Chapter

5

Tutorial: The SDL Validator

The SDL Validator is the tool that you use for validating the behavior of your SDL systems, using state space exploration techniques. In this chapter, 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* as well as *chapter 4, Tutorial: The SDL Simulator*.

In order to learn how to use the Validator, 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 validation functionality in the SDL suite. With validation we mean exploring the state space of an SDL system with powerful methods and tools that will find virtually any kind of possible run-time errors that may be difficult to find with regular simulation and debugging techniques.

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* as well as *chapter 4, Tutorial: The SDL Simulator* before starting with the tutorial on the Validator.

Note: C compiler

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

Note: Platform differences

This tutorial, and all tutorials 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 Validator

In addition to simulating a system, it is also possible to validate the system using the SDL Validator. A validator can be used to automatically find errors and inconsistencies in a system, or to verify the system against requirements.

In the same way as for a simulator, you must generate an executable validator and start it with a suitable user interface.

Note:

In order to generate a validator 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 validator 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 quickly generate and start an executable validator

Quick Start of a Validator

A validator can be generated and started in the same way as described earlier for the simulator, i.e., by using the *Make* dialog and the *Tools* menu in the Organizer. However, we will now show a quicker way.

1. Make sure the system diagram icon is selected in the Organizer.

2. Click the *Validate* quick button. The following things will now happen, in rapid succession:

   – An executable validator is generated. Messages similar to when generating a simulator are displayed in the Status Bar, ending with "Analyzer done." This is the same action as manually using the *Make* dialog and selecting a validator kernel. If you like, you can verify that a validator kernel has been used by looking at the tail of the Organizer log.

   – A graphical user interface to the validator is started. The status bar of the Organizer will read "Validator UI started." This is the same action as manually selecting *Validator UI* from the *Tools* menu.

   – The generated validator is started. The Validator UI shows the message "Welcome to SDL VALIDATOR." This is the same action as manually using the *Open* quick button and selecting the executable validator (named `demongame_xxx.val` **(on UNIX)**, or `demongame_xxx.exe` **(in Windows)**, where the `_xxx` suffix is platform or kernel/compiler specific).

Note:

If you receive errors from the Make process (in the Organizer Log window) or if no Validator is started, do as follows:

- Open the *Make* dialog and change to a Validation kernel reflecting the C compiler used on your computer system, e.g. *gcc-Validation* or *Microsoft Validation*.
- Click the *Full Make* button and check that no errors where reported.
- Click the *Validate* quick button again. A Validator should now be started as described above.

The Validator UI looks like this:

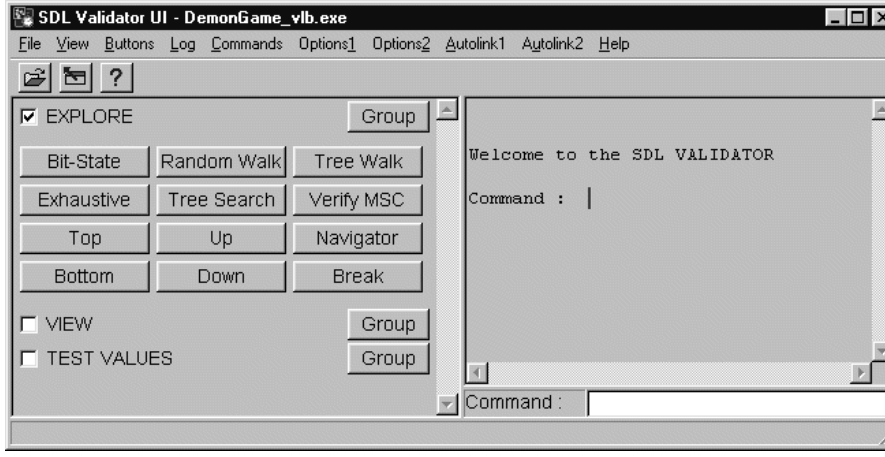

Figure 129: The main window of the Validator UI

As you can see, the graphical user interface of a validator is very similar to a simulator GUI, which you have learned to use in the previous exercises. However, the button modules to the left are different and a few extra menus are available.

A validator contains the same type of *monitor system* as a simulator. The only difference is the set of available commands.

When a validator 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. You can check this by viewing the process ready queue:

1. Locate the button module *View* in the left part of the window, and click the *Ready Q* button. The first entry in the ready queue is Main, waiting to execute its start transition.

   – If the *View* module appears to be empty, you have to click the toggle button to the left of the module's name. The button module is then expanded. You may collapse and expand any button module by using these toggle buttons:

Figure 130: A collapsed button module

– The buttons in the *View* module execute the same type of commands as those in the Simulator UI.

2. If required, resize the Validator UI window so that all button modules are visible. You may also reduce the width of the text area. In the exercises to come, you will have a number of windows open at the same time.

Basics of a Validator

Before you start working with the validator exercises, you should have an understanding about the basic concepts of the SDL Validator.

- When examining an SDL system using the validator, the SDL system is represented by a structure called a *behavior tree*. In this tree structure, a node represents a state of the complete SDL system. The collection of all such system states is known as the *state space* of the system.
- By moving around in the behavior tree, you can explore the behavior of the SDL system and examine each system state that is encountered. This is called *state space exploration*, and it can be performed either manually or automatically.
- The size and structure of the behavior tree is determined by a number of *state space options* in the validator. These options affect the number of system states generated for a transition in an SDL process graph, and the number of possible branches from a state in the behavior tree.

Navigating in a Behavior Tree

In this first exercise, we will explore the state space of the Demongame system by manually navigating in the behavior tree. The validator will then behave in a way similar to when running a simulator. However, there are also important differences, which will be pointed out.

By default, the validator is set up in a way that results in a state space as small as possible. In this set-up, a transition between two states in the behavior tree always corresponds to a complete transition in the SDL process graphs. Also, the number of possible branches from a state is limited to a minimum.

What You Will Learn

- To use the Navigator tool
- To get printed trace and GR trace

Setting Up the Exploration

When interactively exploring the behavior tree, a validator tool called the *Navigator* is used.

1. Start the Navigator by clicking the *Navigator* button in the *Explore* module. The Navigator window is opened:

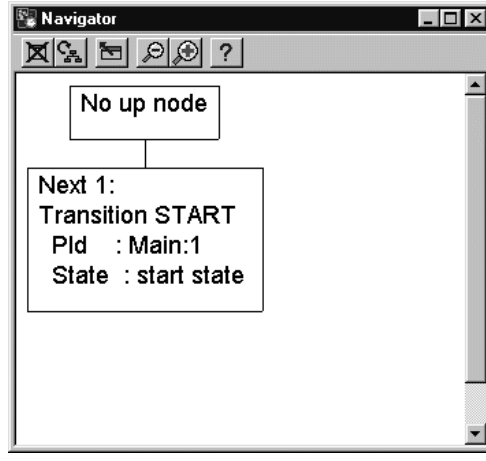

Figure 131: The Navigator tool

The Navigator shows part of the behavior tree around the current system state. In general, the upper box represents the behavior tree transition leading to the current state, i.e., the transition that just has been executed. The boxes below represent the possible tree transitions from the current state. They are labelled *Next 1, Next 2*, etc. and have not yet been executed.

