Chapter

50 *The SDL Simulator*

This chapter is a reference to the simulator user interface.

For a guide to how to use the simulator, see chapter 51, *Simulating a System*.

The Simulator Monitor

A simulator generated by the SDL suite consists of two main parts; the application itself (the simulated SDL system), and an interactive monitor system. The monitor system is the interface between the user and the simulated system. For more information on the structure of a simulator and the features of the simulator monitor, see "Structure of a Simulator" on page 2166 in chapter 51, *Simulating a System*.

Monitor User Interfaces

Two different user interfaces are provided for the simulator monitor, a textual and a graphical.

The textual interface only allows commands to be entered from the keyboard, using the syntax described in the section "Syntax of Monitor Commands" on page 2064.

The graphical interface still allows commands to be entered from the keyboard in the same way, but also provides buttons, menus and dialogs for easy access to commands and other features.

The textual interface is invoked by executing the generated simulator directly from the operating system prompt. This is called running a simulator in *textual mode*. When started, the simulator responds with the following text:

```
Welcome to SDL SIMULATOR. Simulating system <system>

Command :
```

Another prompt may appear if the SDL system contains external synonyms. For more information, see "Supplying Values of External Synonyms" on page 2172 in chapter 51, *Simulating a System*.

> **Note:**
>
> Before a simulator can be run in textual mode **on UNIX**, a command file must be executed from the operating system prompt. The file is called `telelogic.sou` or `telelogic.profile` and is located in the binary directory that is included in the user's path.
> For csh-compatible shells: **`source <bin dir>/telelogic.sou`**
> For sh-compatible shells: **`. <bin dir>/telelogic.profile`**

The graphical interface, known as the *Simulator UI*, runs in a separate window. It is started from the Organizer by selecting *SDL > Simulator UI* from the *Tools* menu. The Simulator UI is described in "Graphical User Interface" on page 2130.

If a file called `siminit.com` exists in the current directory an implicit Include-File command will be done on this file at startup.

Activating the Monitor

Commands can be issued to the interactive monitor system when it becomes active. The simulator's monitor system becomes active:

- When the simulator is started.
- When the last command was Next-Transition or Next-Visible-Transition and the transitions initiated by this command have completed.
- When the last command was Next-Symbol or Step-Symbol and one SDL symbol has been executed.
- When the last command was Next-Statement or Step-Statement and one SDL statement has been executed.
- When the last command was Finish and the currently executing procedure has returned.
- When the last command was Proceed-Until and the value of the simulation time is, for the first time, equal to the time given as a parameter to the command.
- When the last command was Proceed-To-Timer and all transitions up to the next timer output have been executed.
- Immediately before a transition matching a transition breakpoint set by the command Breakpoint-Transition.
- Immediately before a symbol matching a symbol breakpoint set by the command Breakpoint-At.
- Immediately after an output symbol that contains an output matching an output breakpoint set by the command Breakpoint-Output.

- Immediately after a symbol or assignment statement where a variable is changed matching a variable breakpoint set by the command Breakpoint-Variable.
- When there is no transition that can be executed, that is, the system is completely inactive. (**On UNIX**, if no environment is present and the command Go-Forever was issued.)
- Immediately after a symbol that included an SDL semantic error.
- In the Simulator UI, when the *Break* button is clicked in the *Execute* button module; in textual mode, when `<Return>` is pressed during the output of trace information. The interactive monitor then becomes active directly after the current symbol has been executed.

Note:

No other characters may be typed before `<Return>` **is pressed**.

Syntax of Monitor Commands

The monitor system waits for commands from the user by issuing the following prompt:

```
Command :
```

The syntax to be used for the commands and their parameters are described in the following.

Command Names

A command name may be abbreviated by giving sufficient characters to distinguish it from other command names. A special character, the hyphen (-), is used to separate command names into distinct parts. Any part may be abbreviated as long as the command name does not become ambiguous.

Consider, as an example, the command name List-Process. The command name may be typed **`List-P`** or **`L-P`**. However, if only **`List`** is typed, the monitor system will respond with the message

```
Command was ambiguous, it might be an abbreviation
of:
```

followed by a list of all commands starting with “List,” since the command cannot be distinguished from, for example, the commands List-Ready-Queue and List-Timer.

There is no distinction made between upper and lower case letters when matching command names.

Parameters

Command parameters are separated by one or several spaces, carriage returns or tabs. A colon is also accepted between a process name and an instance number, when a PId value is required. If the parameter list following a command name is not complete the interactive monitor system will ask for the missing parameters by prompting with the type of the expected parameters.

Parameters may be abbreviated as long as the parameter value does not become ambiguous. There is no distinction made between upper and lower case letters.

Note:

The SDL keywords **Sender**, **Parent**, **Self** and **Offspring** may **not be abbreviated**.

Underscores in SDL Names

Command parameters that are SDL names may also be abbreviated, as long as the abbreviation is unique. The underscore character ‘_’ is used to separate names into distinct parts in the same way as the hyphen character (-) is used with command names. If a name is equal to the first part of another name, as for example Wait and Wait_For_Me, then any abbreviation of Wait is ambiguous. However, if all characters of the shorter name are given (Wait in this example), this is considered as an exact match and the short name will be selected.

The abbreviation facility means that the introduction of underscores in SDL names simplifies simulation. If, for example, you were considering if two signals should be named GenerateOn and GenerateOff or if they should be named Generate_On and Generate_Off, the last alternative is preferable for simulations, as the abbreviations G_On and G_Off are likely to be unique.

Matching of Parameters

When a (possibly abbreviated) parameter name is entered and is to be matched with an unabbreviated name, only names in the entity class of interest are considered. That is, if the monitor expects a process name as parameter, only the names denoting process types will be part of the search for the full name.

Signal names and timer names are in the same entity class; process formal parameters and process variables are in the same entity class; each other type of name (process, procedure, state, block,...) is in an entity class of its own.

Knowledge of previous parameters is used to narrow the search for a given parameter name. Consider for example the command Output-Via, which takes two parameters, a signal and a channel. The channel name is then only searched for among the channels the given signal can be sent via.

