogic Tau. The variable `spMList` or `ipMList` gives the possibility to convert between textual (symbolic) values and integers.

If `SPInit` succeeds, `SPInit` broadcasts a `SESTARTNOTIFY`, indicating to all other tools in the environment that an additional tool is connected.

```
if ((tool=atoi(argv[1])) == 0 )
  tool = SPConvert(argv[1]);
```

If the parameter `<tool>` was given textually, the function `SPConvert` converts it to the corresponding integer value. If a mapping cannot be found, -1 is returned. This is an error and will be detected in the `SPSendToTool` function, see below.

```
if ((sendEvent=atoi(argv[2])) == 0 )
   sendEvent = SPConvert(argv[2]);
```

Allocating memory for the message parameter is done as follows:

```
for (i=3;i<argc;i++) {
  p = realloc(p, strlen(p) + strlen(argv[i]) + 2);
  if (strlen(p)>0)
    p = strcat(p, " ");
  p = strcat(p, argv[i]);
}
```

The `handleescape` function allows carriage return "`\n`", "`\r`" and "`\0`" to be used when sending messages:

```
p = handleescape(p);
```

A normal service request is issued by:

```
status=SPSendToTool(tool, sendEvent, p, strlen(p)+1;
```

The last parameter tells the length of the parameter provided by `argv[3]` and must be extended by 1 to include the terminating `\0` character.

Now the service request message is sent to the PostMaster which forwards it to the server application. If not busy, the Service application starts to process the service request and when finished it replies with a service reply message to the issues. However, during the processing other messages can be sent or broadcast to the *Service Encapsulator* (the service issuer). Therefor we must enter a loop where we stay until the reply message is received. Other kind of messages are dropped.

```
do {
   if (( status=SPRead(SPWAITFOREVER, &pid,
         &replyEvent,(void*)&replyMessage, &len))!=0
)
      OnError("Error reading from postmaster: ");
} while ( ! SP_MATCHREPLY(sendEvent, replyEvent) );
```

The function `SPRead` reads a message form the PostMaster. The first parameter, the timeout, could be set to wait indefinitely until a message arrives, using the value `SPWAITFOREVER`.

This simple application has a drawback: if something unexpectedly happens to the server, it might be that no service reply will be sent which means that we block in `SPRead` forever. The behavior could be enhanced by setting a timeout, and when the timeout expires, check the state of the server (Use the PostMaster service Get Tool Pid).

If the service `readattribute` was issued, special care must be taken to handle the answer, since it contains binary data. The length of the data part ends the `replyMessage`. The end is indicated by an ASCII 0. The

data part immediately follows the ASCII 0 character and might contain non-ASCII characters. Our printout assumes ASCII data.

```
if ( sendEvent == SEREADATTRIBUTE ) {
   printf("%s", replyMessage+1);
  printf("%s\n", replyMessage+ strlen(replyMessage)
          +1);
}
```

The macro `SP_MATCHREPLY` compares the issued service request message and the reply message and returns true if the correct received message was the correct service reply message.

The service reply is then printed on standard output. The following prints the service reply data:

```
int i;
for( i=2;i<len;i++ )
  putchar(replyMessage[i]);
putchar('\n');
```

Counting starts at position 2, omitting the service reply status.

```
free (replyMessage);
```

`replyMessage` was dynamically allocated using malloc in `SPRead`, so it must be freed.

Before terminating the application, notify all other applications in the environment that it is going to break its connection to the PostMaster

```
SPBroadcast(SESTOPNOTIFY, NULL, 0);
```

Finally we terminate the PostMaster connection by calling:

```
SPExit ();
```

Generating the Application for the SDL Suite

To generate the application do the following:

1. Change the current directory to the sub directory containing the example:

   On UNIX: `cd <Installation Directory>/examples/public_interface`

   In Windows: `cd <Installation Directory>\examples\public_interface`

2. Set the environment variable `telelogic` to point to the right directory. This is platform dependent.

   On UNIX: `setenv telelogic <Installation Directory>`

   In Windows: `set telelogic=<Installation Directory>`

3. Generate the application:

   On Solaris: `make -f Makefile.solaris`

   On HP: `make -f Makefile.hp`

   In Windows using Borland: `make -f Makefile.borland`

   In Windows using Microsoft: `nmake /f Makefile.msvc`

Using the SDL Suite Services

Introduction

This section exemplifies the usage of some SDL suite services. The examples take advantage of the *Service Encapsulator* described in the previous section.

