Chapter

# 4

# *Tutorial: The SDL Simulator*

**The SDL Simulator is the tool that you use for testing the behavior of your SDL systems. In this tutorial, you will practice "hands-on" on the DemonGame system.**

**To be properly assimilated, this tutorial therefore assumes that you have gone through the exercises that are available in chapter 3, *Tutorial: The Editors and the Analyzer*.**

**In order to learn how to use the Simulator, read through this entire chapter. As you read, you should perform the exercises on your computer system as they are described.**

# Purpose of This Tutorial

The purpose of this tutorial is to make you familiar with the essential simulation functionality in the SDL suite. Typically, simulation means executing the system under user control; stepping, setting breakpoints, examining the system, processes and variables, sending signals and tracing the execution, as you would do with a debugger, but applied on the SDL domain.

This tutorial is designed as a guided tour through the SDL suite, where a number of hands-on exercises should be performed on your computer as you read this chapter.

We have on purpose selected a simple example that should be easy to understand. It is assumed that you have a basic knowledge about SDL — this chapter is **not** a tutorial on SDL.

It is assumed that you have performed the exercises in chapter 3, *Tutorial: The Editors and the Analyzer* before starting with the tutorial on the simulator.

### Note: C compiler

You must have a C compiler installed on your computer system in order to simulate an SDL system. Make sure you know which C compiler(s) you have access to before starting this tutorial.

### Note: Platform differences

This tutorial, and the others that are possible to run on both the UNIX and Windows platform, are described in a way common to both platforms. In case there are differences between the platforms, this is indicated by texts like “on UNIX”, “Windows only”, etc. When such platform indicators are found, please pay attention only to the instructions for the platform you are running on.

Normally, screen shots will only be shown for one of the platforms, provided they contain the same information for both platforms. This means that **the layout and appearance of screen shots may differ** slightly from what you see when running the SDL suite in your environment. Only if a screen shot differs in an important aspect between the platforms will two separate screen shots be shown.

# Generating and Starting a Simulator

Once you have designed and analyzed a complete SDL system, it is possible to simulate the system, i.e. to interactively inspect and check its actual behavior. To be able to simulate the DemonGame system, you must first generate an executable simulator and then start the simulator with a suitable user interface.

**Note:**

In order to generate a simulator that behaves as stated in the exercises, you should use the SDL diagrams that are included in the Telelogic Tau distribution instead of your own diagrams. To do this:

- **On UNIX:** Copy all files from the directory `$telelogic/sdt/examples/demongame` to your work directory `~/demongame`.
- **In Windows:** Copy all files from the directory `C:\Telelogic\SDL_TTCN_Suite4.5\sdt\examples\demongame` to your work directory `C:\Telelogic\SDL_TTCN_Suite4.5\work\demongame`.

If you generate a simulator from the diagrams that you have created yourself, the scheduling of processes (i.e. the execution order) may differ.

If you choose to copy the distribution diagrams, you must then reopen the system file `demongame.sdt` from the Organizer.

## What You Will Learn

- To generate an executable simulator
- To start the simulator user interface
- To start a simulator from the user interface

## Generating the Simulator

To generate an executable simulation program, do as follows:

1. Make sure the system diagram icon is selected in the Organizer.
2. Select the *Make* command from the *Generate* menu. The Make dialog is opened:

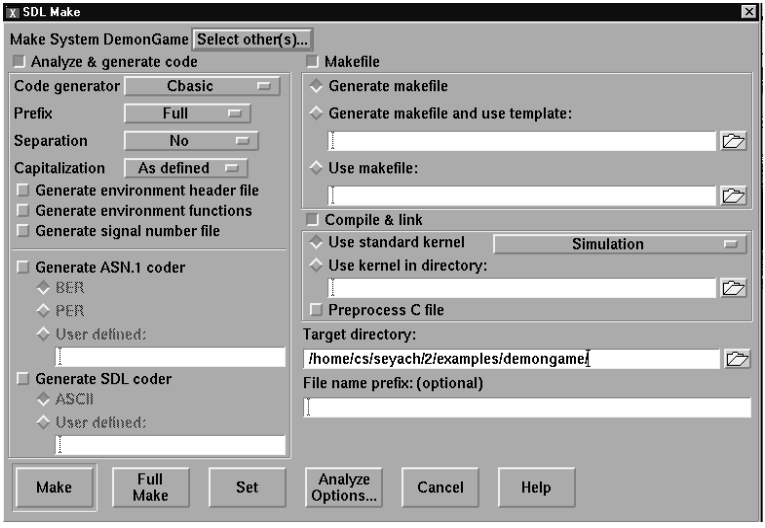


*Figure 95: The Make dialog* **(on UNIX)**

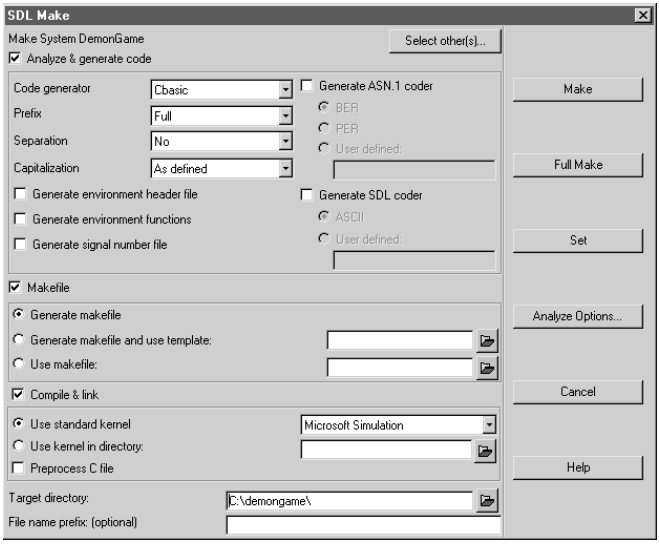


*Figure 96: The Make dialog* **(in Windows)**

3. Adjust the options in accordance to the figure, i.e.
   - *Analyze & generate code* on
   - *Makefile* and *Generate makefile* on
   - *Compile & link* on
   - *Use standard kernel* on. Make sure that a *Simulation* kernel is specified to the right; if not, select it from the option menu.
4. Click the *Make* button.
5. Select the *Organizer log* from the *Tools* menu. Check that no errors occurred. The Organizer's status bar should read "Analyzer done" and the Organizer Log should report no errors between the "Make started" and "Make completed" messages.
6. If errors were reported, bring up the Make dialog again, but click the *Full Make* button instead. This time, no errors should be reported.
   - Another problem could be with the C compiler used on your system. If you still receive errors, try changing to a Simulation kernel reflecting your C compiler, e.g. *gcc-Simulation* or *Microsoft Simulation*, and repeat the Make process.

## Starting the Simulator

The generated simulator is now stored on a file called `demongame_xxx.sct` **(on UNIX)** or `demongame_xxx.exe` **(in Windows)** in the directory from which you started the SDL suite (the `_xxx` suffix is platform or kernel/compiler specific). The simulator contains a *monitor system* that provides a set of commands which can be used to control and monitor the execution of the simulator.

It is possible for you to execute the simulator directly from an OS prompt, in which case you have to enter all commands to the monitor system textually using a simple command-line interface.

The SDL suite provides a user-friendly graphical interface to the simulator that is started from the Organizer.

1. From the *Tools* menu, select the sub-menu *SDL* and the command *Simulator UI*. The Simulator UI window is opened:

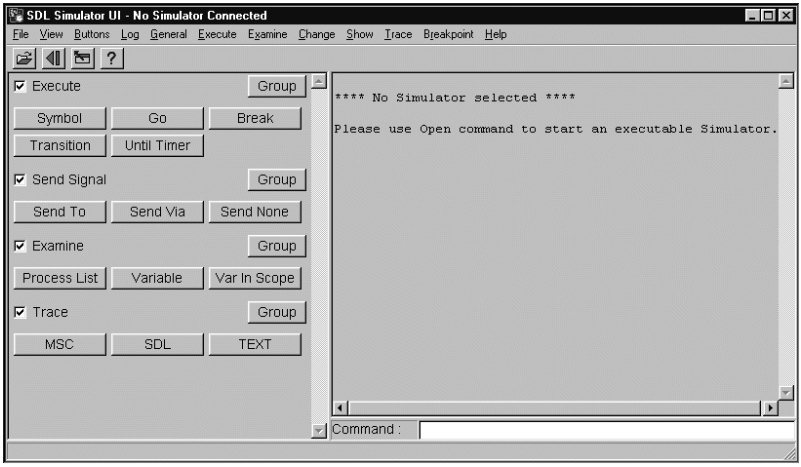


*Figure 97: The main window of the Simulator UI* **(on Windows)**

The text area to the right informs you that no simulator is running. The text area displays the textual input/output from the monitor system, such as entered commands, the results of commands, and error messages.

2. To start a simulator, select *Open* from the *File* menu, or click the *Open* quick button.

3. In the file selection dialog, the generated simulator file described above should be listed. Select it and click *OK* or *Open*.

4. The text area of the Simulator UI greets you with a welcome message to acknowledge that the simulator has been started:

```
Welcome to SDL SIMULATOR. Simulating system Demongame.
```

When a simulator is started, the static process instances in the system are created (in this case Main and Demon), but their initial transitions are not executed. The process in turn to be executed is the Main process.