Since the system now is in its start state, there is no *up node*. The only next node is the start transition of Main.

2. To be able to see the printed trace familiar from simulation, open the Command window from the *View* menu. (The trace is not printed in the main window of the validator.)

3. To switch on GR trace of SDL symbols, select *Toggle SDL Trace* from the *Commands* menu in the Validator window; SDL trace is now enabled. However, an SDL Editor will not be opened until the first transition is executed.

Using the Navigator

1. In the Navigator, execute the next transition by double-clicking on the *Next 1* node. The following happens, in order:

   – In the Navigator, the *Up 1* node shows the just executed transition, while the *Next 1* node shows the next possible transition, the start transition of Demon. You have now moved down to a system state in the next level of the behavior tree.

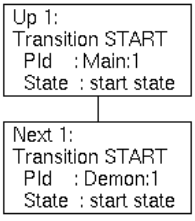

Figure 132: The last and next transition

   – An SDL Editor is opened and the symbols that were just executed becomes selected. Note the difference compared to the simulator, where the SDL Editor instead selects the next symbol to be executed.

– The Command window shows the printed trace for the executed transition, the start transition of Main (you may have to scroll or resize the window to see the trace):

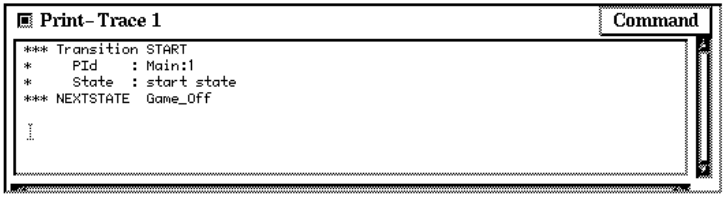

Figure 133: The printed trace for the executed transition

2. If needed, move and resize all opened windows to make them completely visible and still fit on the screen together.

3. Double-click the *Next 1* node to execute the next transition. The start transition of Demon is traced in the Command window and in the SDL Editor.

   At this stage, neither of the two active processes can continue without signal input: Main awaits the signal Newgame from the environment, and Demon awaits the sending of the timer signal T. These are the two transitions from the current state now shown in the Navigator as *Next 1* and *Next 2*. As you can see, the transitions in the boxes are described by the same type of information as in a printed trace.

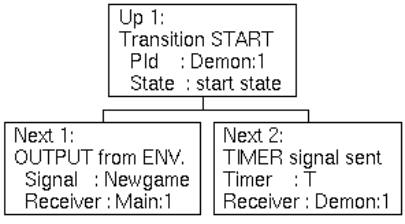

Figure 134: Transition descriptions in the Navigator

This means that the validator gives information of all possible transitions from the current system state, even though they have not been executed yet. (This information cannot easily be obtained when running a simulator.)

4. Send the timer signal by double-clicking the *Next 2* node. The Command window tells us that the timer signal is sent and the Navigator shows that the next transition is the input of the timer T.

5. Execute the next transition by double-clicking the *Next 1* node. This is where the dynamic error in the Demongame system occurs, as explained in the simulator tutorial earlier (see *"Dynamic Errors" on page 153 in chapter 4, Tutorial: The SDL Simulator*). Instead of showing the next transition, the Navigator displays the error message in the next box.

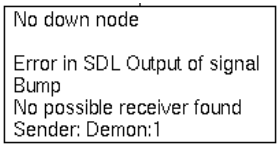

Figure 135: The dynamic error

– The error message can also be found in the tail of the Command window, if you scroll the Print-Trace module.

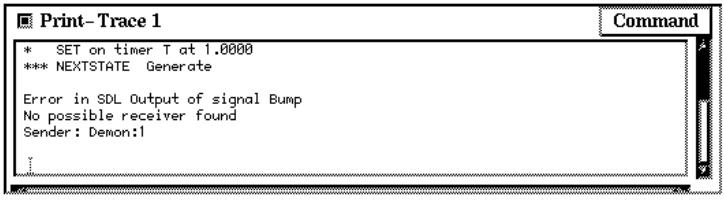

Figure 136: The tail of the Print-Trace module

We cannot go further down this branch of the behavior tree, since a reported error by default truncates the tree at the current state. Instead, we will back up to the state where we could select the output of Newgame.

6. Double-click the *Up 1* node to go back to the previous state. Repeat this action again to go to the state we were in after step 3 above. This way of backing up in the execution is not possible when running a simulator, as you may have noticed when running the Simulator tutorial.

You should also note that the *Next 2* node is marked with three asterisks “***”. This is used to indicate that this is the transition we have been backing up through:

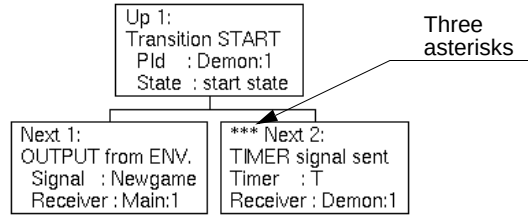

Figure 137: Marking a transition that has been backed through

7. Execute the *Next 1* transition instead. The printed trace shows that the signal Newgame was sent from the environment. The Main process is ready to receive the signal. Note that you do not have to send the signal yourself; this is taken care of automatically by the validator.

8. Execute the next transition. The printed trace and the SDL trace show that Main now is in the state Game_On. The Navigator displays the start transition of the newly created Game process.

9. Execute the start transition of Game. The Navigator will now show the different signal inputs that are required to continue execution: Endgame, Probe, Result, and the timer T.

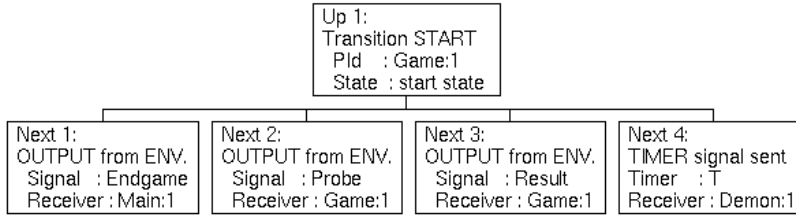

Figure 138: Signal inputs required for continued execution

If the number of transitions from a state is large, it may be difficult to see them all in the Navigator when a tree structure is used. To overcome this problem, you can change the display to a list structure.

10. Click the *Toggle Tree* quick-button to see how the list structure looks like. Now it is easier to see the possible signals.