In these examples, we only use textual values of the parameters `<tool>` and `<service>`. Available textual values is found in the `spMList` variable in `sdt.h`.

In the examples below, the *Service Encapsulator* is executed from a `csh` shell **(on UNIX)**, or from the "DOS" prompt **(in Windows)**. If another shell is used, the examples below might need to be modified in order to supply the parameters correctly. Take care how to supply the quoted strings.

> **Note:**
>
> Due to limitations in DOS, this Service Encapsulator example is not as extensive in Windows as the UNIX version.

Load External Signal Definitions into the Information Server

For this service to be available, the Information Server must be started. This could be done by either starting the Type Viewer from the Orga-

nizer, by requesting *Signal Dictionary* support in the SDL Editor or by using the *Start service.* For a complete service description, See *"Start Tool" on page 539 in chapter 12, The Telelogic Tau Public Interface*.

Requesting the Service

It is assumed that a file `a.pr` is to be loaded. Note that the file to be loaded must be specified with a full directory path.

On UNIX: `serverpc info loaddefinition /usr/ab/a.pr`

In Windows: `serverpc info loaddefinition c:\sdt\ex\a.pr`

If the Information Server is started, the following reply message is returned:

```
Reply status is OK
```

If the Information Server is not started, we get the following reply message.

```
Error sending to postmaster:  server not connected
to postmaster
```

The external files loaded into the SDL Infoserver should have the following appearance e.g.:

```
signal sig1
signal sig2
signal sig3
```

Obtain Source (SDL/GR Reference)

Returns a complete SDT reference of each of the selected symbols in the specified editor.

```
serverpc sdle obtainsource
```

If the editor was already started and there was a selection the reply message might look like:

```
Reply status is OK
 1
"#SDTREF(SDL,c:\Telelogic\SDL_TTCN_Suite4.5\sdt\exam
ples\demongam\demon.spr(1),125(30, 70))"
```

Note that the reference returns a complete file path of the diagram file in which the selection was made.

If the editor did not contain any selection, the reply becomes:

```
Reply status is OK
0
```

For a complete description of the service, see *"Obtain GR Reference" on page 597 in chapter 12, The Telelogic Tau Public Interface*.

Show Source

Selects the object given by the parameters in an editor. The editor is started if necessary as a side effect. A system must however be opened in the Organizer containing the specified reference.

To test this service we could now deselect all selections in the SDL Editor and use the reference extracted by `obtainsource` to select this symbol again. Note how the SDT reference is quoted to pass it from the shell into the tool.

On UNIX:
```
serverpc organizer showsource \
  """#SDTREF(SDL,/usr/sdt-inst/examples/simulator- \
  integration/sunos4/demon.spr(1),125(30,70))"""
```

In Windows:
```
serverpc organizer showsource
\"#SDTREF(SDL,c:\Telelogic\SDL_TTCN_Suite4.5\sdt\exa
mples\demongam\demon.spr(1),125(30,70))\"
```

If the reference is found the following message is replied:

```
Reply status is OK
```

For a complete description of the service, see *"Show Source" on page 596 in chapter 12, The Telelogic Tau Public Interface*.

For a complete description of how to specify a SDT reference, see "Syntax" on page 911 in chapter 19, *SDT References*.

Dynamic Menus

The following example shows how a dynamic menu is created and how a number of menu items are inserted in the dynamic menu. The script `dyn-menu` in the example directory performs the example below.