The Simulator UI is now ready to accept commands to the monitor system. Whenever it is possible to enter a command, the prompt `Command:` is issued in the text area.

# Executing Transition by Transition

## What You Will Learn

- To enter commands textually
- To show the next symbol to be executed
- To interpret printed trace and graphical trace
- To execute the next transition
- To issue commands using buttons
- To send signals from the environment

## Executing the Start Transitions

In this exercise, you will execute the start transitions in the process instances of the system. First, however, we must set the amount of trace information that we want printed during execution. The command **Set-Trace** is used for setting the textual trace level.

For now, you will enter commands textually by using the text field *Command:* just below the text area; this field is called the *input line*.

1. Click in the input line to place the cursor. Enter the command **`set-trace 6`** and hit `<Return>`. The value 6 specifies that we want full information about the actions that are performed during the transitions. The entered command is moved to the text area and the simulator monitor confirms the trace setting; the text area should now show:

```
Welcome to SDL SIMULATOR. Simulating system Demongame.

Command : set-trace 6
Default trace set to 6

Command :
```

   A graphical trace (*GR trace*) is also available, which means that the execution is traced in the SDL diagrams by selecting the next symbol to be executed. GR trace is by default off when you start the simulation.

2. To enable the GR trace, enter the command **`set-gr-trace 1`**. The value 1 specifies that the next symbol to be executed will be displayed in an SDL Editor each time the monitor system is entered.

3. To have an SDL Editor window appear now, enter the command **`show-next-symbol`**. An SDL Editor window appears, showing the diagram for the process Main with its start symbol selected (this is the next symbol in turn to be executed). This SDL Editor window will be used for graphically tracing the execution of the simulator during your simulation session.

4. If needed, move and resize the Simulator UI and SDL Editor windows so that they both are completely visible and fit on the screen together. A few useful advice are:


   – Before resizing the SDL Editor, you may hide the symbol menu and the text window by using the quick buttons. The editor window will not be used for editing any diagrams, only viewing them.

   – Check and, if required, adjust the SDL Editor's option *Always new window* to off. (Use the command *Editor Options* on the *View* menu for this).

   – You may reduce the width of the Simulator UI window somewhat, which only affects the width of the text area. You may also reduce the height, but you should make sure that all buttons in the left part of the window are still visible.

   – During this simulator tutorial, you will not need to see the contents of the Organizer window. You may cover it, or minimize it by using the window system.

   To determine the transition in turn to be executed, you can look in the editor window. The start state of process Main is selected and the next state symbol is Game_Off. To execute this (empty) transition, you will use the command **Next-Transition**.

5. Execute the command Next-Transition by simply entering **`n-t`** in the input line. All commands may be abbreviated as long as the abbreviation is unique among all available commands.

   The start transition of Main is now traced in two ways:

   – In the text area, the textual trace information of the transition contains the process instance, the name of the initial state, and the current value of the simulation time. It ends with the Next-state action, giving the name of the resulting process state:

```
*** TRANSITION START
*       PId     : Main:1
*       State   : start state
*       Now     : 0.0000
*** NEXTSTATE  Game_Off
```

– In the SDL Editor, the GR trace selects the next symbol to be executed. Since this is the start state of the process Demon, that diagram is loaded into the editor.

The next transition is the start transition of Demon, which contains the setting of a timer. To execute it, you can use another feature of the command interface:

6. Place and click the pointer on the input line and press the arrow key <Up>. The command you entered previously appears (n-t). Execute it by pressing <Return>.

   – You can use the <Up> and <Down> arrow keys repeatedly on the input line to show previously entered commands. This feature is commonly known as a *command history*. You may also edit a command before it is executed, for instance by changing the value of a parameter.

   The start transition of Demon executes. Also, the SDL Editor sets the selection on the text symbol where the declaration of the timer T is found. This is a convention adopted in the SDL suite to show that the next event to take place in the system, if no signal is sent from the environment, is the expire of a timer.

   Note that the printed trace also contains the action of setting the timer T.

## Sending Signals from the Environment

To make something of interest happen in the system you have to send signals from the environment into the system. We will start by sending the signal Newgame to the Main process. For this, you can use the command **Output-Via**, which takes as parameters a signal name, the parameters of the signal (none in this case), and a channel name.

In this exercise, however, you will execute commands using buttons instead of entering them textually on the input line. The command buttons are arranged into different “modules” in the left part of the Simulator UI. You can “preview” the command that is executed by a button by se-

lecting it and move the mouse pointer away from the button before you release the mouse. The associated command is then listed in the Status Bar at the bottom of the window.

1. Locate the button module *Send Signal* and click on the *Send Via* button.

   This button executes the Output-Via command, as is shown in the text area. A dialog is opened, asking for the value of the first parameter, the signal name. The list contains all signals possible to send from the environment:

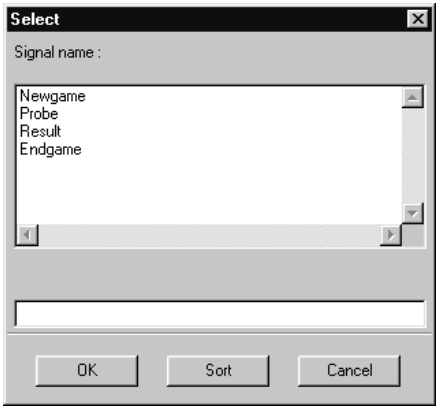


*Figure 98: Sending the signal Newgame*

2. Select the signal Newgame and click the *OK* button.
3. Another dialog is opened, asking for the channel name.

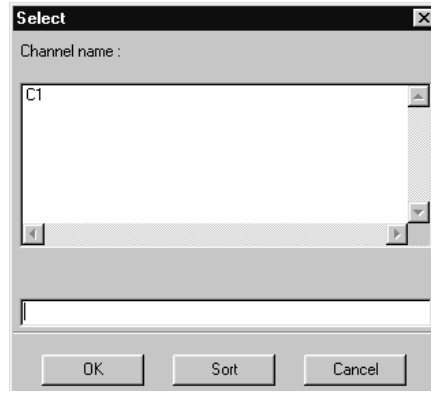


*Figure 99: Selecting the channel to send via*

4. As there is only one channel from the environment to the system, you do not have to select it explicitly. Simply click the *OK* button.

   The signal is now sent, which is confirmed in the text area. The GR trace shows that the next symbol to execute is the input of Newgame (in process Main).

5. Execute the next transition by using the button *Transition* in the module *Execute* (this executes the Next-Transition command). The printed trace information shows the actions of the executed transition up until the state Game_On. Note that the start of the transition is described by the combination of a state (Game_Off) and the input of a signal (Newgame):

```
*** TRANSITION START
*       PId    : Main:1
*       State  : Game_Off
*       Input  : Newgame
*       Sender : env:1
*       Now    : 0.0000
*    CREATE Game:1
*    ASSIGN  GameP := Game:1
*** NEXTSTATE  Game_On
```

   Since the process Game was created in the transition, the GR trace shows that the next symbol to execute is the start state of Game.

This clearly demonstrates the difference between printed trace and GR trace:

– The printed trace describes what happened in the previously executed transition, including the initial state and the reached state of the process.

– The GR trace shows what will happen next, if the system is left on its own. The start symbol of the next transition is selected, which may be in a different process diagram.

6. Execute the start transition of Game with the *Transition* button. The Game process reaches the state Losing, and the GR trace changes back to the Demon process. The SDL Editor selection shows again the text symbol with the declaration of the timer T.

7. Execute the next transition, in order to have the timer expire. This transition is a timer output, i.e., a timer that sends its signal to the process which earlier executed the Set action. A timer output is also considered to be a transition. Note that the simulation time has now been updated to 1:

```
*** TIMER signal was sent
*   Timer    : T
*   Receiver : Demon:1
*** Now      : 1.0000
```

8. Execute another transition. The start of this transition is described by the combination of the Generate state and the input of the timer T. The signal Bump is now sent to the Game process:

```
*** TRANSITION START
*      PId    : Demon:1
*      State  : Generate
*      Input  : T
*      Sender : Demon:1
*      Now    : 1.0000
*   OUTPUT of Bump to Game:1
*   SET on timer T at 2.0000
*** NEXTSTATE  Generate
```

The GR trace shows the next symbol to be the input of Bump in Game. In the process diagram, note that after the input of Bump, the Game process will be in the state Winning awaiting either the input of another Bump or the input of a Probe signal.

9. Execute the next transition to put the process Game in the state Winning. The GR trace switches back to the Demon process and indicates the next default behavior which is the expire of the timer T. However, you will instead send the signal Probe from the environment:

10. Send the Probe signal by using the *Send Via* button as before. The GR trace switches back to the Game process.

11. Execute the next transition. The Probe signal is consumed and the signal Win is output to the environment. The process returns to the state Winning and awaits a new Bump (or Probe) signal.

```
*** TRANSITION START
*       PId     : Game:1
*       State   : Winning
*       Input   : Probe
*       Sender  : env:1
*       Now     : 1.0000
*    OUTPUT of Win to env:1
*    ASSIGN  Count := 1
*** NEXTSTATE  Winning
```

We have now shown how you can use the commands Next-Transition and Output-Via to reach a certain point or state in the simulation.