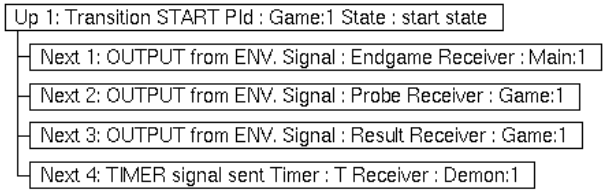

Figure 139: The list structure

11. Change back to the tree structure.

We will not continue further down in the behavior tree in this exercise. Figure 140 on page 199 shows the part of the behavior tree we have explored so far. The nodes in the figure represent states of the complete SDL system. Each node lists the active process instances that have changed since the previous system state, what process state they are in and the content of their input queues. The arrows between the nodes represent the possible tree transitions. They are tagged with a number and the SDL action that causes the transition. The arrow numbers are the same numbers as printed in the *Next* nodes in the Navigator.

Note that this is a somewhat different view of the behavior tree compared to the Navigator. In the Navigator, the nodes represent the tree transitions and the process states are not shown.

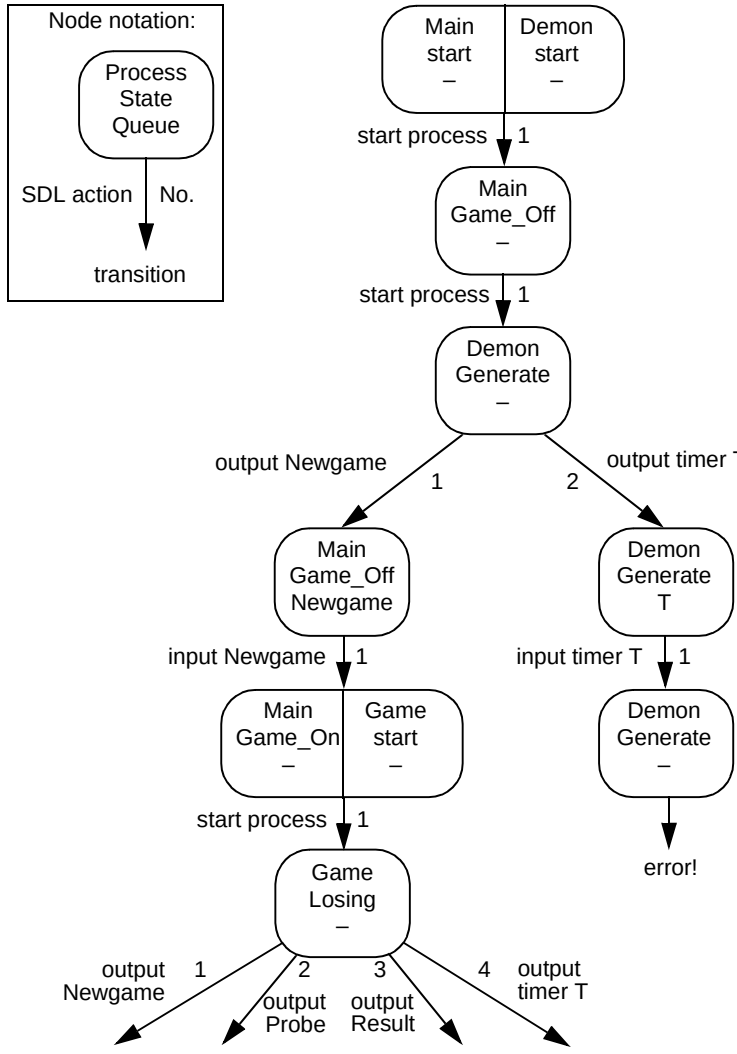

Figure 140: A Demongame behavior tree

More Tracing and Viewing Possibilities

In this exercise, we will take a look at some of the additional tracing and viewing possibilities of the validator.

What You Will Learn

- To print a complete trace from the start state
- To use the view commands
- To use the MSC trace facility
- To go to a state by using the path commands

Using the View Commands

1. Make sure you are still in the same state as after the last step in the previous exercise.

   To see a complete printed trace from the start state to the current state, you can use the **Print-Trace** command. As parameter, it takes the number of levels back to print the trace from.

2. On the input line of the Validator UI, enter the command `pr-tr 9` (you can use any large number). The trace is printed in the text area of the main window. This trace gives an overview of what has happened in the SDL system so far.

3. The SDL Validator supports the same viewing possibilities as the SDL Simulator. Click the *Timer List* button in the *View* module to list the active timer set by the Demon process.

4. Examine the GameP variable in the Main process by first setting the scope to the Main process (click the *Set Scope* button and select the Main process), and then clicking the *Variable* button and selecting the GameP variable.

   – You may also use the Watch window in the validator to continuously monitor the values of variables.

Using MSC Trace

In addition to textual and graphical traces, the validator can also perform an MSC trace.

1. First, turn off SDL trace by selecting *Toggle SDL Trace* from the *Commands* menu. Then, turn on MSC trace from the same menu. An MSC Editor is opened, showing a Message Sequence Chart for the trace from the start state to the current state.

   – You may also close down the SDL Editor to avoid having too many windows on-screen.

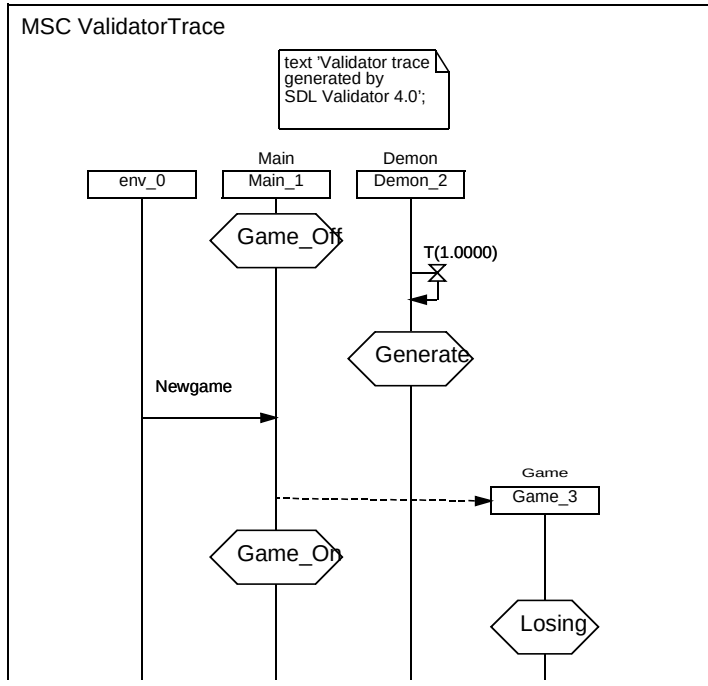

Figure 141: The current MSC trace

2. When the MSC appears, execute, with a double-click, one of the signal transitions in the Navigator, e.g. Probe. The message is appended to the MSC (but it is not yet consumed).