Note:

In Windows, this example is intended for Windows NT/2000. If you are running Windows 95/98, all references below to `CMD` should be changed into `COMMAND.COM`

For a complete description of the services, see "Add Menu" on page 579, "Delete Menu" on page 580, "Clear Menu" on page 581 and *"Add Item to Menu – Graphical Editors" on page 591 in chapter 12, The Telelogic Tau Public Interface*.

Add a new menu to the SDL Editor:

```
serverpc sdle menuadd \"dyn-menu\"
```

Reply:

```
Reply status is OK
```

Add a menu item which executes the OS command `ls` **(on UNIX)** or `DIR` **(in Windows)** after a confirmation from the user:

On UNIX:
```
serverpc sdle menuadditem  \
\"dyn-menu\" \"ls\" 0 \
\"Perform OS command ls\"  0 0 0 \
\"OK to perform ls!\" 1 1 \"ls\"
```

In Windows:
```
serverpc sdle menuadditem \"dyn-menu\"
\"DIR\" 0 \"Perform OS command DIR\" 0 0 0 \"OK to
perform OS command DIR!\" 1 1 \"CMD /C DIR /W\"
```

Reply:

```
Reply status is OK
```

Then a menu item is added displaying the SDT reference of the selected symbol on standard output **(on UNIX)**, or in the Organizer Log window **(in Windows)**:

On UNIX:
```
serverpc sdle menuadditem  \
\"dyn-menu\" \"SDT-ref\" 1 \
\"export SDT ref\"  0 0 1 \
\"\" 1 1 \"echo \'%g\'\"
```

In Windows:
```
serverpc sdle menuadditen \"dyn-menu\"
\"SDT-ref\" 1 \"export SDT-ref\" 0 0 1 \"\" 1 1
\"CMD /C echo %%g\"
```

Reply:

```
Reply status is OK
```

Note that only dimmed if more than one item is selected. Normally a selected symbol includes a selection of in-going and out-going flowlines. These flowlines must be de-selected in order to get the SDT reference of the symbol.

Finally display the file containing the selected symbol on standard output **(on UNIX)**, or in the Organizer Log window **(in Windows)**:

On UNIX:
```
serverpc sdle menuadditem  \
\"dyn-menu\" \"filename\" 0 \
\"export filename\"  0 0 1 \
\"\" 1 1 \"ls -ls %a\"
```

In Windows:
```
serverpc sdle menuadditem \"dyn-menu\"
\"filename\" 0 \"export filename\" 0 0 1 \"\" 1 1
\"CMD /C DIR %%a\"
```

Reply:

```
Reply status is OK
```

Extended Object Data Attributes (UNIX only)

The SDL suite allows external attributes to be added to SDL suite objects. These extensions are persistent. That is, they are stored in the normal SDL diagram files and could be used at subsequent sessions. This **UNIX only** example gives a brief introduction to such extended attributes.

An description of extended attributes are found in "Extended Data Attribute" on page 615 in chapter 12, *The Telelogic Tau Public Interface*. Extended attributes are handled as binary data by the SDL suite. As such, their services are preferably accessed via C program interface, in which binary data is easily handled.

However, in this example extended attribute services are accessed via the *Service Encapsulator*, which works for simple examples.

The example assumes one selection in an SDL Editor. This reference is saved in a shell variable

```
set ref=`serverpc sdle obtainsource`
```

If we have exactly one selection, we set the extended attribute. Parameter 9 will give us the SDT reference.

```
serverpc sdle createattribute 0 0 $ref[9] 0 \
  \"myComment\" \"\"  6\"\0\"MyData
```

Note that the length of data (`MyData)` must be manually calculated and that the data part must be preceded by a “\0”.

We get the reply:

```
Reply status is OK
```

Now we extract the extended attribute on the selection.

```
serverpc sdle readattribute 0 0 $ref[9]
```

Which replies:

```
Reply status is OK
0 “myComment” “” 6MyData
```

Note that data immediately follows its’ length without any spaces. In the reply message, there is a ASCII 0 character after the length, preceding the data part.