# Viewing the Internal Status

In this exercise we will introduce some of the available commands for viewing the internal status of the system. With the graphical user interface, it is also possible to continuously view the internal status without having to execute commands manually.

In the previous exercise, you learned how to interpret the available traces. We will not focus on these details anymore, unless we need to point out some important aspect.

## What You Will Learn

- To restart the simulator without leaving the Simulator UI
- To use the Command and Watch windows
- To view and interpret the process ready queue
- To view the signal input port of a process
- To view variable values
- To examine process instances
- To view active timers

## Restarting the Simulator

Before continuing, you need to restart the simulation from the beginning and set the trace level:

1. Select the *Restart* command from the *File* menu.
2. A dialog informs you that the current simulation will terminate. Confirm this by clicking *OK*. The text area is cleared and the simulator is now reset.
3. Set the trace level to 6 and the GR trace level to 1, as before. The easiest way to do this is to use the up arrow key on the input line to find the previous Set-Trace 6 and Set-GR-Trace 1 commands and then hit `<Return>`.
   - The Simulator UI remembers all previous commands entered on the input line in the same session, even if you have started a new simulator.

## Viewing Process and Signal Queues

To view the internal status continuously, the *Command window* is used. This window shows, if your preferences are set up adequately, the process ready queue and a list of all processes.

This information is displayed in separate “modules” in the Command window by executing suitable monitor commands (**List-Ready-Queue** and **List-Process**). These modules are similar to the button modules in the main window.

1. Select *Command Window* from the *View* menu to open the Command window. Resize the window so that both command modules become visible (see Figure 100). Move the window so that you can still see the SDL Editor window and use the main window.

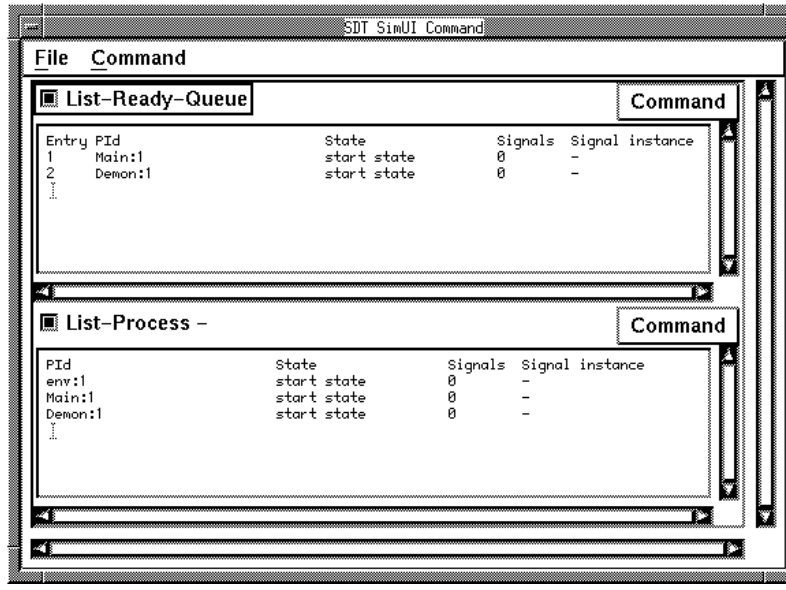


*Figure 100: The Command window*

**Note:**

Whether the Command window shows any commands or not is preference dependent. If the window does not show these command modules at start-up, you may add the commands using the *Add Command* menu choice from the *Command* menu and specify each of the commands to add. See Figure 101.

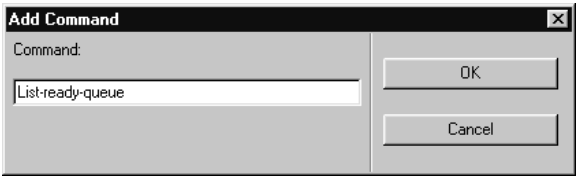


*Figure 101: Specifying a command to add*

The command modules show the following information:

– The List-Ready-Queue command displays the process ready queue, i.e., an ordered list of all processes that are ready to execute a transition. The list contains an entry number (the position in the queue), the process instance, its current state, the number of signals in its input port, and the name of the signal that will cause the next transition. At this stage, both processes (Main and Demon) are to execute their start transition, so no signals are listed.

– The List-Process command displays a list of all active processes in the system. It shows the same information as the List-Ready-Queue command above (with the exception of the additional "env" process, which represents the environment). At this stage, the two lists are identical.

2. Execute the next transition and note the changes in the Command window. The Main process is removed from the ready queue since it needs a signal input (Newgame) to execute the next transition, but this signal has not yet been sent. The process list shows the new state of Main (Game_Off).

3. Execute the next transition. The ready queue is now empty since the Demon process needs an input of the timer signal T, but this timer has not yet expired.

4. Send the signal Newgame from the environment. The Command window shows that Newgame has entered the signal input port of Main, thus adding Main to the ready queue.

5. You may also print a list of all signals in the input port of the process in turn to execute. The command for this is not available through a command button. Instead, locate the menu *Examine* in the menu bar and select the *Input Port* command. For each signal an entry number (the position in the signal queue), the signal name, and the sender of the signal is printed. The asterisk before the entry number of Newgame indicates that this signal will be consumed by the process in the next transition.

## Viewing Variables and Process Instances

Apart from the Command window, you can also continuously monitor variable values by using the *Watch window*. We will now monitor the variable GameP in process Main to see how its value changes as the process Game is started and later stopped.

1. Open the Watch window by selecting *Watch Window* from the *View* menu. If needed, move it so that you can also see the contents of the Command window.

2. In the Watch window, select *Add* from the *Watch* menu to add a variable to the list of variables to display.

3. In the dialog, you have to specify both the process (within parenthesis) and the variable name. Enter **`(Main) GameP`** and click *OK*.

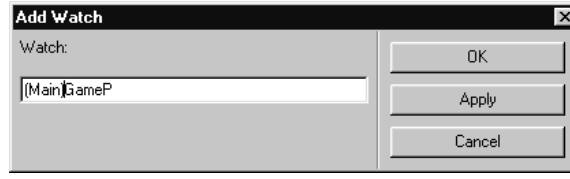


*Figure 102: Adding a variable to watch*

4. The value `null` should now be displayed in the Watch window:

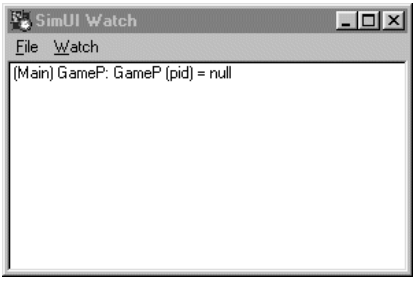


*Figure 103: Adding GameP to the Watch window*

If necessary, resize the window so that the value becomes visible.

5. Execute the next transition and check that the value of GameP in the Watch window changes to the value Game:1 as the process Game is created. Game is also added to the lists printed in the Command window.

6. You can examine the newly created Game process. The information is printed by giving the command **Examine-PId** (enter `ex-pid` on the input line) and contains the current values of Parent, Offspring, and Sender. Parent is `Main:1`, as expected.

7. Send the signal Endgame from the environment. Notice that Main is added to the ready queue, but **after** Game.

8. Execute the next transition, which is the start transition of Game.

9. You can now examine the Main process, since it is the next to execute. Give the command Examine-PId again and compare the values printed with those from the Game process.

10. Execute the two next transitions to stop the Game process. Notice that the value of GameP is reset to null and that Game no more is listed in the Command window.

## Other Viewing Options

There are a number of other viewing commands available in the *Examine* menu. You can list the active timers in the system, check the parameters of signal and timer instances, etc. We will conclude this exercise by showing that the system is not idle, even though the ready queue is now empty.

1. Check that the timer T is still active by choosing *Timer List* in the *Examine* menu. The timer's name, corresponding process instance, and expiration time is printed.

2. Execute the next transition. Try to examine the timer instance by giving the command **Examine-Timer-Instance** (enter `ex-tim-ins` on the input line). You are informed that the timer queue is empty, i.e. the timer T is no longer active.

# Dynamic Errors

## What You Will Learn

- To recognize and interpret a dynamic error
- To find the SDL symbol last executed
- To continue an interrupted transition

## Finding a Dynamic Error

In this exercise, a dynamic error in the Demongame system will be detected. The error is found by simply executing the first four transitions of the system:

1. Select the *Restart* command from the *File* menu.

2. Set the trace level to 6.

3. We will not use graphical trace in this exercise. So, exit the currently open SDL Editor from its *File* menu.

4. As you will not need the Command and Watch windows, close them by selecting *Close All* from the *View* menu.

5. Execute the four next transitions until a warning message is printed in the text area:

```
********************* WARNING ********************
Warning in SDL Output of signal Bump
Signal sent to NULL, signal discarded
Sender: Demon:1
TRANSITION
  Process       : Demon:1
  State         : Generate
  Input         : T
  Symbol        :
#SDTREF(SDL,c:\Telelogic\SDL_TTCN_Suite4.5\work\...
TRACE BACK
  Process       : Demon
  Block         : DemonBlock
  System        : Demongame
**************************************************
```

The message indicates that there was no receiver for the Bump signal sent from the Demon process. This is quite true, as no process instance of type Game has been created. The definition of the Demon game is thus not correct, as it is actually requires that the user always has a game running, when Bump signals are sent. A better (and correct) solution would be to direct the Bump signals from Demon to Main, which then retransmits the signal to the instance of the Game process, if it exists.