Qualifiers

Names can still cause problems, if, for example, there are two process types with the same name in two different blocks, or if the system and a block contain signal definitions with the same name.

In the first situation the process name will always be ambiguous and in the second case the system's signal will always be used. To solve cases like this, qualifiers with the same syntax as in SDL can be used. To reach a process P defined in block B in system S, the following notations can be used:

```
System S / Block B P
<<System S / Block B>> P
```

The words "system," "block," "process," "procedure," and "substructure" in the qualifier **may not be abbreviated**, while all names of, for example, blocks and processes may be abbreviated according to the usual rules. It is only necessary to give those parts of the qualifier that make the qualifier path unique (this is an extension of the qualifier concept in SDL). The slash (/) in a qualifier may be omitted and replaced by one or more spaces. The angle brackets that are part of the qualifier when printed may be omitted when entering the qualifier.

Default Parameters

Some command parameters may be omitted, indicating a default value for the parameter. To accept a default parameter value at the prompt, just press <Return>.

If the parameter is given on the same line as the command name and/or other parameters, type a hyphen '-' to indicate a default parameter value.

Example 307: Default Parameters in Simulator Command

Consider as an example the command Breakpoint-Transition, which takes at least eight parameters. To specify default values for all parameters, except the first and fourth parameter:

```
Breakpoint-Transition P - - State1 - - - -
```

Signal and Timer Parameters

When the parameters of a signal instance or a timer instance are to be entered, the interactive monitor system will ask for the parameters one by one, in the same way as for command parameters. The parameters can also be entered directly after the signal or timer name, possibly enclosed by parenthesis.

Example 308: Signal and Timer Parameters in Simulator Command –

```
Signal name : S
Parameter 1 (Integer) : 3
Parameter 2 (Boolean) : true
```

The same specification could also be given as:

```
Signal name : S 3 true
```

or as:

```
Signal name : S (3 true)
```

When entering signal parameters it is not necessary to give all parameter values. By entering '-' at the parameter's position, the parameter is given a "null" value (i.e. the computer's memory for the value is set to 0). By entering ')', the rest of the parameters are given "null" values.

Example 309: Signal Parameters in Simulator Command ––––––––––

```
Signal name : S -, true
```

will give the first parameter a “null” value.

```
Signal name : S (3, -)
```

will give the second parameter a “null” value. Could also be given as:

```
Signal name : S (3)
```

––

Errors in Commands

If an error in a command name or in one of its parameters is detected, an error message is printed and the execution of the command is interrupted, that is, a command is either executed completely or not at all. **On UNIX**, this is the only way to cancel a command that has been entered, when not using the simulator’s graphical user interface.

Context-Sensitive Help

By typing a question mark (‘?’), a context-sensitive help is obtained. The monitor will respond to a ‘?’ by giving a list of all allowed values at the current position, or by a type name, when it is not suitable to enumerate the values. After the presentation of the list, the input can be continued.

If no default value exists for the requested parameter, the list of all allowed values is also printed if the user simply presses `<Return>` at the prompt. In these cases, there is no need to type ‘?’.

Input and Output of Data Types

The syntax of literals of the predefined data types is in most cases obvious and follows the SDL definition of literals. There are, however, some extensions that are described below. As an option, it is also possible to use the ASN.1 syntax for values. On input the monitor system can manage both value notations, while there are commands in the monitor to select the type of output to be produced (SDL-Value-Notation and ASN1-Value-Notation).

Integer, Natural Values

The format of integers conforms exactly with the SDL and the ASN.1 standard, that is, an integer consists of a sequence of digits, possibly preceded by a '+' or '-'. However, with the command Define-Integer-Output-Mode it is possible to define the base of integers on output (decimal, hexadecimal, octal), which also affects how they may be entered. Hexadecimal values are preceded with "0x", and octal values are preceded with '0' (a zero).

Boolean Values

Boolean values are entered (and printed) as `true` and `false`, using upper or lower case letters. Abbreviation is allowed on input. In ASN.1 mode the value is printed in capital letters (`TRUE`, `FALSE`).

Real Values

The SDL literal syntax of real values has been extended to include the E-notation for exponents, in the same way as in many programming languages.

Example 310: Real Values in SDL Syntax ––––––––––––––––––––

The real number $1.4527 * 10^{24}$ can be written `1.4527E24`

The real number $4.46 * 10^{-4}$ can be written `4.46E-4`

––

The syntax for real values in ASN.1 is easiest described by an example:

Example 311: Real Values in ASN.1 Syntax ––––––––––––––––––––

```
{mantissa 23456, base 10, exponent -3}
```

This is the value 23.456.

Time, Duration Values

The format for Time and Duration values follows the SDL standards, i.e. real values without E-notation, with one extension. On input time values can either be absolute or relative to NOW. If the time value is given without a sign an absolute time value is assumed, while if a plus or minus sign precedes the value, a value relative to NOW is assumed.

Example 312: Time Values in SDL Syntax ––––––––––––––––––––

`123.5` is interpreted as 123.5

`+5.5` is interpreted as NOW + 5.5

`-8.0` is interpreted as NOW - 8.0

Character Values

Character values are entered and printed according to the SDL standard, including the literals for the non-printable characters.

Example 313: Character Values in SDL Syntax ––––––––––––––––––

```
'a', '3', '''', ' ', NUL, DEL
```

On input, the quotes may be omitted for all characters except ‘ ’ (space) and ‘[‘. A ‘should be entered as ‘’’.

Charstring Values

Charstring values can be entered and printed according to the SDL standard, that is, a single quote (’) followed by a number of characters followed by a single quote. Any quote (’) within a charstring has to be given twice. On output a non-printable character within a charstring is

printed as a single quote followed by the character literal followed by a single quote.

The ASN.1 syntax for Charstring is similar to the SDL syntax. The delimiter character ‘ (single quote) is however replaced by the character “ (double quote).

Example 314: Charstring Values in SDL Syntax ––––––––––––––––––

`''`	An empty string
`'abc'`	A string of three characters
`'a'NUL'c'`	The second character is NUL

––

PId Values

Apart from the value null, which is a valid PId value, a PId value consists of two parts, the name of the process type and an instance number, which is an integer greater than 0.