3. Go up a few levels in the Navigator.

Note how the selection in the MSC Editor changes to reflect the MSC event corresponding to the current state!

4. Go down again, but select a different path than before, i.e., send one of the other signals.

   Note how the MSC diagram is redrawn to show the new behavior of the system!

5. *Toggle MSC trace* off in the *Commands* menu. Unless other MSC diagrams were opened, the MSC Editor is closed.

Going to a State Using Path Commands

You can use the commands **Print-Path** and **Goto-Path** to return to a state where you have been before.

1. Execute the command Print-Path from the input line. The output represents the path taken in the behavior tree from the start state to the current state.

   ```
   Command : print-path
   1 1 1 1 1 3 0
   ```

   – The numbers in the path are the same as the transition numbers in the Navigator, and the arrow numbers shown in Figure 140 on page 199.

2. Go up a few levels in the Navigator.

3. In the text area, locate the path printed by the Print-Path command above (you may have to scroll the text area). **On UNIX**, select the numbers in the path with the mouse by dragging the mouse to the end of the line. Make sure you select the final zero.

4. In the input line, enter **`goto-path`** and the path printed by the Print-Path command. **On UNIX**, paste in the path numbers by positioning the mouse pointer at the end of the entered text and clicking the **middle** mouse button.

5. Hit `<Return>` to execute the command. You now end up in the previous state.

   – If you make an error while entering the path numbers, you can clear the input line by using the `<Down>` arrow key and try again.

Validating an SDL System

In the previous exercises, we have navigated manually in the behavior tree. We have also found an error situation by studying the Navigator and the printed trace in the Command window.

In this exercise, we will show how to find errors and possible problems by automatically exploring the state space of the Demongame system. This is referred to as *validating* an SDL system.

What You Will Learn

- To perform an automatic state space exploration
- To examine reported errors using the Report Viewer
- To change state space and exploration options
- To restrict the state space without affecting the behavior
- To check the system coverage of an exploration
- To use user-defined rules
- To perform a random walk exploration

Performing a Bit State Exploration

Automatic state space exploration can be performed using different algorithms. The algorithm called *bit state exploration* can be used to efficiently validate reasonably large SDL systems. It uses a data structure called a *hash table* to represent the system states that are generated during the exploration.

An automatic state space exploration always starts from the current system state. Since we want to explore the complete Demongame system, we must first go back to the start state of the behavior tree.

1. Go to the top of the tree by clicking the *Top* button in the *Explore* module.

2. Start a bit state exploration by clicking the *Bit-State* button. After a few seconds, a tool called the *Report Viewer* is opened. We will soon describe this window; in the meantime, just move it away from the main window.

3. For a small system such as Demongame, the exploration is finished almost immediately and some statistics are printed in the text area. They should look something like:

```
** Starting bit state exploration **
Search depth    : 100
Hash table size : 1000000 bytes

** Bit state exploration statistics **
No of reports: 1.
Generated states: 2569.
Truncated paths: 156.
Unique system states: 1887.
Size of hash table: 8000000 (1000000 bytes)
No of bits set in hash table: 3642
Collision risk: 0 %
Max depth: 100
Current depth: -1
Min state size: 68
Max state size: 124
Symbol coverage : 100.00
```

Of the printed information, you should note the following:

- `Search depth    : 100`
  The *search depth* limits the exploration; it is the maximum depth, or level, of the behavior tree. If this level is reached during the exploration, the current path in the tree is truncated and the exploration continues in another branch of the tree. It is possible to change the search depth by setting an option in the Validator UI.

- `No of reports: 1.`
  The exploration found one error situation. This error will be examined in the next exercise.

- `Truncated paths: 156.`
  The maximum depth was reached 156 times, i.e., there are parts of the behavior tree that were not explored. This is a normal situation for SDL systems with infinite state spaces. Demongame is such a system, since the game can go on forever.

- `Collision risk: 0 %`
  The risk for collisions was very small in the hash table that is used to represent the generated system states. If this value is greater than zero, the size of the hash table may have to be increased by setting an option; otherwise, some paths may be truncated by mistake. This situation will not occur in this tutorial.

– `Symbol coverage : 100.00`
  All SDL symbols in the system were executed during the exploration. If the symbol coverage is not 100%, the validation cannot be considered finished. This situation will occur in a later exercise.

Examining Reports

The error situations reported from a state space exploration can be examined in the Report Viewer. The Report Viewer window displays the reports in the form of boxes in a tree structure.

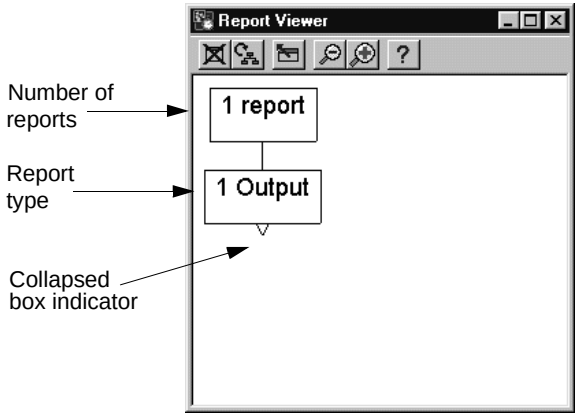

Figure 142: The Report Viewer

- The top box shows how many reports there are (in this case only one).
- On the next level in the report tree, there is one box for each type of report, stating the number of reports of that type.
- On the next level, it is possible to see the actual reports. However, this level of the tree is by default collapsed, indicated by the small triangle icon below the report type boxes.

1. To expand the report, double-click on the report type box *Output*. You will now see a box reporting the error we have found manually earlier. In addition, the tree depth of the error situation is shown.

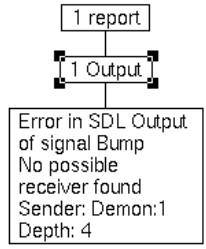

Figure 143: An expanded report

If you look in the Navigator and Command windows, you can see that the validator is still in the start state of the system, even though a state space exploration has been performed. We will now go to the state where the error has occurred.

2. Double-click the report box in the Report Viewer. The following things will now happen:

   – The printed trace of the error situation is displayed in the text area of the Validator UI and in the Command window.

   – The Navigator moves to the error state and displays the error.

   – An MSC Editor is opened, showing the MSC trace to the current state. You can see that the signal Bump was not received by any process, since the Game process has not yet been created. You should move the MSC Editor window so that it does not cover the other windows.

   Once you have used the Report Viewer to go to a reported situation, you can easily move up and down the path to this state. Simply use the *Up* and *Down* buttons in the *Explore* module, instead of double-clicking a node in the Navigator:

3. Move up two steps by using the *Up* button. Of the two transitions possible from this state, the one that is part of the path leading to the error is indicated by three asterisks "***" (see Figure 137 on page 197). This is the transition chosen when using the *Down* button.