6. When no GR trace is in effect, you can still see where the error occurred. Choose *Prev Symbol* in the *Show* menu. This opens an SDL Editor and selects the last symbol that was executed. In this case, the output of Bump in the Demon process.

   After a dynamic error has occurred it is, of course, possible to continue the simulation, both to execute more transitions and to examine the status of the system. Note that the execution was stopped directly after the symbol in which the dynamic error occurred, i.e. the transition was interrupted.

7. To execute the interrupted transition to its end, issue a Next-Transition command as usual. In the printed trace you can see that no signal was sent in the erroneous output statement.

# Using Different Trace Values

The amount of trace information printed during transitions is set by the command Set-Trace. So far, you have used this command to set the trace value to 6. The higher the trace value you set, the more information is printed.

You can also define trace values for different parts of the system. In this way, blocks, process types, process instances, etc. can have different trace values. If a process does not have a trace value defined, the value for the enclosing block is used. If the block does not have a defined value, the value for the next enclosing structure is used, etc. The system always have a trace value defined, which initially is 4.

In this exercise, you will use these facilities to run the demon game and only print trace information for transitions executed by the processes Main and Game. The process Demon will not be traced. This is accomplished by setting the trace value for the system to 0 and the value for the block GameBlock to 6.

The GR trace value will be set to 1 throughout this exercise.

## What You Will Learn

- To set trace values for different parts of a system
- To list the current trace values
- To execute the next SDL symbol only
- To execute a sequence of transitions until trace is printed

## Setting Trace Values

The command Set-Trace actually takes two parameters, the name of a unit and a trace value, and assigns the trace value to the unit. To easily specify the unit, you will now execute Set-Trace by using a menu, instead of entering it on the input line.

1. First, restart the simulator. If needed, resize and move the SDL Editor window that is open.

2. Locate the *Trace* menu and select the *Text Level : Set* entry. In the first dialog, select the unit *System Demongame* and click OK. In the second dialog, select the trace value 0. Note that all possible trace values (0-6) have a short explanation.

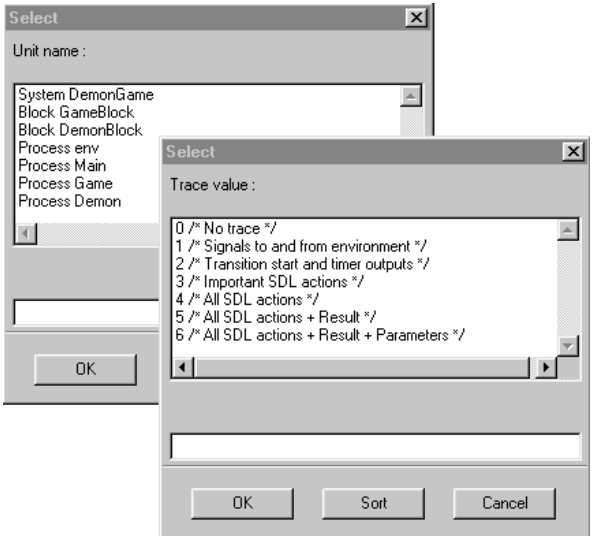


*Figure 104: Setting the trace 0 for System DemonGame*

3. In a similar way, set the trace value for the block GameBlock to 6.
4. The menu choice *SDL Level : Set* can be used to set the GR trace value in the same way. Use it to set the trace value for the system DemonGame to 1.
5. Check that you have set the correct trace values by using the menu choices *Text Level : Show* and then *SDL Level : Show*. The following information should be printed:

```
Default    4 = All SDL actions
System  Demongame  : 0 = No trace
Block   GameBlock  : 6 = All SDL actions + Result +
Parameters

Default    0 = GR trace off
System  Demongame 1 = Show next symbol when entering
monitor
```

## Executing Symbol by Symbol

To clearly see that the Demon process is not traced in the text area, we will follow the execution in smaller steps than complete transitions. The smallest execution step possible is one SDL symbol at a time. The command **Step-Symbol** is used for this.

1. First, send the Newgame signal from the environment.
2. Execute the start symbol of Main by clicking the button *Symbol* (in the *Execute* module). Note that the printed trace does not include information about the next state (Game_Off) since that symbol has not yet been executed (it is selected to be executed next):

```
*** TRANSITION START
*       PId     : Main:1
*       State   : start state
*       Now     : 0.0000
```

3. Execute the next symbol with the *Symbol* button. Now, the printed trace gives information about the Game_Off state being reached:

```
*** NEXTSTATE  Game_Off
```

4. The execution now continues in the Demon process. Execute the three symbols in the start transition of Demon. Note that no trace is printed in the text area, since Demon is not part of the block GameBlock.
5. Continue executing the symbols in the Main and Game processes until the Demon process is entered again (you will need to press the *Symbol* button a number of times; watch the SDL Editor window for monitoring the execution). Note that trace is printed for each symbol.
6. When the Demon process is entered, you can continue to execute symbol by symbol, or you may execute the complete transition by using the *Transition* button as usual. No trace is printed.
7. Stop executing when you are back in the Game process.

## Hiding Uninteresting Transitions

If you would continue to execute transition by transition at this point, trace would only be printed while executing the Game process. But, you would still have to manually execute the “silent” transitions in the Demon process. To avoid this, you can use another command, **Next-Visible-Transition**. This command executes a sequence of transitions; it stops after it has reached a process with a trace value greater than 0, i.e., when the first “visible” transition is executed. In this way, transitions by uninteresting parts of the system are hidden.

1. Execute the command by choosing *Until Trace* in the *Execute* menu. The execution does not stop until the Game process is entered again and the state Winning (or Losing) is reached. Trace is then printed for the last executed transition.

2. Repeat the command a number of times. The printed trace shows that you are now switching between the states Losing and Winning in the Game process. The execution in the Demon process is hidden in the printed trace.

   You should note, however, that the GR trace only shows the Demon process. Remember that the GR trace selects the next symbol to be executed, which is always in the Demon process when the Game process has reached the state Winning or Losing. If you want to check where in the Game process you are, do as follows:

3. Choose *Prev Symbol* in the *Show* menu to select the last executed symbol. This should be a state symbol, Winning or Losing, in the Game process.

# Looking at the External Behavior

## What You Will Learn

- To use the signal log facility
- To add command buttons to the Simulator UI
- To execute transitions up to a certain time value

## Setting Trace and Signal Logging

During this exercise, you will look at the external behavior of the system, which is the same as actually playing the Demon game. To achieve this, we will set the system trace to 1. This means that you will see only signals sent to the environment and none of the actions performed during transitions. In order to log the external behavior on a file, you will also use the *signal log* facility.

1. As usual, restart the simulator.

2. Set the trace value for the system to 1.

3. To log the signals sent to and from the environment, enter the command **`signal-log`** in the input line and hit `<Return>`. (This command has no associated button or menu choice.)

   The **Signal-Log** command takes two parameters, which are now asked for in dialogs. The first parameter is a unit name. All signals sent to, from or through the specified unit will be logged to file.

4. Instead of selecting one of the units in the list, enter the unit name **`env`** in the dialog's text field and click *OK*. This is the way to specify the environment of the system.

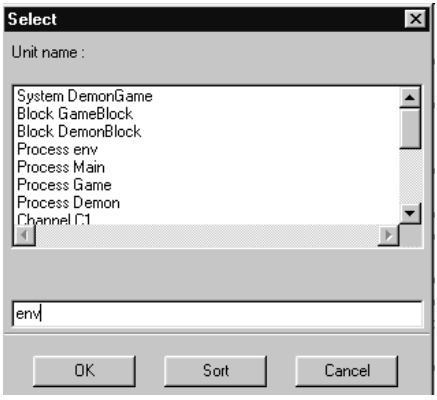


*Figure 105: Specifying the environment*

The second parameter is a the name of a file name to which information about the signals will be written. A file selection dialog is opened.

5. In the *File* field, enter the file name `signal.log` and click *OK*.

## Adding Buttons for Common Commands

When you are playing the Demon game, you are sending signals to the system from the environment. You will start by sending the signal Newgame. Since this is an action often performed in the simulation of this system, we will first define a new button that executes the proper command. In this way, you only need to click the button to send the signal.

1. In the *Send Signal* module, select *Add* from the *Group* menu to the far right:

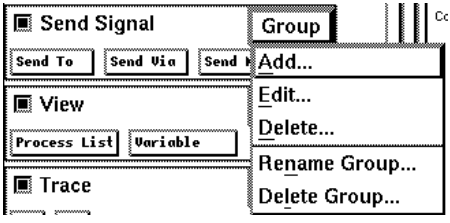


*Figure 106: Adding a new button to a module*

2. In the dialog, enter **`Newgame`** as the button label, but **do not** hit `<Return>`. Enter **`output-via newgame -`** as the command definition.

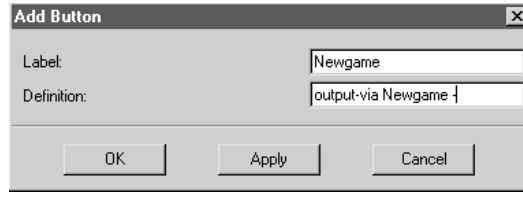


*Figure 107: Adding a button*

   – The '-' as the last parameter to Output-Via indicates the default value, in this case for the channel parameter. Specifying '-' as a parameter is the same as just clicking *OK* in the corresponding parameter dialog.