The first process instance of a process type that is created will have instance number 1, the second that is created will have instance number 2, and so on. The syntax is Name:No, where Name is the process name and No is the instance number.

On input the process name and the instance number may, as an alternative, be separated by one or more spaces, if the command parameter is a PId value. In the same circumstances the instance number is not necessary (and will not be prompted for) if there is only one process instance of the process type. If, however, the command parameter is a unit that might be a process type or a process instance, only a colon (‘:’) is allowed between the process name and the instance number and the colon must follow directly after the process type name. Examples of such situations are the unit parameter in Set-Trace and Signal-Log.

On output a PId value may be followed by a plus sign (‘+’), which indicates that the process instance is dead; that is, has executed a stop action. The plus sign is chosen as it is reminiscent of the ‘†’ character.

Bit

The Bit sort contains two values, 0 and 1. This syntax is used for input and output.

Bit_String

For Bit_String values the following syntax is used:

```
'0110'B
```

The characters between the two single quotes must be 0 or 1. On input the syntax for Octet_String, see below, can also be used.

Octet

The syntax used for an octet value is two HEX digits. Examples:

```
00   46   F2   a1   CC
```

The characters 0-9, a-f, and A-F are allowed.

Octet_String

The syntax for Octet_String is the following

```
'3A6F'H
```

Each pair of two HEX values in the string is treated as an Octet value. If there is an odd number of characters an extra 0 is inserted last in the string.

Object_Identifier

The sort Object_Identifier is in SDL treated as a String(Natural). This means that the syntax, in case SDL value notation is used, will be:

```
(. 2, 3, 11 .)
```

On input the items in the list should be separated by a comma and/or spaces - tabs. If ASN.1 value notation has been selected, the syntax will be:

```
{ 2 3 11 }
```

On input the items in the list should be separated by a comma and/or spaces - tabs.

Enumerated Values

Types that in SDL are defined as an enumeration of possible values can be entered and printed using the literals of the SDL data type definition. On input, the literals can be abbreviated as long as they are unique.

STRUCT Values

A struct value is entered and printed as the two characters "(." followed by a list of components followed by the two characters ".)". The components should, on input, be separated by a comma and/or a number of spaces (or carriage returns or tabs). Example:

```
(. 23, true, 'a' .)
```

If ASN.1 syntax is used, the component names will also be present. Example:

```
{ Comp1 2, Comp2 TRUE, Comp3 'a' }
```

On input the components using ASN.1 syntax may come in any order. Components not given any value will have the value 0, whatever that means for the data type.

On input the components using ASN.1 syntax may come in any order.

Optional components that are not present, will not be printed. In SDL syntax that means an empty position between two commas.

#UNION Values

A #UNION value uses, in SDL value notation, the same syntax as a struct value. However, only the tag value and the active component is printed. In ASN.1 value notation, #UNION values use the Choice value syntax described below.

Choice Values

The syntax for a choice value is `ComponentName:ComponentValue`. If, for example, a choice contains a component C1 of type integer, then

```
C1:11
```

is a valid choice value.

#UNIONC Values

The syntax for a #UNIONC value is ! `ComponentName ComponentValue` or -> `ComponentName ComponentValue`. If, for example, a #UNIONC contains a component C1 of type integer, then

```
! C1 11
```

is a valid #UNIONC value. However, this syntax will not be used when printing, since the value contains no tag and hence it is not possible to know which component to use.

Array Values

An array value is entered and printed as "(:" followed by a list of components followed by a ":)". The components should, on input, be separated by a comma and/or a number of spaces (or carriage returns or tabs). Note that there should also be a space between the last component and the terminating ":)". In ASN.1 syntax, '{' and '}' are used as delimiters.

There are two syntaxes for array components depending on the implementation that the SDL to C Compiler has selected for the array implementation. This selection is described in *"Array" on page 2601 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler*. These easiest way to determine which syntax to use on input is to look at a variable of the array sort.

- If an array is a simple array (i.e. index type is a simple type with one range condition and a limited range), the SDL syntax for an array value is according to the following example:

  ```
  (: 1, 10, 23, 2, 11 :)
  ```

  If ASN.1 value notation is selected, replace "(:" and ":)" by '{' and '}'.

- If the array is a general array, the a syntax according to the following example should be used:

  ```
  (: (others:2), (10:3), (11:4) :)
  ```

  This would mean that for index 10 the value is 3, for index 11 the value is 4, and for all other indexes the value is 2. On input the commas, the parenthesis, and the colons in the components may be replaced by one or more spaces (or carriage returns or tabs).

For simple arrays the second syntax is also accepted. If the first syntax is used for simple arrays it is not mandatory to enter all values for the array components; by entering ":)" or "}" the rest of the components are set to a "null" value (i.e. the computer's memory for the value is set to 0).

String Values

A string value starts with "(." and ends with ".)". The components of the string is then enumerated, separated with commas. Example:

```
(. 1, 3, 6, 37 .)
```

On input the commas can be replaced by one or more spaces (or carriage returns or tabs). In ASN.1 syntax, '{' and '}' are used as delimiters instead of "(." and ".)".

Powerset Values

A powerset value starts with a '[' and ends with a ']'. The elements in the powerset is then enumerated, separated with commas. Example:

```
[ 1, 3, 6, 37 ]
```

On input the commas can be replaced by one or more spaces (or carriage returns or tabs).

Bag Values

A bag value starts with a '{' and ends with a '}'. The elements in the bag is then enumerated, separated with commas. Example:

```
{ 1, 3, 6, 37 }
```

On input the commas can be replaced by one or more spaces (or carriage returns or tabs). If the same value occurs more than once, then this value is in SDL syntax not enumerated several times. Instead, the number of occurrences is indicated after the value. Example

```
{ 1, 3:4, 6:2, 37 }
```

This is identical to

```
{ 1, 3, 3, 3, 3, 6, 6, 37 }
```

In ASN.1 syntax, each member is given explicitly according to the last example. On input items are separated by comma and/or one or more

spaces. It is also allowed to mention the same value several times, with or without a number of items each time.