Then we update the attribute by;

```
serverpc sdle updateattribute 0 0 $ref[9] 0
\”newComment\” \”\” 7”\0”NewData
```

Reply:

```
Reply status is OK
```

Finally we read the extended attribute and should receive the updated value.

```
serverpc sdle readattribute 0 0 $ref[9]
```

Which replies:

```
Reply status is OK
0 “NewComment” “” 7NewData
```

The script `extended-attributes` performs the above described actions.

Using TTCN Suite Services

This section is meant to exemplify the usage of a few TTCN suite services. The examples are made using the Service Encapsulator described in a previous section. For the complete list of supported services, see "ITEX Services" on page 554 in chapter 12, *The Telelogic Tau Public Interface*.

Opened Documents

This command retrieves a list of information about the opened TTCN documents.

```
serverpc itex openeddocuments
```

This will yield a list of open TTCN documents (not necessarily shown) that looks like this (see complete list for description of the list format):

```
Reply status is OK
Doc1:1:/path/doc1.itex:/path/#doc1.itex
Doc2:2:/path/doc2.itex:/path/#doc2.itex
```

If the TTCN environment is not yet started the message will be:

```
Error sending to postmaster: server not connected to
postmaster
```

Find Table

To use this command the TTCN document in which the table is located must be open.

To invoke the service the following Service Encapsulator command is made:

```
serverpc itex findtable 1 \"Foo\"
```

which will search for the table named `Foo` in the open TTCN document with buffer identifier 1 and open it the Table Editor.

If the table is found the table will be shown and the following message will be returned:

```
Reply status is OK
```

If the table can not be found, an error message is issued:

```
Unable to find object 1 Foo
```

Integrating Applications with SDL Simulators

An example of utilizing the functionality of the PostMaster is to connect a simulator generated by the SDL suite to another application, typically a user interface (UI). The section "Example of Use (UNIX only)" on page 688 presents a detailed description of such an example **(UNIX only)**. The sections below concentrate on overall design issues and serves as an introduction to that example.

The communication between an SDL simulator and another application is handled by the PostMaster, and can be seen as occurring on two levels:

- Sending and receiving SDL signals
- Sending and receiving PostMaster messages, containing the SDL signals

These two levels of communication are described further below.

Transferring SDL Signals

A simulator communicates with the world outside by sending and receiving SDL signals to/from its environment. For another application to communicate with the simulator, it must also be able to interpret those SDL signals, and to send and receive them. This includes mapping the information contained in the SDL signal (name and parameters) to components and actions in the UI, e.g. invoking a command or changing the contents of an output field.

To ensure successful communication with an SDL simulator, the SDL signal interface to the environment should be designed with regard to the connected applications. Decisions made when designing a UI can influence the design of the simulator interface, and vice versa. It is important to have a clearly defined signal interface to the applications that will communicate with the simulator.

Transferring PostMaster Messages

The PostMaster communicates with different tools by sending messages of a defined format. The SDL signals must therefore be transformed to PostMaster messages before they can be transferred between the tools. A few predefined PostMaster messages are available for the purpose of handling SDL signals.

Each tool designed to communicate by using SDL signals must have an interface to the PostMaster that handles the transformation to and from an appropriate message.

In the case of an SDL simulator, this interface is generated automatically. To invoke the transformation, so that SDL signals to and from the environment are transferred using the PostMaster, the monitor command Start-SDL-Env is to be used.

For other tools, the interface to the PostMaster must be implemented separately. The interface must use functions described later in this section to connect to the PostMaster, and to send and receive PostMaster messages containing SDL signals. This includes packing the information contained in the SDL signal (name and parameters) so that it can be sent using a PostMaster message, and unpacking the same information of a received message. See *"Input and Output of Data Types" on page 2069 in chapter 50, The SDL Simulator* for more information on suitable data formats.