3. Click *Apply*. The new button appears in the module, and the dialog is ready for another button definition.

4. Since sending the Probe signal also is a common action, add a button *Probe* in the same way as above.

5. If you wish, add buttons for the signals Result and Endgame in the same way. Finally, close the dialog with the *OK* button.
   (If the *Error message "Button label must not be empty"* occur, just ignore the message.)

## Playing the Game

You are now ready to start playing the game. You will use the new buttons to send signals to the game, and the command **Proceed-Until** to execute transitions up to the next point in time when you want to send a signal.

1. Send the signal Newgame with a click on the *Newgame* button.

2. Execute transitions until the time is 5.5 by selecting *Until Time* (in the *Execute* menu). Enter the value **`5.5`** in the dialog. This executes the command `Proceed-Until 5.5`. This will execute all transitions up to the point in time when the simulation time first becomes equal to the specified time value.

3. Send a Probe signal.

4. Execute transitions until the time is 10.3. Note the output of the signal Win or Lose to the environment.

5. Send a Probe signal again. Then, send another Probe signal. The two signals will enter the input port of the Game process. Check this by selecting *Input Port* in the *Examine* menu:

```
Input port of Game:1
Entry   Signal name          Sender
*1      Probe                env:1
 2      Probe                env:1
```

6. Send the signal Result. Use the button if you have defined one; otherwise, use the *Send Via* button or enter the command on the input line.

7. Execute transitions until the time is 13.5. Note the output of the signals Lose or Win (one for each Probe) and Score to the environment.

## Examining the Signal Log File

1. Exit the simulation by choosing *Stop Sim* in the *Execute* menu. This is needed to finish the signal logging. The Simulator UI itself is not closed by this command.

2. Examine the file `signal.log` from outside the simulator. The file contains a specification of all signals sent to and from the environment. It should look like this:

```
Signal log for system Demongame with unit Process
env on file ...
0.0000 Newgame from env:1 to Main:1
5.5000 Probe from env:1 to Game:1
5.5000 Win from Game:1 to env:1
10.3000 Probe from env:1 to Game:1
10.3000 Probe from env:1 to Game:1
10.3000 Result from env:1 to Game:1
10.3000 Lose from Game:1 to env:1
10.3000 Lose from Game:1 to env:1
10.3000 Score from Game:1 to env:1
Parameter(s) : -1
```

# Using Breakpoints

The facility of a simulator demonstrated in this exercise is the *breakpoint*. A breakpoint can be used to stop the execution and activate the monitor system at a certain point of interest. There are four kinds of breakpoints; symbol, transition, output and variable. The first two kinds will be explained in the following.

## What You Will Learn

- To set a breakpoint on an SDL symbol
- To set a breakpoint on a transition
- To list all defined breakpoints
- To graphically trace each executed SDL symbol
- To get a textual SDT reference to an SDL symbol
- To continuously execute the system

## Setting Up the System

To see where a breakpoint is reached, you will start the execution of the system with the Go command. This command continuously executes transitions until an error occurs, a breakpoint is reached, or the system is completely idle. You should first set up the system in a way suitable for continuous execution:

1. Restart the simulator.
2. Set the trace value for the system to 0 to avoid trace information being printed during execution (you may use the *Text level : Set* entry from the *Trace* menu to do this) .
3. Set the GR trace value for the system to 2. Each SDL symbol will then be selected in the SDL Editor as it is executed, allowing you to follow the execution even though no trace is printed.

## Setting a Symbol Breakpoint

A *symbol breakpoint* is set at a specific SDL symbol in the process diagrams. Symbol breakpoints are checked **before** symbols are executed, i.e. the symbol is not executed when the breakpoint is reached. We will now show how to set a breakpoint on the first task symbol in the Game process, i.e. the initializing of the variable Count to 0.

1. First, send the signal Newgame (using the new button). This is very important, as otherwise the Game process will not be created and the breakpoint will never be reached!

2. In the SDL Editor, bring up the Game process diagram from the *Diagrams* menu.

   – If, for some reason, this diagram is not included in the menu, you have to open it explicitly. Either click the *Open* quick button and select the file `Game.spr`, or double-click the Game diagram icon in the Organizer window.

3. In the Simulator UI, choose *Connect sdle* in the *Breakpoint* menu (you may have to resize the window to see the menu). This establishes a connection between the Simulator and the SDL Editor. As a consequence, a new menu *Breakpoints* appears in the SDL Editor's menu bar (you may have to resize the window to see the menu).

4. Go back to the SDL diagram and select the task symbol "Count:=0". Then, select the **second** command *Set Breakpoint* from the new *Breakpoints* menu in the editor (the one without trailing dots). The symbol breakpoint is now defined.

   A red "stop" sign is added to the task symbol to indicate the breakpoint. Back in the Simulator UI, the definition of the symbol breakpoint is printed.

5. Start executing the system by pressing the *Go* button in the *Execute* module. Note how the SDL symbols are selected in rapid succession as they are executed. Finally, the breakpoint is reached and the execution stops. The symbol where the breakpoint was set is next to be executed.

## Setting a Transition Breakpoint

A *transition breakpoint* is set at a specific transition in the system. Transition breakpoints are checked **before** transitions are executed, i.e. the transition is not executed when the breakpoint is reached. We will set a breakpoint in the Demon process, when it is in the state Generate and receives the timer T.

1. To define the breakpoint, choose *Transition* in the *Breakpoint* menu of the Simulator UI. This command takes a number of parameters. In the dialogs that appear:
2. Select the Demon process and click *OK*:

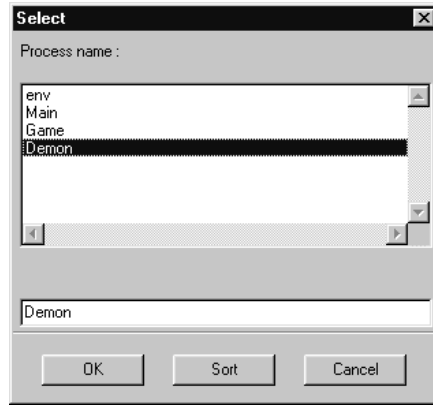


*Figure 108: Specifying the process*

3. Leave the instance number empty and click *OK*:

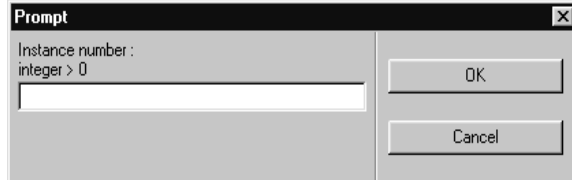


*Figure 109: Leave the instance number empty*

4. Do not specify a service name; simply click *OK*:

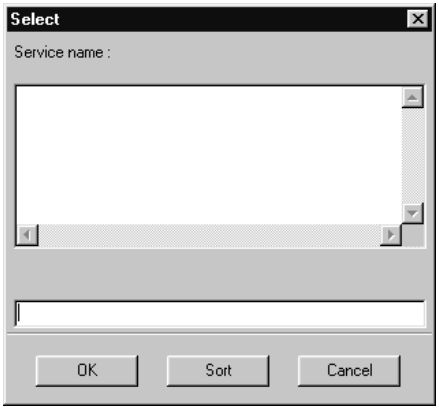


*Figure 110: Do not specify a service name*

5. Select the Generate state and click *OK*:

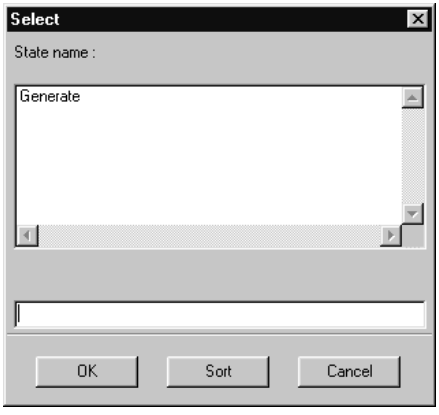


*Figure 111: Specifying the state Generate*

6. Select the timer T and click *OK*:

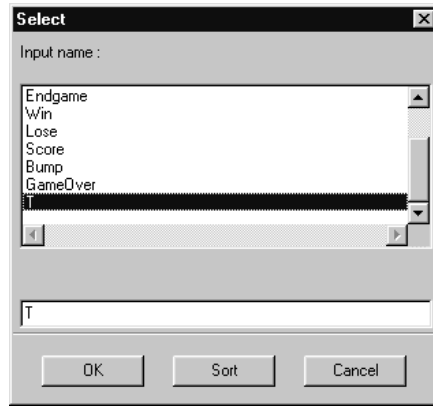


*Figure 112: Specifying the timer T*

7. Simply click *OK* in the remaining dialogs:

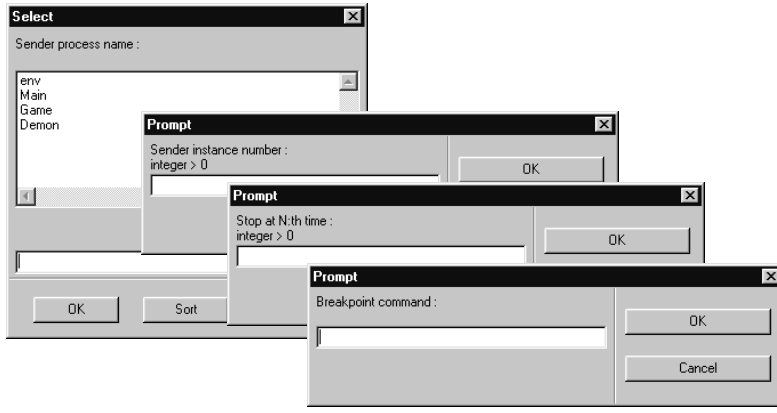


*Figure 113: Leaving the remaining dialogs empty*

To omit selecting a parameter value is interpreted so that any value, name or number, will match this parameter. In this case, any instance of Demon and any sender will match the breakpoint.