Ref Values

There are two possible syntaxes for Ref values. Either the pointer address as a HEX value, or the value of the data area referenced by the pointer. The value Null is printed as `Null` in both syntaxes. In the monitor system two commands are available to determine the syntax to be used (REF-Address-Notation and REF-Value-Notation). On input both syntaxes are allowed independently of what syntax that has been selected.

Example of the address notation:

```
HEX(23A20020)
```

In the value notation the keyword `new()` is used to indicate a reference to a value. If, for example, the following data types are available in SDL

```
newtype Str struct
  data integer;
  next StrRef;
endnewtype;

newtype StrRef Ref(Str)
endnewtype;
```

then a value of the StrRef type can look like the examples below:

```
Null
new( (. 1, Null .) )
new( (. 1, new( (. 2, Null .) ) .) )
```

The last example is a linked list with two elements. To handle recursive data structures, e.g. graphs, a special syntax is used: `old n`, where `n` is an integer. Example:

```
new( (. 1, new( (. 2, old 1 .) ) .) )
```

The notation `old 1` in the example above means a reference to the same data area as the first `new()` notation refers to. So here we have two structs with next pointers referring to the other struct value. `Old n` would refer to the n:th `new()` seen from the left of the expression.

The same applies for Own and ORef values.

Monitor Commands

This section provides an alphabetical listing of all available commands in the simulator monitor. Simulator UI commands are described in "SimUI Commands" on page 2156.

@ (Keyboard Polling)

Parameters:

(None)

The monitor will repeatedly look at the keyboard for a carriage return, indicating an immediate break in the execution of the current transition. Using the command @, this facility can be turned off. Each time the command @ is entered the monitor toggles between having the keyboard polling for carriage return on and off. At start up keyboard polling is on.

> **Note:**
>
> **No other characters** may be typed together with the carriage return. If some other characters are typed by mistake, please delete them before typing the carriage return.

To turn keyboard polling off is useful in some situations, for example if a user wants to paste a sequence of commands into the monitor. If the sequence contains an empty line this might cause an unwanted interrupt.

? (Interactive Context Sensitive Help)

Parameters:

(None)

The monitor will respond to a '?' (question mark) by giving a list of all allowed values at the current position, or by a type name, when it is not suitable to enumerate the values. After the presentation of the list, the input can be continued.

Typing '?' at the prompt level gives a list of all available commands, after which a command can be entered. For further help, see the command Help.

ASN1-Value-Notation

Parameters:

(None)

The value notation used in all outputs of values is set to ASN.1 value notation. See also the command SDL-Value-Notation.

Assign-Value

Parameters:

```
[ '(' <PId value> ')' ]
<Variable name> <Optional component selection>
<New value>
```

The new value is assigned to the specified variable in the process instance, service instance, or procedure given by the current scope. Sender, Offspring, and Parent may also be changed in this way, but their names may not be abbreviated.

It is, in a similar way as for the command Examine-Variable, possible to handle components in structured variables (struct, strings, array, and Ref) by appending the struct component name or a valid array index before the value to be assigned. Nested structs and arrays can be handled by entering a list of index values and struct component names.

If a PId is given within parenthesis the scope is temporarily changed to this process instance instead.

Breakpoint-At

Parameters:

```
<SDT reference> <Optional breakpoint commands>
```

A breakpoint is defined at the symbol specified by the SDT reference. If the execution reaches the symbol, the monitor becomes active immediately before that symbol. SDT references may be obtained by using the *Show GR Reference* command in an SDL Editor, see "Show GR Reference" on page 1961 in chapter 44, *Using the SDL Editor*.

The <Optional breakpoint commands> parameter can be used to give monitor commands that should be executed when the breakpoint is triggered. Monitor commands should be separated by `" ; "`, i.e. space semicolon space.

Breakpoint-Output

Parameters:

```
<Signal name>
<Receiver process name> <Receiver instance number>
<Sender process name> <Sender instance number>
<Counter>
<Optional breakpoint commands>
```

A breakpoint is activated and a breakpoint condition is defined. If a breakpoint condition is matched by an output, the monitor becomes active immediately after the symbol containing the output. The breakpoint condition defines one or several outputs and is specified by the parameters. Any of the parameters may be omitted, which implies that any value will match the missing fields in the breakpoint condition. Initially no breakpoints are active.

The <Counter> parameter is used to indicate how many times the breakpoint condition should be true before the monitor should become active. Default value for this parameter is 1, which means that the monitor should be activated each time the breakpoint condition is true.

The <Optional breakpoint commands> parameter can be used to give monitor commands that should be executed when the breakpoint is triggered. Monitor commands should be separated by `" ; "`, i.e. space semicolon space.

Breakpoint-Transition

Parameters:

```
<Process name> <Instance number> <Service name>
<State name> <Signal name> <Sender process name>
<Sender instance number> <Counter>
<Optional breakpoint commands>
```

A breakpoint is activated and a breakpoint condition is defined. If a breakpoint condition is matched by a transition, the monitor becomes active immediately before that transition is started. The breakpoint condition matches one or several transitions and is specified by the parameters. Any of the parameters may be omitted, which implies that any value will match the missing fields in the breakpoint condition. Initially no breakpoints are active.

The <Counter> parameter is used to indicate how many times the breakpoint condition should be true before the monitor should become active. Default value for this parameter is 1, which means that the monitor should be activated each time the breakpoint condition is true.

The <Optional breakpoint commands> parameter can be used to give monitor commands that should be executed when the breakpoint is triggered. Monitor commands should be separated by `" ; "`, i.e. space semicolon space.

Breakpoint-Variable

Parameters:

```
<Variable name> <Optional breakpoint commands>
```

A breakpoint is defined on the specified variable in the process instance given by the current scope. If the variable's value is changed, the monitor becomes active immediately after the symbol or assignment statement where the value is changed. The value is only checked between symbols and between assignment statements in tasks.

The breakpoint is also triggered when the variable no longer exists, i.e. the PId containing the variable is stopped or the procedure containing the variable has reached its end. In this case, the breakpoint is automatically removed.

The <Optional breakpoint commands> parameter can be used to give monitor commands that should be executed when the breakpoint is triggered. Monitor commands should be separated by `" ; "`, i.e. space semicolon space.