4. Move up to the top of the tree (click the *Top* button in the *Explore* module). Move down again to the error by using the *Down* button repeatedly.

   Note that you do not have to chose which way to go when the tree branches. The path to the error is remembered by the validator until you manually chose another transition.

Exploring a Larger State Space

We will now run a more advanced bit state exploration, with a different setting of the state space options. This will make the state space much larger, so that more error situations can be found.

1. Go back to the top of the behavior tree (use the *Top* button).
2. In the *Options1* menu, select *Advanced*. This sets a number of the available state space options in one step, as you can see by the commands executed in the text area:

```
Command : def-sched all

Command : def-prio 1 1 1 1 1

Command : def-max-input-port 2
Max input port length is set to 2.

Command : def-rep-log maxq off
No log for MaxQueueLength reports
```

   Note that the Navigator now shows two possible transitions from the start state; this is an immediate effect of the larger state space.

3. In addition, we will increase the search depth of the exploration from 100 (the default) to 300. From the *Options2* menu, select *Bit-State: Depth*. In the dialog, enter `300` and click *OK*.

Figure 144: Specifying Depth = 300

Since the behavior tree becomes much larger with these option settings, the exploration will take longer to finish. We will therefore show how to stop the exploration manually.

4. Start a new bit state exploration. In the text area, a status message is printed every 20,000 transitions that are executed. Stop the exploration after one of the first status messages by pushing the *Break* button in the *Explore* module. The text area should now display something like this:

```
*** Break at user input ***

** Bit state exploration statistics **
No of reports: 2.
Generated states: 50000.
Truncated paths: 1250.
Unique system states: 21435.
Size of hash table: 8000000 (1000000 bytes)
No of bits set in hash table: 41557
Collision risk: 0 %
Max depth: 300
Current depth: 235
Min state size: 68
Max state size: 168
Symbol coverage : 100.00
```

Note:

If the exploration finishes by itself before you have had a chance to stop it manually, redo this exercise from step 1. on page 207 but increase the search depth even more, e.g. 400 or 500.

Note the following differences in the printed information compared to the previous exploration:

- `No of reports: 2.`
  The exploration found an additional error situation. This is an effect of more transitions being able to execute from each state in the behavior tree.

- `Max depth: 300`
  `Current depth: <number>`
  The exploration was at the printed depth in the behavior tree at the moment it was stopped. However, since the exploration uses a depth-first algorithm, the maximum depth of 300 was reached at an earlier stage. The exploration may be continued from the current depth if you wish to explore the remaining parts of the behavior tree.

5. In the Report Viewer, open the two report type boxes to see both reports with a double-click on each. The Report Viewer window should now look something like:

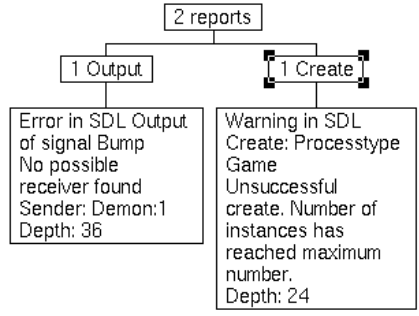

Figure 145: The two reports as displayed in the window

6. For now, just note on which depth each of the reported situations occurred; **do not** double-click any of the reports. (The depths may be different from the ones shown in the figure.)

7. Continue the exploration by clicking the *Bit-State* button again. A dialog is opened, asking if you would like to continue the interrupted exploration or restart it from the beginning.

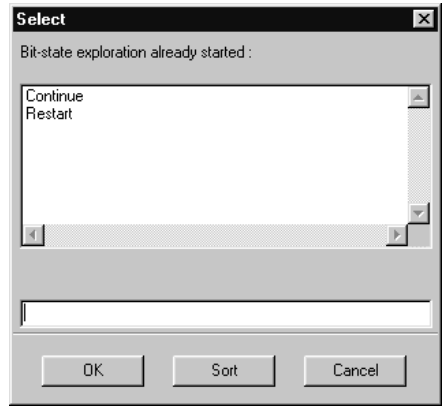

Figure 146: Continuing the exploration

8. In the dialog, select *Continue* and click *OK*. Wait for the exploration to finish by itself.

9. In the Report Viewer, open the two reports again. Note that the depth values have changed. This is because only one occurrence of each report is printed; the one found at the lowest depth so far.

10. Go to the state where an unsuccessful create of the Game process was reported (double-click the Create report).

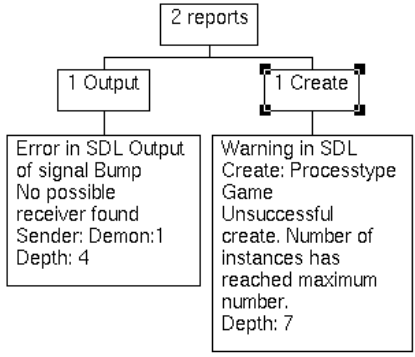

Figure 147: The report about an unsuccessful process create

11. To see what caused the unsuccessful create, look at the MSC trace.

    At the receipt of the last Newgame signal, the Main process attempts to create a Game process. However, the already active Game process has not yet consumed the previous GameOver signal, and has therefore not been terminated. Since you cannot have more than one instance of the Game process in the Demongame system, the process create could not be executed!

Restricting the State Space

The Validator makes it possible to limit the state space in several different ways. We will now explore one of these methods that in many cases is very efficient. This is done by using the **Define-Variable-Mode** command.

This command is used to instruct the Validator to ignore certain variables when matching states during the state space exploration. The

mode can for each variable be set to either “Skip” or “Compare”. The implication of setting the mode to “Skip” is that the search may be pruned even if a new state is encountered during the search. This happens if the only difference between the new state and a previously visited state is that the values of some of the skipped variables are different.

We will now apply this to our DemonGame system. The variable *Count* in the *Game* process keeps track of the current score for the game, and the value of this variable does not have any real impact on the behavior of the system. So, we will now instruct the validator to ignore this variable when performing a search.

1. Go to the top of the tree by clicking on the *Top* button.
2. Enter the command **`define-variable-mode`** in the command line in the Validator UI, select the *Game* process in the first dialog, the *Count* variable in the second dialog and *Skip* in the last dialog. You have now instructed the Validator to ignore the Count variable.
3. Start a bit state exploration by clicking on the *Bit-State* button. (Select to *Restart* the exploration if a dialog is opened.)
4. When the search stops compare it with the previous exploration. The only difference between the two explorations is that the second one ignores the Count variable. However, while the first exploration took a long time to finish, the second one only took a few seconds! The printed statistics show very small numbers in comparison.

The lesson to learn from this is that it in many cases it is possible to drastically reduce the time needed for explorations by checking the variables in the system. Look for variables that do not have any impact on the behavior (i.e. that does not influence decision statements or the expression used in an “output to” statement). Also look for variables that do not change their value during the exploration. This can for example be arrays that are initialized at system start up but then never changes (or at least not changed in the intended exploration). The mode for these types of variables should be set to “Skip”.