8. To see how the new breakpoint was defined, list all breakpoints with the *Show* entry in the *Breakpoint* menu:

```
1
Process name     : Demon
Instance         : any
Service name     : any
State            : Generate
Input            : T
Sender name      : any
Sender instance  : any
Stop each time
```

9. Resume execution of transitions by clicking the *Go* button. When the breakpoint is reached, you can see that the current state of the system matches the breakpoint definition:

```
Breakpoint matched by transition
PId    : Demon:1
State  : Generate
Input  : T
Sender : Demon:1
Now    : 1.0000
```

# Changing the System

There are a number of commands in the simulator monitor that change the behavior of the system. These commands should be used with care, since it is no longer the original system that is simulated after such a command has been issued. These commands are still useful, especially in debugging situations, for making minor changes so that it is possible to continue the simulation after an error has been detected. They can also be used to force the system into certain situations, that otherwise would require a large number of transitions to be attained.

## What You Will Learn

- To change and add commands in the Command window
- To create a process manually
- To change the process scope
- To change the value of a process variable
- To change the state of a process
- To set and reset a timer manually

## Some Preparations

In the following two exercises, you will be changing processes and timers. Before continuing, you will set up the simulation and the Simulator UI in a suitable way.

1. Restart the simulator.
2. Open the Command window through the *View* menu. You will now change the commands executed in the Command window. The List-Process command is to be replaced by **Examine-PId**, and a new command, **List-Timer**, will be added.
3. In the List-Process command module, select *Edit* from the *Command* menu to the far right. (This menu works in the same way as the *Group* menu in the button modules.)
4. In the dialog, change the command to `examine-pid` and click *OK*.
5. Go to the Command window's menu bar and select *Add Command* from the *Command* menu. In the dialog, enter the command `list-timer` and click *OK*.
6. A new command module is added to the window. Resize the window so that all three modules are visible.
   – You may also reduce the number of text lines displayed in a module by selecting *Size* from the module's *Command* menu. In a dialog, you can set the number of lines with a slider.

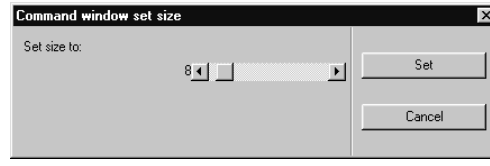


*Figure 114: Adjusting the number of lines*

7. Set the system trace value to 6 to get full trace.
8. Execute the two first transitions so the processes Main and Demon are started.
9. If the GR trace in the SDL Editor window makes it difficult to see the output in the Command window, set the system GR trace to 0.

## Creating a Process

In this exercise, we will put the system in the state it would be in after the reception of a Newgame signal. This will be accomplished without actually sending the signal. Instead, we will manually create an instance of process type Game, using the Create command.

1. Create the Game process by selecting the *Create Process* entry (in the *Change* menu). Select the process Game, and click *OK*.
2. In the next dialog, select the parent process Main and click *OK*. This sets up the Parent-Offspring link between the process instances.

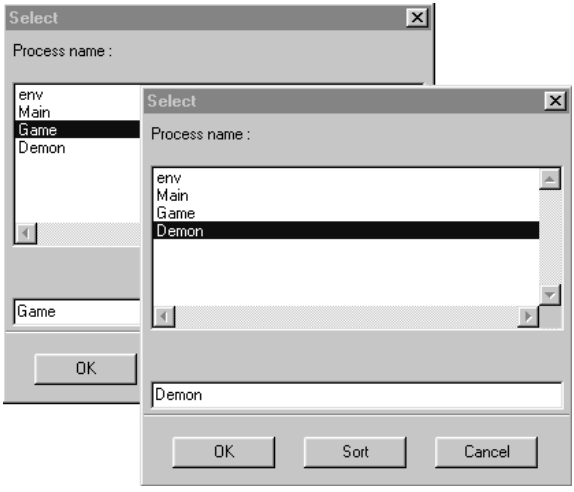


*Figure 115: Creating the process Game from process Main*

To complete the actions taken when a Newgame signal is received, you must also set the GameP variable in Main, and put Main in the state Game_On. This is done with the commands **Assign-Value** and **Nextstate**. However, these commands operate on the process next to execute, which at this stage is Game, as can be seen in the ready queue printed in the Command window.

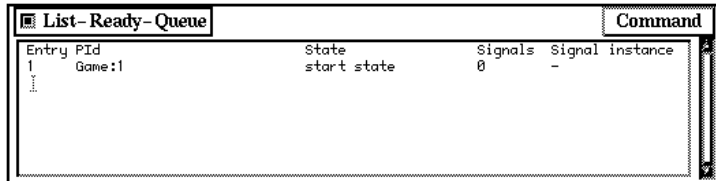


*Figure 116: The Ready Queue*

*Next process to execute is Game.*

Therefore, you first have to change the current process, also known as the *process scope*.

3. Change the scope by selecting *Set Scope* in the *Examine* menu. Select the process Main in the dialog and click *OK*.

   The Examine-PId command in the Command window shows that Main is the current process. The variable GameP must be set to the value of Offspring in Main. In the Command window, check that this value is `Game:1`.

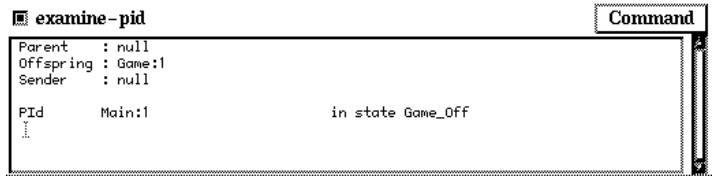


*Figure 117: Process Main is the current process*

*The Offspring of the current process is Game:1.*

4. Assign the GameP variable by selecting *Variable* in the *Change* menu. In the dialogs that follow, select the variable GameP, and enter the PId value **`Game:1`**.

5. To put the Main process in the correct state, use the *State* entry in the *Change* menu. Select the state Game_On and click *OK*.

   The system is now in exactly the state it would be in after the reception of a Newgame signal. Even though you have changed the process scope, the next transition to be executed is still the start transition of Game. (You can check this by viewing the process ready queue. See Figure 116 on page 171.)

6. Execute the next transition and check that the Game process is started.

## Changing the State of Timers

In this exercise, we will execute Set and Reset actions on timers directly in the monitor system. At this stage, the timer T is active, as it has been set by the Demon process. You can check this by looking at the List-Timer module in the Command window.

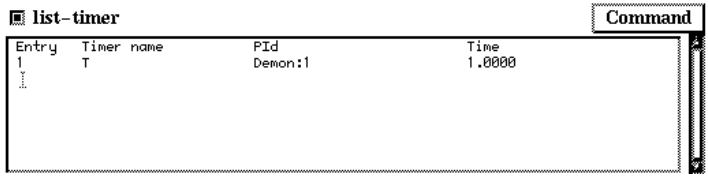


*Figure 118: The timer T is active*

1. Reset the timer by choosing *Reset Timer* in the *Change* menu. Select the timer T and click *OK*.

2. Try to execute the next transition and note the message printed:

   ```
   No process instance scheduled for a transition
   ```

   The system is now completely idle, i.e., there are no transitions in the system that can be executed. The Command window shows that both the ready queue and the timer queue is empty. To restart the system you must perform a set operation on timer T in process Demon.

3. Choose *Set Timer* in the Change menu, select the timer T and enter a time value of `10`.

4. Execute the next transition and check that the timer was set at time 10 (look at the trace in the main window).

   ```
   *** TIMER signal was sent
   *   Timer    : T
   *   Receiver : Demon:1
   *** Now      : 10.0000
   ```

# Generating Message Sequence Charts

In this exercise, we will demonstrate the power of Message Sequence Charts as a method of illustrating, in a graphical way, the dynamic behavior of the system. This can easily be done when simulating the system by using *MSC trace*. MSC trace transforms some of the SDL events that take place into MSC events; typically sending of signals and dynamic creation of processes. The trace can then be graphically logged in an MSC Editor during the execution.

Earlier in this tutorial, you drew a Message Sequence Chart which illustrated a simple sequence of messages. We will now run the simulator and generate the MSC trace of the events which actually take place.

## What You Will Learn

- To start and stop logging of MSC events
- To trace back from an MSC to an SDL diagram

## Initializing the MSC Trace

1. Restart the simulation.
2. Make sure the GR trace is disabled to avoid having the SDL Editor window being updated and raised (select the *SDL Level : Show* entry in the *Trace* menu and verify that system GR trace is 0).