Example of Use (UNIX only)

This section describes an example of how to connect a user interface (UI) to an existing SDL simulator. The simulated system used in the example is the well-known Demon game, described in "The Demon Game" on page 41 in chapter 3, *Tutorial: The Editors and the Analyzer, in the SDL Suite Getting Started*. All necessary code for the example is provided with the release.

The DemonGame simulator is connected through the PostMaster to a control program, consisting of two parts; an interface to the PostMaster and a user interface (see Figure 166).

The PostMaster interface establishes a connection to the PostMaster and communicates with the DemonGame simulator. Messages are sent via the PostMaster to the simulator where they are seen as SDL signals coming from the environment. Correspondingly, SDL signals to the en-

vironment are sent via the PostMaster as messages back to the PostMaster interface.

The UI facilitates sending and receiving of the SDL signals sent to and from the environment in the Demon game. A command-based UI (written in C) is implemented. However, the ideas are general and could be used if a graphical user interface is to be used. The UI must be able to forward the SDL signals Newgame, Endgame, Probe and Result, and to present the SDL signals Win, Lose and Score.

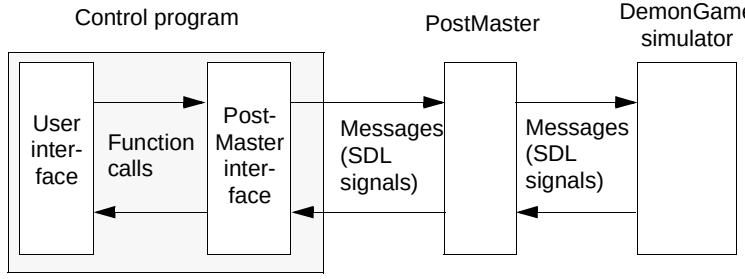

Figure 166: System parts and their communication

Note:

The simulator and the control program have the **same communication interface** to the PostMaster. The simulator's interface to the PostMaster is **generated** automatically by the SDL suite.

All C functions, C preprocessor symbols and PostMaster messages used in this example are described in "PostMaster Reference" on page 490 in chapter 11, *The PostMaster*.

The PostMaster Interface

All communication with the DemonGame simulator is handled by the interface to the PostMaster, a C program called `env.c`. This program contains the functions `Init`, `Send_to_PM`, `Receive` and `Exit_PM`, all of which must be called by the UI. Please note that the error handling in this program is very simple and does not comply with the design recommendations mentioned above.

Functions in env.c

- `Init` establishes a connection to the PostMaster by calling `SPInit`, identifying this tool as an SDL environment. The function broadcasts the message `SESTARTNOTIFY` to inform other tools that this tool has started. Finally it calls `Init_UI`, a function in the UI.
- `Send_To_PM` takes the name of an SDL signal as parameter and broadcasts this signal as an `SESDLSIGNAL` message to the simulator.
- `Receive` calls `SPRead` to wait for messages from the PostMaster, and acts suitably:
  - If the message contains an SDL signal from the simulator (Score, Lose or Win), it is passed on to the UI by calling `Send_To_UI`, a function in the UI to present the receipt of the signal. It also returns true in these cases.
  - If the message is a request to stop executing (`SESTOP`), it is accepted by calling `Exit_UI`, a function in the UI that in turn calls `Exit_PM`.
  - If the message is `SEOPFAILED`, the PostMaster has failed to handle a message properly. You should then decode the message's parameters by calling
- `Exit_PM` broadcasts the message `SESTOPNOTIFY` and breaks the connection to the PostMaster by calling `SPExit`

Command-Based User Interface

The command-based UI prompts for and recognizes the commands "Newgame", "Endgame", "Probe", and "Result", corresponding to the SDL signals. When any of the SDL signals Score, Win or Lose are received, this information is printed. The command "Exit" exits the program.