Checking the Validation Coverage

If the symbol coverage after an automatic state space exploration is less than 100%, the Coverage Viewer can be used to check what parts of the system that have not been executed. To attain a symbol coverage less than 100% for the Demongame system, we will set up the exploration in a special way.

1. Go to the top of the tree.
2. First, we need to restore the smaller, default state space. Select *Default* from the *Options1* menu. Note that the Navigator changes back to display only a single possible transition from the top node.
3. To avoid reaching all system states, we will reduce the search depth of the exploration from 100 to just 10. Use the *Bit-State: Depth* menu choice from the *Options2* menu and specify a maximum depth of `10`.
4. Start a bit state exploration. The printed statistics should now inform you that the symbol coverage is about 82%.
   - If the symbol coverage still is 100%, select *Reset* from the *Options1* menu and repeat steps 3 and 4 above.
5. To find out which parts of the Demongame system that have not been reached, open the Coverage Viewer from the *Commands* menu.

   A symbol coverage tree is displayed, showing all symbols which have not been executed yet.

6. Change to a transition coverage tree by clicking the *Tree Mode* quick-button.

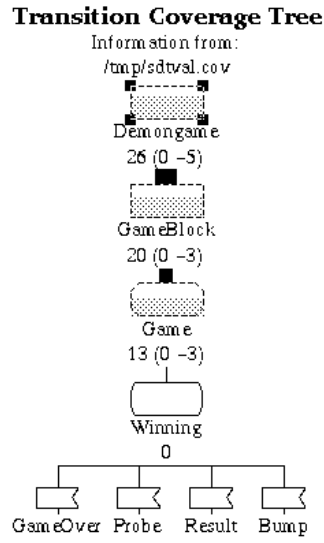

Figure 148: The transition coverage tree

You can now see that none of the transitions from the state Winning in the Game process has been executed. To explore this part of the system in the validator, you can go to the state Winning and start a new exploration from there. How to do this is explained in the following exercises.

Going to a State Using User-Defined Rules

To go to a particular system state, you could use the Navigator to manually find the state by studying the transition descriptions and the printed trace in the Command window. This can be both tedious and difficult, especially for larger systems than Demongame. Instead, we will show an easier way: by using a *user-defined rule*.

When performing state space exploration, the validator checks a number of predefined rules in each system state that is reached. It is when such a rule is satisfied that a report is generated.

In this exercise, we will show how to define a new rule to be checked during state space exploration. The rule will be used to find the state Winning in the Game process.

1. Make sure you still are at the top of the behavior tree.
2. Define a new rule by selecting *Define Rule* from the *Commands* menu. In the dialog that appears, enter the rule definition **`state(Game:1)=Winning`**

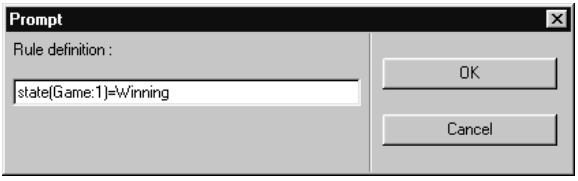

Figure 149: Specifying a new rule

   This very simple rule says that the state of the process instance Game:1 must be equal to Winning. By defining the rule, a report will be generated when a state space exploration reaches a state that satisfies the rule.

3. Start a bit state exploration. Since we have not changed any of the options since the last exploration the same statistics will be printed, with the exception that an additional report is generated.
4. From the Report Viewer, go to the reported situation where the user-defined rule was satisfied. You have now reached the first place in the behavior tree where the Game process is in the state Winning.
5. We now instruct the validator to use this state as the root of the behavior tree. To do this, enter the command **`define-root`** on the input line and select *Current* in the dialog.

We can now change options, define a new rule or load an MSC. These new settings will then be used in all explorations based on the new root. Also all list/goto-path commands will use the path from the new root and the MSC trace will give the trace from the new root.

6. Before continuing, do not forget to clear the user-defined rule. To do this, enter the command **`clear-rule`** on the input line.

In our case we will only clear the rule and start another type of state space exploration from this state; a random walk.

Performing a Random Walk

Apart from bit state exploration, there is another exploration method known as *random walk*. A random walk simply explores the behavior tree by repeatedly choosing a random path down the tree. This is mainly useful for SDL systems where the state space can be very large. But also for a small system like Demongame, it can be as effective as other exploration methods.

1. Start a random walk exploration from the current state by clicking the *Random Walk* button. From the printed statistics, you can see that the symbol coverage now has become 100%.

2. Load the Coverage Viewer with the new coverage information by selecting *Show Coverage Viewer* from the *Commands* menu. Change to transition coverage and display the whole tree. Note that all transitions have executed a large number of times. When the exploration selects a random path down the tree, there is no mechanism to avoid that already explored paths are explored once more. Therefore, the same transition may be executed any number of times

3. Exit the Coverage Viewer from the *File* menu.

4. Reset the system by selecting *Reset* from the *Options1* menu. You are now back at the top of the tree, and the root of the tree is reset to the original root, the start state of the system.

Verifying a Message Sequence Chart

Another main area of use for a validator is to verify a Message Sequence Chart. To verify an MSC is to check if there is a possible execution path for the SDL system that satisfies the MSC. This is done by loading the MSC and performing a state space exploration set up in a way suitable for verifying MSCs.

What You Will Learn

- To verify an MSC

Verifying a System Level MSC

In this exercise, we will verify one MSC made on the system level, i.e., an MSC that only defines signals to and from the environment. The name of the MSC file is `SystemLevel.msc` and is located in the same directory as the remaining files for the DemonGame example. The MSC is shown in the figure below.

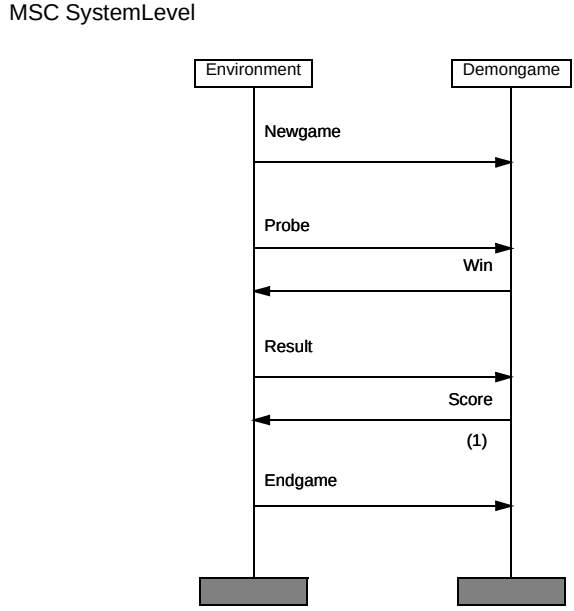

Figure 150: A system level MSC

1. Reset the system. This time do it by choosing *Restart* in the *File* menu. Choose “No” if you are asked to save options. The *Restart* command will actually terminate the running Validator and start it again.