Call-Env

Parameters:

```
(None)
```

The Call-Env command performs one call of the function `InEnv`. See the command Start-Env for more details.

> **Note:**
>
> This command and the commands Start-Env and Stop-Env are only available when the SDL to C Compiler is used to generate applications.

Call-SDL-Env

Parameters:

```
(None)
```

The command makes one call to the function that looks for incoming signals from the SDL suite communication mechanism. This means that all signals that have come to the simulator will be sent to their appropriate receiving process instances. See the command Start-SDL-Env for more details.

Cd

Parameters:

```
<Directory>
```

Change the current working directory to the specified directory.

Clear-Coverage-Table

Parameters:

```
(None)
```

This command is used to reset the coverage table to 0 in all positions, which means restart counting coverage from now.

Close-Signal-Log

Parameters:

```
<Entry number>
```

Stops the signal log with the specified entry number and closes the corresponding log file; see the command Signal-Log for more details. Entry numbers assigned by the command List-Signal-Log should be used.

Command-Log-Off

Parameters:

```
(None)
```

The command log facility is turned off; see the command Command-Log-On for details.

Command-Log-On

Parameters:

```
<Optional file name>
```

The command enables logging of all the commands given in the monitor. The first time the command is entered a file name for the log file has to be given as parameter. After that any further Command-Log-On commands, without a file name, will append more information to the previous log file, while a Command-Log-On command with a file name will close the old log file and start using a new file with the specified name.

Initially the command log facility is turned off. It can be turned off explicitly by using the command Command-Log-Off.

The generated log file is directly possible to use as a file in the command Include-File. It will, however, contain exactly the commands given in the session, even those that were not executed due to command errors.

The concluding Command-Log-Off command will also be part of the log file.

Connect-To-Editor

Parameters:

```
(None)
```

This command will create a new *Breakpoints* menu in the SDL Editor and tell the SDL Editor to show all graphical breakpoints. If no SDL Editor is opened, the menu will appear when an SDL Editor is opened the next time. The commands given in the new menu will only be handled in those simulations that are connected. See also the Disconnect-Editor command.

Create

Parameters:

```
<Process name> <Parent's PId value>
<Process parameters>
```

A process instance of the specified process type is created. If the parent PId is not equal to null, Offspring is set in the specified parent instance. If the number of instances of the specified process type is greater than or equal to the maximum number of concurrent instances, the user has to verify the create action.

Define-Continue-Mode

Parameters:

```
"On" | "Off"
```

Defines whether the execution of the simulation shall continue after a command is given. The default is "Off".

Define-Integer-Output-Mode

Parameters:

```
"dec" | "hex" | "oct"
```

Defines whether integer values are printed in decimal, hexadecimal or octal format. In hexadecimal format the output is preceded with "0x", in octal format the output is preceded with '0' (a zero).

On input: if the format is set to hexadecimal or octal, the string determines the base as follows: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" hexadecimal conversion. Otherwise, decimal conversion is used.

The default is “dec”, and no input conversion is performed.

Define-MSC-Trace-Channels

Parameters:

```
"On" | "Off"
```

Defines whether the env instance should be split into one instance for each channel connected to env in the MSC trace. The default is “Off”.

Detailed-Exa-Var

Parameters:

```
(None)
```

When printing structs containing components with default value, these values are explicitly printed after this command is given.

Disconnect-Editor

Parameters:

```
(None)
```

This command will remove the connection to the SDL Editor. If this simulation is the only connected to the SDL Editor, it will also remove the *Breakpoints* menu in the SDL Editor. See also the Connect-To-Editor command.

Display-Array-With-Index

Parameters:

```
"On" | "Off"
```

When Display-Array-With-Index is On and you examine an array, the value of the array element is printed with its index. Index is added before the value of the array element. The default is “Off”.

Down

Parameters:

```
<Optional service name>
```

Moves the scope one step down in the procedure call stack. If the current scope is a process containing services, one of the services should be specified. See also the commands Stack, Set-Scope and Up.

Examine-PId

Parameters:

```
[ '(' <PId value> ')' ]
```

Information about the process instance given by the current scope is printed (see the Set-Scope command for an explanation of scope). This information contains the current values of Parent, Offspring, Sender and a list of all currently active procedure calls made by the process instance (the stack). The list starts with the latest procedure call and ends with the process instance itself. If the process contains services, these services will be listed, each with its own procedure call stack.

If a PId is given within parenthesis information about this process instance is printed instead.

Examine-Signal-Instance

Parameters:

```
<Entry number>
```

The parameters of the signal instance at the position equal to entry number in the input port of the process instance given by the current scope are printed (see the Set-Scope command for an explanation of scope). The entry number is the number associated with the signal instance when the command List-Input-Port is used.

Examine-Timer-Instance

Parameters:

```
<Entry number>
```

The parameters of the specified timer instance are printed. The entry number is the number associated with the timer when the List-Timer command is used.

Examine-Variable

Parameters:

```
[ '(' <PId value> ')' ]
<Optional variable name>
<Optional component selection>
```

The value of the specified variable or formal parameter in the current scope is printed (see the Set-Scope command for an explanation of scope). Variable names may be abbreviated. If no variable name is given, all variable and formal parameter values of the process instance given by the current scope are printed. Sender, Offspring, Self, and Parent may also be examined in this way. Their names, however, may not be abbreviated and they are not included in the list of all variables.

Note:

If a variable is exported, both its current value and its exported value are printed.

It is possible to examine only a component of a struct, string or array variable, by appending the struct component name or a valid array index value as an additional parameter. The component selection can handle structs and arrays within structs and arrays to any depth by giving a list component selection parameters. SDL syntax with '!' and "( )" as well as just spaces, can be used to separate the names and the index values.

It is also possible to print a range of an array by giving "FromIndex: To-Index" after an array name. Note that the space before the ':' is required if FromIndex is a name (enumeration literal), and that no further component selection is possible after a range specification.

To see the possible components that are available in the variable, the variable name must be appended by a space and a '?' on input. A list of components or a type name is then given, after which the input can be continued. After a component name, it is possible to append a '?' again to list possible sub components.