The C program for the command-based UI is called `command.c`. It contains the functions:

- `main`
- `GetInput`
- `Appl_Receive`
- `Init_UI`
- `Exit_UI`
- `Send_To_UI`

Functions in command.c

- `main` calls `Init` in the PostMaster interface, prompts for input and examines the reading state of the file descriptors for the keyboard and the PostMaster message port. If input comes from the keyboard, `GetInput` is called and any valid SDL signal is sent by calling `Send_To_PM` in the PostMaster interface. If input comes from the message port, the message is read by calling `Appl_Receive`.
- `GetInput` reads the keyboard and compares the entered string with the available commands. If it is an SDL signal, the function returns true. If "exit" is entered, `Exit_UI` is called.
- `Appl_Receive` simply calls `Receive` in the PostMaster interface. It is included to make the functional interfaces equivalent for both user interfaces.
- `Init_UI` sets a file descriptor for the PostMaster message port and initializes unbuffered terminal output.
- `Exit_UI` calls `Exit_PM` in the PostMaster interface and exits the program.
- `Send_To_UI` prints the name of the received signal and any signal parameters on the terminal.

The figures below show how the different functions and other parts in the DemonGame UI communicate with each other.

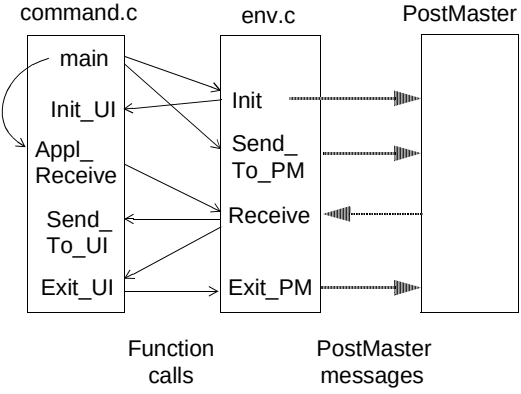

Figure 167: Communication diagram

Running the Example

To be able to build and execute the example program, the software provided on the distribution media must first be properly installed.

Building the Tools

This section describes how to build the tools that will communicate via the PostMaster, i.e. the DemonGame simulator and the UI to the simulator.

The tools are provided in source code form and in binary form for the architecture on which you have purchased the SDL suite.

The DemonGame Simulator

In the original implementation of DemonGame provided on the Telelogic Tau CD (in the `examples/demongame` directory), a design error has deliberately been introduced into the game definition. See *"Dynamic Errors" on page 153 in chapter 4, Tutorial: The SDL Simulator, in the SDL Suite Getting Started* for more information.

For the example program to execute correctly, this error must be corrected. New diagrams for the block GameBlock (`gameblock.sbk`) and the process Main (`main.spr`) are therefore provided, and are to be used to generate a new simulator. You can find the necessary files in the directory named `sdt/examples/simulatorintegration`.

1. Either replace the above diagrams in the original DemonGame implementation, or copy the DemonGame diagrams to a new directory.
2. Re-generate the simulator from the Organizer in the usual way.
3. Change to the directory containing the files `env.c`, `command.c`, and `makefile`.
4. Build the application with the UNIX `make` command. An executable named `command` is now created.

Executing a Session

This section describes how to start the communicating tools and initiate a session.

1. Make sure you have the SDL suite running, so that an instance of the PostMaster is started.
2. Start the newly generated DemonGame simulator from the Organizer.
3. Give the following commands to the simulator, preferably by including the provided command file `init.sim`.

   `Start-SDL-Env`

   Start handling of the SDL environment. See *"Start-SDL-Env" on page 2110 in chapter 50, The SDL Simulator*.

   `Set-Trace 1`

   Restrict trace output to show only signals sent to and from the SDL environment.

   `Go`

   Start executing the simulation.
4. Start the command-based UI (`command`) from the UNIX prompt.
5. Send SDL signals from the UI by entering the signal name or pushing the corresponding button. (See the description earlier in this section.) Note the trace output in the simulator and the output in the UI.
6. Exit by entering "exit" or using the *File* menu.

Chapter 13 Using the Telelogic Tau Public Interface