2. Start an MSC verification by clicking the *Verify MSC* button. A file selection dialog is opened, in which you select the MSC to verify.

3. Select `SystemLevel.msc` and click *OK*. A state space exploration is now started, which is guided by the loaded MSC.

   In the printed statistics, note that the exploration is completed without any truncated paths. This is because the loaded MSC restricts the size of the behavior tree; only the parts dealing with the events in the MSC are executed. The maximum depth of it is not more than 20.

   Note the line that tells if the MSC was verified or violated:

   ```
   ** MSC SystemLevel verified **
   ```

In this case the MSC was verified, i.e., the behavior described in the MSC was indeed possible. In the Report Viewer, however, one (or two) of the reports is a *violation* of the loaded MSC, while the other one is a verification of the MSC. The exploration may very well find states that violate the MSC; it is the existence of states that verify the MSC that determines the result of the verification.

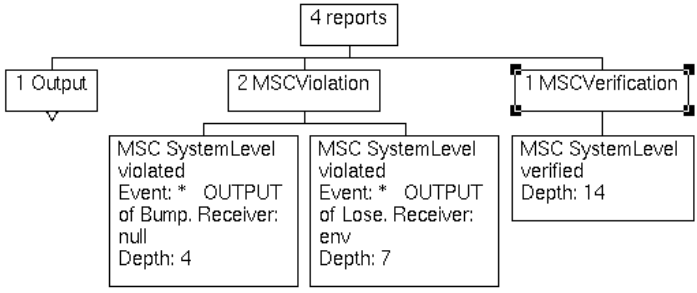

Figure 151: Violations and verifications of the MSC

4. Go to the state where the MSC was verified. The printed trace in the Command window shows that the Main process has received the Endgame signal, and sent the GameOver signal to the Game process:

```
*   OUTPUT of GameOver. Receiver: Game:1
*     Signal GameOver received by Game:1
```

5. Take a look at the MSC trace and compare it with the loaded MSC in Figure 150 on page 217. Note that the loaded MSC only defines signals to and from the environment and therefore is less detailed than the MSC trace. An MSC trace in the validator is always made on the process level.

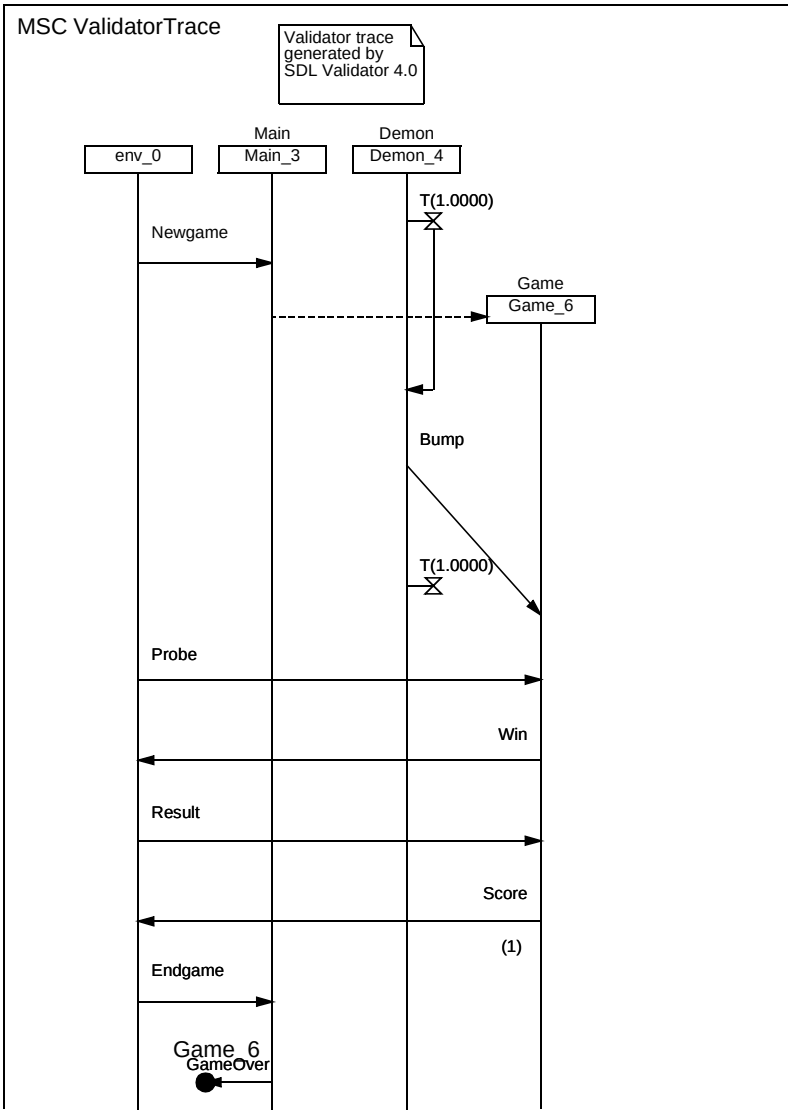

Figure 152: The MSC trace

The trace in the figure does not show the condition symbols that indicates the state of the processes.

Exiting the Validator UI

The first part of the validator tutorial is now finished. Close the validator windows in the following way:

1. To close the Navigator and the Report Viewer, click the *Close* quick button in these windows.
2. To close the Command window, select *Close* from the *File* menu.
3. Exit the Validator UI from the *File* menu. You may be asked in a dialog whether to save changes to the Validator options.

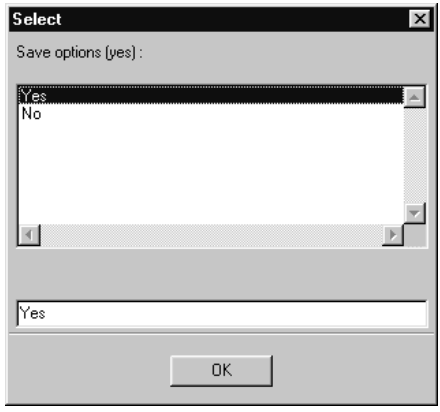

Figure 153: Saving changed options

4. If you select *Yes* and click *OK*, the option settings are saved in a file called `.valinit` **(on UNIX)**, or `valinit.com` **(in Windows)**. This file is read each time the Validator UI is started from the same directory, or when the validator is restarted or reset from the Validator UI. You should select *No* and click *OK*.

Using Test Values

In this final exercise we will explore the test value feature in the Validator. This feature is used to control the way the environment interacts with the system during state space exploration. In practise, the test values define what signals will be sent from the environment to the system, including the exact values of their parameters.