To print the value of the data referenced by a Ref pointer it is possible to use the SDL syntax, i.e. the "*>" notation. If, for example, `Iref` is a Ref(Integer) variable then `Iref*>` is the integer referenced by this pointer. If `Sref` is a Ref of a struct, then `Sref*> ! Comp1` is the Comp1 component in the referenced struct. The sequence `*> !` can in the monitor be replaced by `->` (as for example in C).

If a PId is given within parenthesis the scope is temporarily changed to this process instance instead.

Exit

Parameters:

`(None)`

The simulation is terminated. If the command is abbreviated the monitor asks for confirmation. This is the same command as Quit.

Finish

Parameters:

`(None)`

This command will execute the currently executing procedure up to and including its return, i.e. it will finish this procedure. The execution will also be interrupted at the end of the transition. In a process, Finish will behave just like the command Next-Transition.

Go

Parameters:

(None)

The execution of the simulation is resumed. The execution will continue until one of the conditions listed in "Activating the Monitor" on page 2063 becomes true. To stop the execution of transitions, press <Return> (and only this key). The interactive monitor will then become active when the execution of the current SDL symbol is completed.

This command has to be used with care in the interactive monitor, as it might result in executing the simulation program "forever" (if none of the conditions for activating the monitor becomes true). In a catastrophic situation (e.g. an endless loop within an SDL symbol) it is possible to terminate the program by using the way to stop the execution of a program defined in the operating system (<Ctrl+C> **on UNIX**).

Go-Forever

Parameters:

(None)

The command Go-Forever behaves, in most situations, in the same way as the command Go.

For Go-Forever to behave differently than Go, there has to be an environment to the SDL system that can send signals to the SDL system (communicating simulations or environment functions). When the command Go is issued and the SDL system becomes completely idle (no possible transition and no active timer) the monitor becomes active again and is entered. If Go-Forever was used instead of Go, the monitor is not entered in this case; instead the simulation continues to wait for external stimulus. This feature is valuable when a simulation communicates with other simulations or applications.

Help

Parameters:

<Optional command name>

Issuing the Help command without a parameter will print all the available commands. If a command name is given as parameter, this command will be explained.

Include-File

Parameters:

```
<File name>
```

This command provides the possibility to execute a sequence of monitor commands stored in a text file. The Include-File facility can be useful for including, for example, an initialization sequence or a complete test case. It is allowed to use Include-File in an included sequence of commands; up to five nested levels of include files can be handled.

This command also provides the possibility to perform script calling with variable substitution.

In a script file it is possible to refer to variables using $1, $2, ... , $99. The numbered variables refer to the arguments (actual values) given when calling a script using the command Include-File.

Unix shell script expressions like $0, $#, $? and $* are not available to be given as variable references in script files. Letters are not used as reference variables. For example $a, $1b, $ac are not available to be used as reference variables.

The actual values will be given as parameters in the Include-File command after script-file name. It is possible to pass up to 99 actual values. The numbered variables will be substituted by the actual values during execution of commands in the script files.

Arguments should be separated by spaces. $1 will be substituted by arg1, $2 will be substituted by arg2, $3 will be substituted by arg3, and $4 will be substituted by arg4 and so on.

If there is an argument missing, the corresponding reference variable will not get a value, it is up to the user to provide the correct arguments when using the Include-File command.

List-Breakpoints

Parameters:

```
(None)
```

All active breakpoints are listed. Each breakpoint is assigned an entry number that can be used in the Remove-All-Breakpoints and Show-Breakpoint commands.

List-GR-Trace-Values

Parameters:

```
(None)
```

The values of all currently defined GR traces are listed. The list contains the trace unit type (system, block, process, PId), the unit name, and the GR trace value (both numeric and a textual explanation). See also the commands Set-GR-Trace and Reset-GR-Trace.

List-Input-Port

Parameters:

```
[ '(' <PId value> ')' ]
```

A list of all signal instances in the input port of the process instance given by the current scope is printed (see the Set-Scope command for an explanation of scope). For each signal instance an entry number, the signal type, and the sending process instance is given. A '*' before the entry number indicates that the corresponding signal instance is the signal instance that will be consumed in the next transition performed by the process instance. The entry number can be used in the commands Examine-Signal-Instance, Rearrange-Input-Port and Remove-Signal-Instance.

If a PId is given within parenthesis information about this process instance is printed instead.

List-MSC-Log

Parameters:

```
(None)
```

This command returns the current status of the MSC log (off / interactive / batch). See also the commands Start-Interactive-MSC-Log, Start-Batch-MSC-Log and Stop-MSC-Log.

List-MSC-Trace-Values

Parameters:

```
(None)
```

This command returns the current status for the MSC trace parameters. The list contains the trace unit type (system, block, process, PId), the

unit name, and the MSC trace value (both numeric and a textual explanation). See also the commands Set-MSC-Trace and Reset-MSC-Trace.

List-Process

Parameters:

```
<Optional process name>
```

A list of all process instances associated with the specified process type is produced. If no process name is specified all process instances in the system are listed. The list will contain the same details as described for the List-Ready-Queue command.

List-Ready-Queue

Parameters:

```
(None)
```

A list is produced containing the process instances in the ready queue, i.e., those instances that have received a signal that can cause an immediate transition, but that have not yet had the opportunity to execute this transition to its end. If the process contains services, the currently active service is also listed. If a process/service instance has active procedure calls, the current executing procedure instance is also listed. For each instance in the list the following information is given:

1. An entry number (process instances only).
2. An identification of the process/service/procedure instance.
3. The name of its current state. If the state name is followed by a '*', then the process/service/procedure is currently executing a transition starting from this state.
4. The number of signal instances in the input port of the process instance.
5. The signal name of the signal instance that will cause (or has caused) the transition by the process instance. This signal instance is marked with a '*' before the entry number, if the command List-Input-Port is issued. If the transition has started the signal is no longer in the input port.

The process instance, the state, and the signal instance determine uniquely the transition that will be executed by the process instance when it gets permission to do so.

The entry number can be used in the command Rearrange-Ready-Queue.

List-Signal-Log

Parameters:

```
(None)
```

Print information about the currently active signal log; see the command Signal-Log for more details. Each log is assigned an entry number which can be used in subsequent Close-Signal-Log commands.