   By default, the MSC trace is enabled for the entire system. You must, however, explicitly start the interactive logging of MSC events:

3. Choose *MSC Trace : Start* in the *Trace* menu. A dialog is issued, where you are prompted to specify the amount of symbols to include in the MSC trace. You will include states in the MSC trace, so select 1 and click *OK*.

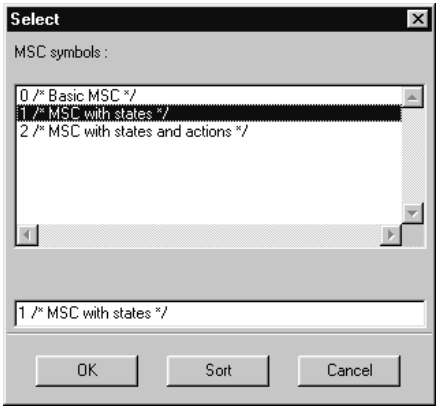


*Figure 119: Specifying to include states in the MSC trace*

4. An MSC Editor is opened, displaying an MSC diagram named "SimulatorTrace." If needed, move and resize the window to make it fit on the screen together with the Simulator UI.


   – Before resizing the MSC Editor, you may hide the symbol menu and the text window by using the quick buttons. You should not edit the generated MSC, so there is no need for these windows.

   Initially, the system has three active instances, the processes `Main_1_1` and `Demon_1_2`, as well as `env_0` which has been introduced in order to represent the environment to the system:

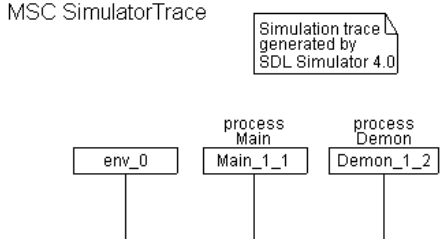


*Figure 120: The initial appearance of the MSC*

## Tracing the Execution in the MSC

1. Send the signal Newgame from the environment. This is now displayed in the MSC Editor as a message sent from the instance `env_0`.

   At this stage, the message is connected to the instance Main_1_1, since it has not yet been consumed. Instead, the message is temporarily drawn as a “lost” message, indicated by the filled circle. You may also see the text “Main_1_1” associated with the circle, indicating the intended receiver of the message.

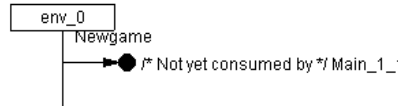


*Figure 121: The sent Newgame signal*

2. Execute the next transition. A condition symbol with the text Game_Off appears on the instance axis for the process Main. This symbol shows that the process has started executing and has reached the corresponding SDL state, Game_Off.


*Figure 122: The Condition symbol*

3. Execute the next transition. The timer T is set in the Demon process. The vertical coordinate is incremented downwards in the MSC, enhancing the impression of an absolute order of events. Also, a condition symbol with the text Generate is drawn on the instance axis.

4. Execute the next symbol only (use the *Symbol* button). The Main process consumes the Newgame message; the filled circle disappears. Note that the start point and the end point of the message have different vertical positions, since the timer T was set after the message was sent.

5. Execute the next symbol. An instance of the Game process is created, thus adding a new instance head and instance axis. The MSC should now look like this:

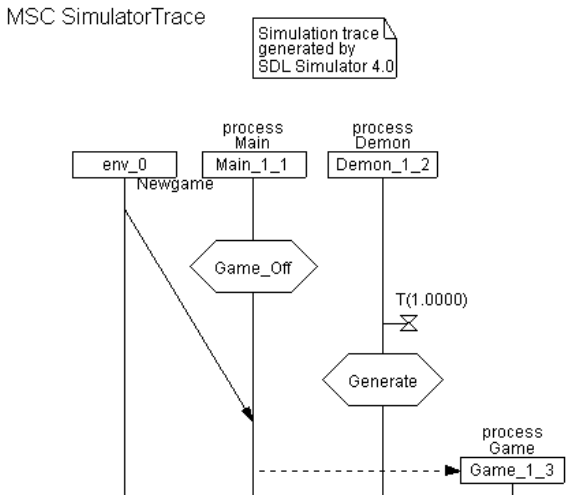


*Figure 123: The MSC after creating the Game process*

6. Execute the next transition by clicking the *Transition* button twice. The second transition causes the Game process to enter the state Losing.

7. Execute the next three transitions. The timer signal T is consumed and the signal Bump is sent and consumed. The Game process is now in the state Winning. Note how the signal interchange is shown in the MSC.

8. Next we illustrate a message which is consumed immediately. Send the signal Probe from the environment and execute the next transition. First, the message Probe is displayed (marked with filled circle), then it is redrawn, keeping its horizontal alignment.

9. The system responds with the signal Win.

10. Send the signal Result and execute the next transition. In the MSC, you can see that the message Score has the parameter 1.

11. End the game by sending the Endgame signal and execute the next two transitions. The Game process is stopped.

The MSC should now look like in the figures below. (You may note a dotted horizontal line in the MSC diagram on screen. This indicates where a page break will occur if you would print out the diagram.)

Compare this diagram with the one in Figure 90 on page 119. You will notice differences between the hand-drawn and the generated diagram. These discrepancies are quite natural, since it is impossible to predict the dynamic behavior of a system just by looking at the SDL diagrams.

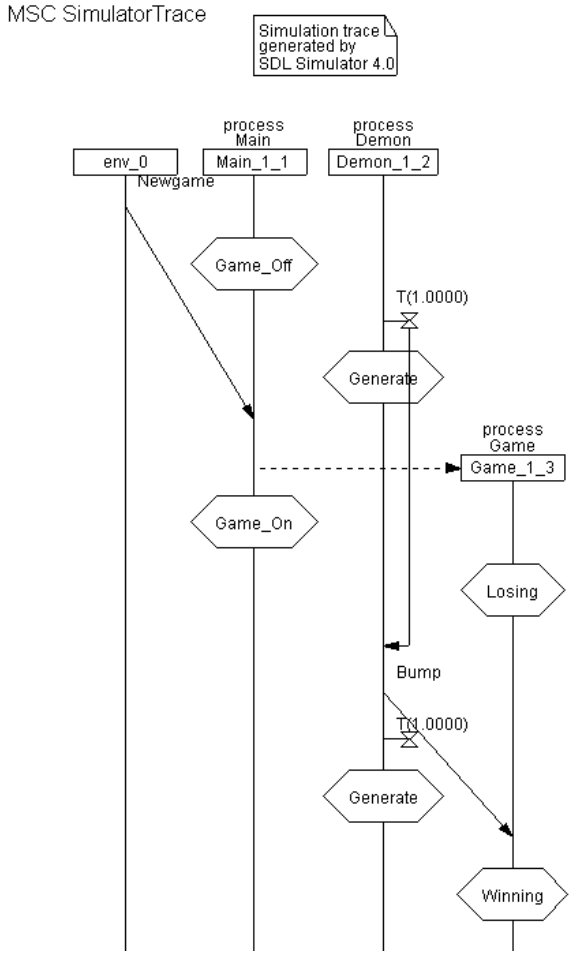


*Figure 124: The finished MSC 1(2)*

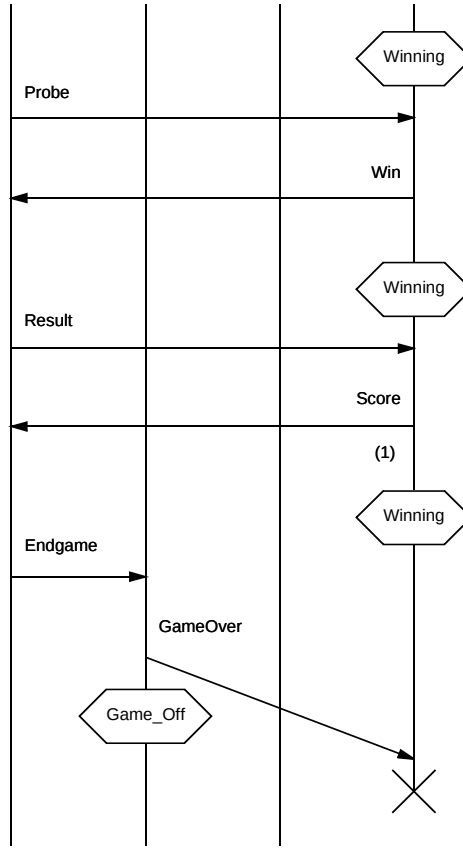


*Figure 125: The finished MSC 2(2)*

## Trace-Back to SDL Diagrams

From the generated MSC diagram, you may obtain a trace back to the SDL source diagrams.

1. From the MSC Editor's *Window* menu, select *Info Window*. A window is opened, containing information about the graphical object which is currently selected. (The amount and type of information depends on what sort of object you have selected.)


*Figure 126: The Info window*

*The window shows information related to the message Bump.*

2. Select a few different objects in the MSC Editor and note how the information in the Info window changes.

3. Select the message Bump and click on the button *Show SDL Symbol* in the Info window. An SDL Editor is opened, in which the symbol corresponding to the actions of sending or consuming the SDL signal Bump is selected:

   – If you have selected the message Bump by clicking on a point which is closer to the start point of the message than to the end point, the corresponding SDL **output** symbol will be selected.