In this part of the Validator tutorial we will use another SDL system, the Inres system.

1. Copy the Inres system from the installation to a working directory of your own. Copy all files from the directory `$telelogic/sdt/examples/inres` **(on UNIX)**, or `C:\Telelogic\SDL_TTCN_Suite4.5\sdt\examples\inres` **(in Windows)**.

What You Will Learn

- To examine and use the automatically generated test values
- To manually change the test values

Using the Automatic Test Value Generation

When the Validator is started, test values for a number of SDL sorts are automatically generated. For example, all integer parameters will have the test values -55, 0 or 55. We will now take a look at the automatically generated test values for the Inres system.

1. Open the Inres system file from the Organizer's *File* menu. If any part of the existing DemonGame system needs saving, you are first prompted to do so before the *Open* file dialog appears. Locate the file `inres.sdt` that you have copied and open it.

2. Generate and start a Validator for the system by clicking on the *Validate* quick button; see "Quick Start of a Validator" on page 190 for more information. If you are asked in a dialog whether to start a new Validator UI or use an existing one, select the existing validator in the list and click *OK*.

3. Expand the *Test Values* button module in the Validator UI and make the window bigger so you can see all the buttons.

   The button module contains four rows of buttons. The top three buttons *List Value*, *Def Value* And *Clear Value* make it possible to define test values for each sort (data type) in the SDL system. The

middle row with the buttons *List Par*, *Def Par* and *Clear Par* handles test values for specific signal parameters. The bottom rows handles test values for entire signals.

4. Click on the *List Value* button to see what default test values have been generated. The following values should be listed:

```
Sort integer:
0
-55
55

Sort Sequencenumber:
zero
one

Sort IPDUType:
CR
CC
DR
DT
AK
```

As you can see, there were test values defined for the predefined sort *integer* and for two system specific enumerated sorts *Sequencenumber* and *IPDUType*. For enumerated types, all the values will by default be used as test values if there are 10 or less values. Note that only sorts that appear on parameters to signals to or from the environment are listed.

5. Click on the *List Signal* button to see what signals will be sent to the system based on the test values for the sorts.

You should now see a list of signals similar to the following. Note that there might be differences in the parameters to the MDATreq signal since this is computed using a random function that is depending on the compiler used.

```
ICONreq
IDATreq(0)
IDATreq(-55)
IDATreq(55)
IDISreq
MDATreq((. CR, zero, -55 .))
MDATreq((. CR, zero, -55 .))
MDATreq((. CC, one, 55 .))
MDATreq((. AK, one, 55 .))
MDATreq((. CR, one, 55 .))
MDATreq((. DT, one, -55 .))
MDATreq((. CC, one, 0 .))
```

```
MDATreq((. CC, one, -55 .))
MDATreq((. AK, one, 55 .))
MDATreq((. DR, one, 0 .))
```

The signals *ICONreq* and *IDISreq* have no parameters so there will only be one signal definition for each of these signals. The IDATreq signal has one integer parameter, and as you can see there will be three test values for this signal, one for each of the test values for the integer sort.

The MDATreq signal takes a parameter that is a structure with three fields: one IPDUType, one Sequencenumber and one integer. Whenever the Validator finds a structure, it tries to generate test values for the sort based on all combination of test values for each field. However, if the number of test values is larger than a maximum value, a randomly chosen subset is used instead. The maximum number is by default 10, but can be changed with the Define-Max-Test-Values command.

The consequence of this is that for the MDATreq signal, 10 different randomly chosen parameter values are generated.

Now, let us check how the test values influence the behavior of the system during state space exploration.

6. Start the Navigator by clicking on the *Navigator* button in the *Explore* button module.

7. Double-click on the down node 4 times (until there is more than one alternative down node).

   You should now have a choice between 11 different down nodes that each one represents an input from the environment to the SDL system. If you check the inputs more carefully, you will see that these 11 inputs correspond to the test values defined for the signals ICONreq and MDATreq.

   This is the way the test values have an impact on the state space exploration. Whenever a signal can be sent from the environment to the system, the Validator uses the test values defined for the signal to determine what parameters to use when sending the signal.

Changing the Test Values Manually

Now, we will use the other commands in the *Test Values* button module to manually change the test values.

1. Click on the *Top* button in the *Explore* module to return to the start state in the state space.

   First we will change the test values for integer to only test the values 1 and 99.

2. Click on the *Clear Value* button, select the *integer* type in the dialog and give the value '-' (a dash) in the value dialog. Dash indicates that we would like to remove all test values currently defined for the sort.

   Note that the Validator tries to recompute test values for various sorts and signals when you have changed the test values for integer. Since integer is used in a number of other sorts and signals, the Validator is now unable to compute test values for these sorts and signals.

3. Check the signal definitions that now is used by clicking on the *List Signal* button. The current signal definitions should now be:

   ```
   ICONreq
   IDISreq
   ```

   Since there are no test values for integer, only the signals that does not contain integer parameters are listed. In this case this means that only ICONreq and IDISreq would have been sent to the system from the environment if you would start an exploration.

4. Click on the *Def Value* button, select *integer* in the sort dialog and give the value `1` in the value dialog.

5. Click on the *Def Value* button once more. Select integer in the sort dialog again, but this time give the value `99` in the value dialog.

6. Click on the *List Signal* button to check the signal definitions and make sure that the signals with integer parameters are once again on the list. This time with the test values 1 and 99.

```
ICONreq
IDATreq(1)
IDATreq(99)
IDISreq
MDATreq((. AK, zero, 1 .))
MDATreq((. DR, zero, 99 .))
MDATreq((. AK, one, 1 .))
MDATreq((. DR, one, 1 .))
MDATreq((. DR, zero, 99 .))
MDATreq((. AK, zero, 1 .))
MDATreq((. AK, one, 99 .))
MDATreq((. AK, zero, 99 .))
MDATreq((. AK, one, 1 .))
MDATreq((. CR, one, 1 .))
```

You have now explored some on the most frequently used test value features in the Validator. There are also possibilities to set test values for specific parameters and to enumerate all signal definitions manually. You can find more information about this in the section "Defining Signals from the Environment" on page 2375 in chapter 54, *Validating a System, in the User's Manual*.

Exiting the Validator

To exit the Validator follow the same steps as before:

1. Select *Exit* from the *File* menu.
2. Choose *Yes* when asked whether you want to save the new options or not. To select *Yes* in this dialog implies that commands that recreates your new test value definitions will be saved in the file `.valinit` **(on UNIX)** or `valinit.com` **(in Windows)**.

So Far...

By practicing this and the previous tutorials, you have learned the basics of the SDL suite and we hope you have enjoyed the "tour". The examples you have been practising on, the DemonGame and Inres systems, are however rather simple. To deepen your knowledge about the SDL suite components, 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*.