List-Timer

Parameters:

```
(None)
```

A list of all currently active timers is produced. For each timer, its corresponding process instance and associated time is given. An entry number will also be part of the list, which can be used in the command Examine-Timer-Instance.

List-Trace-Values

Parameters:

```
(None)
```

The values of all currently defined traces are listed. The list contains the trace unit type (system, block, process, PId), the unit name and the trace value (both numeric and a textual explanation). See also the commands Set-Trace and Reset-Trace.

Log-Off

Parameters:

```
(None)
```

The command Log-Off turns off the interaction log facility, which is described in the command Log-On.

Log-On

Parameters:

```
<Optional file name>
```

The command Log-On takes an optional file name as a parameter and enables logging of all the interaction between a simulation program and the user that is visible on the screen. The first time the command is entered, a file name for the log file has to be given as parameter. After that any further Log-On commands, without a file name, will append more

information to the previous log file, while a Log-On with a file name will close the old log file and start using a new file with the specified file name.

Initially the interaction log facility is turned off. It can be turned off explicitly by using the command Log-Off.

News

Parameters:

```
(None)
```

The News command summarizes the changes in the textual monitor user interface from previous releases.

Next-Statement

Parameters:

```
<Optional number of statements>
```

This command is used to step statement for statement through SDL transitions. A statement is the same as a symbol, except that a task symbol may contain several assignment statements; compare with the Next-Symbol command.

Next-Statement steps over procedure calls, i.e. all actions in the procedure will be executed and the monitor will be entered again when the statement after the procedure call is reached; compare with the Step-Statement command.

After making the step(s) the monitor is entered, making it possible to, for example, examine the temporary status of the actual process instance.

> **Note:**
>
> The right hand side of an assignment may contain a value returning procedure call.

Next-Symbol

Parameters:

```
<Optional number of symbols>
```

This command is used to step symbol for symbol through SDL transitions. A symbol may contain several statements; compare with the Next-Statement command.

Next-Symbol steps over procedure calls, i.e. all actions in the procedure will be executed and the monitor will be entered again when the symbol after the procedure call is reached; compare with the Step-Symbol command.

Using the optional integer parameter, a specified number of symbols can be stepped through. Next-Symbol will, however, never step from within one transition into another transition.

After making the step(s) the monitor is entered, making it possible to, for example, examine the temporary status of the actual process instance.

Note:

Join is not considered a symbol by this command.

To determine how far a transition has been executed, the commands Show-Next-Symbol and Show-Previous-Symbol, together with trace printouts and GR traces can be used. The last trace printout is from the last executed symbol (if it has caused a printout, which depends on the trace level), while the symbol next to be executed is selected by the GR trace, if GR trace is on.

Next-Transition

Parameters:

(None)

The next transition is executed. If real time is used and the next transition is a timer output scheduled in the future (more than a second from now), the command Next-Transition will wait one second, after which the monitor is entered again.

Executing a Next-Transition command within a transition will execute the remaining part of the transition.

Next-Visible-Transition

Parameters:

(None)

A number of transitions are executed up to and including the next transition with a trace value > 0. For a timer output transition, it is the trace value for the corresponding process instance that is considered.

This command should be used with care in the interactive monitor, as it might result in executing the simulation program “forever” (if no transition with trace>0 is ever executed). To stop the execution of transitions, press <Return>. The interactive monitor will then become active when the execution of the current SDL symbol is completed.

Nextstate

Parameters:

<State name>

The state of the process/service/procedure instance given by the current scope is changed (see the Set-Scope command for an explanation of scope). This command implies that all actions in a nextstate are performed, i.e. recalculation of enabling conditions and continuous signals, and searching for a signal to be received according to the input and save sets. As a consequence the ready queue for process instances may change.

Now

Parameters:

(None)

The current value of the simulation time is printed.

Output-From-Env

Parameters:

-

The command will list all signals possible to send into the system, together with receiver. If the signal contains parameters, an empty parenthesis pair will be added after the signal name. Only signals that will trigger a transition directly will be listed. So signals causing an immediate null transition or signals that will be saved are not listed.

By just giving Output-From-Env in the Simulator UI, the possible choices will be given in a dialogue and by selecting one of the signals it is sent into the system. If the signal contains parameters they will be given "null" values.

Output-Internal

Parameters:

<Signal name> <Signal parameters>
<Receiver’s PId value> <Sender’s PId value>

The command is used to simulate an output of an internal signal instance between two process instances in the system. The parameter <Sender's PId value> may also denote "env:1". No check is made to ensure that a path of signal routes and channels exists. The monitor responds with either of the three messages below.

```
Signal <name> is not in valid input signal set of
process <name>
```

The signal was not in the valid input signal set of the receiver and can thus not be handled by the receiver. The signal was immediately discarded.

```
Signal <name> was sent to <receiver> from <sender>
```

The signal instance was successfully sent and was placed in the input port of the receiver.

```
Signal <name> caused an immediate null transition
```

The receiving process instance is currently waiting in a state where the specified signal type neither may cause an input operation, nor is saved. The signal instance is immediately discarded (in SDL this is called a null transition, i.e a transition with no actions, leading back to the same state).

Output-None

Parameters:

```
<PId value> <Optional service name>
```

The command is used to send a signal "none" to the specified process instance. If the process instance contains services, one of the services should to be specified as well. The none signal is used to indicate that a spontaneous transition should be initiated in the current state.

Output-To

Parameters:

```
<Signal name> <Signal parameters>
<Receiver's PId value>
```

The command is used to send a signal from the environment of the system to a process instance in the system. It will be checked that a path of channels and signal routes exists from the environment to the specified process instance.

The monitor will respond with either of the messages described below, or by an error message if no path existed.

```
Signal <name> was sent to <receiver> from <sender>
```

The signal instance was successfully sent and is placed into the input port of the receiver.

```
Signal <name> caused an immediate null transition.
```

The receiving process instance is currently waiting in a state where the specified signal type neither may cause an input operation, nor is saved. The signal instance is immediately discarded (in SDL this is called a null transition, a transition with no actions, leading back to the same state).