   – Otherwise, the corresponding SDL **input** symbol will be selected.

## Ending the MSC Trace

1. Stop the logging of MSC events by selecting the *MSC Trace : Stop* entry in the *Trace* menu.

2. In the MSC Editor, save the generated MSC diagram under the file name `SimulatorTrace.msc`.

3. *Exit* the MSC Editor.

# The Coverage Viewer

In this final exercise of the SDL Simulator, you will learn to use the Coverage Viewer. The Coverage Viewer is a graphical tool that shows how much of a system has been covered during a simulation in the terms of executed transitions or symbols. By checking the system coverage, you can for instance see what parts of the system that have not been executed in the simulation so far.

## What You Will Learn

- To create a coverage file
- To start the Coverage Viewer
- To interpret and change the coverage tree information
- To open a more detailed coverage chart
- To exit the Simulator UI

## Starting the Coverage Viewer

1. Restart the simulator.
2. Send the signal Newgame. Execute seven (7) transitions until the printed trace shows that the Game process is in the state Winning.
3. Send the signal Probe and execute the next transition.

   Let us see how much of the system we have executed so far. By simply starting the Coverage Viewer from the Simulator UI, the current coverage information is displayed.
4. Select *Coverage* from the *Show* menu. The main window of the Coverage Viewer is opened.

## Using the Coverage Viewer

1. If a *symbol coverage tree* is displayed, switch to a transition coverage tree by clicking on the *Tree Mode* quick button.

2. To see all of the *transition coverage tree*, click on the *All Nodes* quick button.

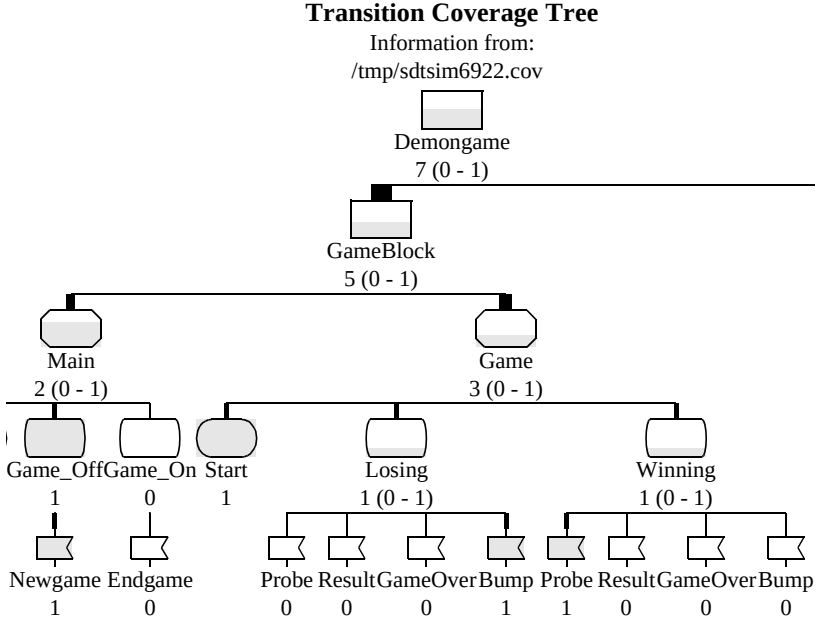


*Figure 127: A transition coverage tree*

*Only a part of the tree is depicted.*

– Since the coverage tree is quite large, you may maximize the window (by using the window manager) while working with the Coverage Viewer. When returning to the Simulator UI, you can simply restore the size of the window to its original size.

– Alternatively, you may also zoom out the window contents with a number of clicks on the quick button.

The coverage tree shows the diagram structure of the Demongame system. Beneath each process diagram (Main, Game, Demon), you see all possible transitions in that process, defined by a state and the possible signal inputs from that state. The start transitions are represented by an SDL start symbol.

The number below each symbol in the tree is the number of times the symbol has been executed so far. In addition, each symbol is filled with a gray pattern to indicate to what extent it has been executed. Parts of the system that never have been executed have a zero

value and “empty” symbols. Parts that have been completely traversed by the execution so far have a non-zero value and completely filled symbols.

From the displayed tree, you can find out the following information:

– In the states Losing and Winning, one out of four transitions have been executed. Thus, these state symbols are filled to 1/4.

– In the Main process, two out of three transitions have been executed. Thus, the process symbol is filled to 2/3.

– In the Demon process and the block DemonBlock, all transitions have been executed at least once. Thus, these symbols are completely filled.

– In the system as a whole, a little more than half of all possible transitions have been executed.

3. To only see those transitions that never have been executed, click on the *Least* quick button. You can now see which signals must be sent in which states to execute the rest of the system

   – The *Least* quick button actually shows those symbols that have been executed the **least** number of times. The symbols that are dashed are present only to make the structure complete.

4. In the same way, to only see those transitions that have been executed at least once, click on the *Most* quick button. You can now see which signals have been sent so far in the system.

   – The *Most* quick button actually shows those symbols that have been executed the **most** number of times. In this case, no transition has been executed more than once.

5. To see the whole tree again, click on the *All Nodes* quick button. If you want to see a transition in the SDL Editor, just double-click on one of the signal input symbols or start state symbols. Try this!

The Coverage Viewer can also show *a symbol coverage tree*, i.e. how many times each SDL symbol in the process diagrams have been executed:

6. Switch to a symbol coverage tree by clicking on the *Tree Mode* quick button. Beneath each process diagram, you will now see a small icon for each SDL symbol. To see which SDL symbol an icon represents, double-click the small icon.
7. Switch back to a transition coverage tree and go back to the Simulator UI.

## Augmenting the Coverage

1. Execute three more transitions to put the Game process in the state Losing again.
2. Send the signal Result. Execute four more transitions to return to state Winning.
3. Send the signal Endgame. Execute two more transitions to stop the Game process.
4. Check the current system coverage in the Coverage Viewer by selecting *Coverage* from the *Show* menu.
5. Change to a transition coverage tree and show *All Nodes*. As you can see, the Main process has now been completely executed. The Losing and Winning states are also more filled.
6. To see what transitions have still not been executed, you can click the *Least* quick button. If, however, you click the *Most* quick button you will only see the most executed transition (and other symbols), i.e. the input of timer T, not all transitions that have been executed.
7. To see more of the tree, select *Increase Tree* from the *Tree* menu. The input of Bump should now be added. Select the command again and all remaining executed transitions should be added.

## Looking at Coverage Details

1. Select the system diagram in the Coverage Viewer. From the *Tools* menu, select *Show Details*, or click the quick button for this. The *Coverage Details* window is opened.

   The displayed coverage chart shows how many transitions that have been executed a certain number of times. The chart should contain four bars:

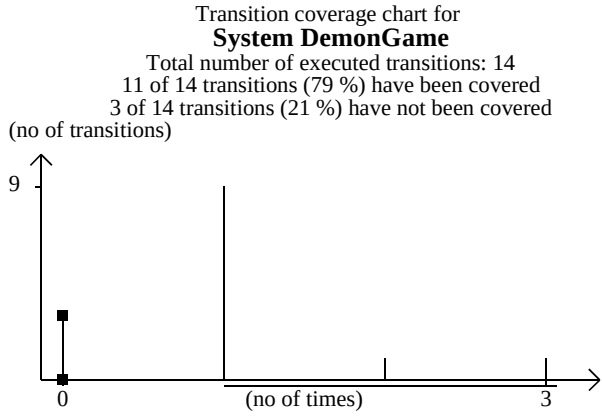


*Figure 128: The Coverage Details window*

2. Select the bars one at a time and look at the text in the Status Bar at the bottom of the window. You will now see how many transitions that have been executed 0, 1, 2 and 3 times. To see which transitions that have not been executed at all, do as follows:

3. Select the "zero bar" furthest to the left. Click the *Show Editor* quick button. The SDL Editor is opened, showing the three transitions in the Game process that remain to be executed.

   – If the transitions can be found in more than one diagram, a confirmation dialog is issued between each diagram that is opened. Simply click *OK* to see the transitions in the next diagram.

4. Select another symbol in the coverage tree in the main window. The Coverage Details window is now updated to show the coverage chart for that symbol.

5. “Play around” in the Coverage Viewer as much as you like. You should note, however, that the Demongame system is a bit too simple to give full justice to the power of the Coverage Viewer.

### Exiting the Simulator UI

You will now close down the Simulator UI.

1. First, exit the Coverage Viewer from the *File* menu.
2. Then, exit the Simulator UI from the *File* menu. You will be asked whether to save the changes to the sets of variables in the Watch window, commands in the Command window, and buttons in the button area. If you choose to save them (with the suggested file names), they will become the default the next time you start the Simulator UI from the same directory.

## So Far...

You have now learned how to “animate” an SDL system by generating, executing and tracing a simulator.

If your configuration includes the Validator tool, we suggest that you proceed with the exercises on the Validator. These exercises start in *chapter 5, Tutorial: The SDL Validator*.

In all cases, the example you have been practising on, the system DemonGame, is rather simple. To deepen your knowledge of the SDL suite, you may practise on a number of exercises that illustrate the advantages of SDL-92 when adopting an object-oriented design methodology. These exercises are described in chapter 6, *Tutorial: Applying SDL-92 to the DemonGame*.