Output-Via

Parameters:

```
<Signal name> <Signal parameters>
<Optional channel name>
```

The command is used to send a signal from the environment of the system to a process instance in the system via the specified channel. If no channel is given, it is assumed that any channel from the environment to the system may be selected. It will be checked that there exists one process instance that can receive the signal, according to the rules of outputs in SDL.

The monitor will respond with either of the messages below, or by an error message if the number of possible receivers is not equal to one.

```
Signal <name> was sent to <receiver> from <sender>
```

The signal instance was successfully sent and is placed into the input port of the receiver.

```
Signal <name> caused an immediate null transition.
```

The receiving process instance is currently waiting in a state where the specified signal type neither may cause an input operation, nor is saved. The signal instance is immediately discarded (in SDL this is called a null transition, a transition with no actions, leading back to the same state).

Print-Coverage-Table

Parameters:

```
<File name>
```

This command can be used to obtain test coverage information and profiling information. Each time the command is issued a coverage file, with the name specified as parameter, is produced in the work directory

containing the relevant information. The coverage file always reflects the situation from the start of the simulation. To study the information in the coverage file, it can be opened in the Coverage Viewer; see chapter 48, *The SDL Coverage Viewer*.

> **Note:**
>
> The specified file is always overwritten, i.e. there is no confirmation message if an existing file is specified.

The generated file consists of two sections, first a summary with profiling information, containing the number of transitions and the number of symbols executed by each process type. Secondly the file contains detailed information about how many times each symbol and each state - input combination is executed.

Example 315: Profiling Information in Coverage File ––––––––––––––

```
*************** PROFILING INFORMATION **************

2  System DemonGame : Transitions = 13, Symbols = 40
3  Block GameBlock
4  Process Main : Transitions = 3 (23%),
                  Symbols = 10 (25%), MaxQ = 2
4  Process Game : Transitions = 6 (46%),
                  Symbols = 15 (37%), MaxQ = 2
3  Block DemonBlock
4  Process Demon : Transitions = 4 (30%),
                  Symbols = 15 (37%), MaxQ = 1
```

––

This information in Example 315 should be interpreted in the following way:

- As *transitions*, the number of executed input symbols + continuous signal symbols + start symbols are counted.
- As *symbols*, the number of executed process symbols (task, output, decision, set, reset, call, export, stop, return, create, nextstate, input, continuous signal, start) are counted.
- The number of transitions and the number of executed symbols are, as shown in the example above, presented both for the system (total for all processes) and for each process type. The relative numbers for the processes are, of course, relative to the numbers for the system.

- *MaxQ* is the maximum input port queue length for any instance of the process type. Note that start up signals, used to implement create, and continuous signals are counted as signals in MaxQ.
- The numbers at the beginning of each line are the scope level of the that unit. This can be used to determine if, for example, a procedure is defined within or after another procedure.

> **Note:**
>
> To be true, profiling information **execution time** ought to be measured instead of the number of transitions and number of executed symbols, but this information is still very valuable for getting a feeling for the load distribution.

Example 316: Coverage Table in Coverage File ————————————

```
************** COVERAGE TABLE DETAILS **************

2 System DemonGame

3 Block GameBlock

4 Process Main
  1 : Start  #SDTREF(SDL,main.spr(1),131(30,10),1)
  1 : Game_Off  Newgame  #SDTREF(SDL,main.spr(1),137
  1 : Game_On  Endgame  #SDTREF(SDL,main.spr(1),119(
  --------------------
  1 START : #SDTREF(SDL,main.spr(1),131(30,10),1)
  1 INPUT : #SDTREF(SDL,main.spr(1),137(31,44),1)
  1 INPUT : #SDTREF(SDL,main.spr(1),119(56,44),1)
  1 NEXTSTATE : #SDTREF(SDL,main.spr(1),134(34,29),1)
  1 CREATE : #SDTREF(SDL,main.spr(1),140(37,59),1)
  1 TASK : #SDTREF(SDL,main.spr(1),143(34,72),1)
  1 NEXTSTATE : #SDTREF(SDL,main.spr(1),146(35,89),1)
  1 OUTPUT : #SDTREF(SDL,main.spr(1),122(56,59),1)
  1 TASK : #SDTREF(SDL,main.spr(1),125(60,72),1)
  1 NEXTSTATE : #SDTREF(SDL,main.spr(1),128(59,89),1)
```

————————————————————————————————————

For each process type two types of information are given, separated by a line of hyphens:

- The first part contains information about the start transition and each relevant state-input combination: the number of times executed and a GR or PR reference to the associated process symbol. Continuous signals are counted as a separate input.

- The second part contains information about each process symbol: the number of times it has been executed, the type of symbol, and a GR or PR Reference to the symbol.

To identify the process symbols, the command *Show GR Reference* command in the SDL Editor's *Tools* menu can be used for the printed GR references. (See "Show GR Reference" on page 1961 in chapter 44, *Using the SDL Editor*.)

Proceed-To-Timer

Parameters:

```
(None)
```

This command will execute all transitions up to but not including the next timer output. The timer output will not be executed even if it is the next transition.

Proceed-Until

Parameters:

```
<Time value>
```

The execution of the simulation is resumed. The monitor will become active when the value of the simulation time first becomes equal to the time value given as parameter.

Note that relative time values can be given using the '+' sign. Entering "+5.0" as parameter is interpreted as the time value NOW+5.0. See also "Input and Output of Data Types" on page 2069.

Quit

Parameters:

```
(None)
```

The simulation is terminated. If the command is abbreviated the monitor asks for confirmation. This is the same command as Exit.

Rearrange-Input-Port

Parameters:

```
<Entry number> <New entry number>
```

This command is used to change the order of signal instances in the input port of the process instance given by the current scope (see the Set-Scope command for an explanation of scope).

The entry number parameters refer to the numbers assigned by the command List-Input-Port. The signal instance with entry number equal to the first parameter will be moved to a position where it in a subsequent List-Input-Port command will be assigned the entry number given as the second parameter.

Rearrange-Ready-Queue

Parameters:

```
<Entry number> <New entry number>
```

This command is used to change the order in which transitions by process instances are executed. The entry number parameters refer to the entry numbers assigned by the command List-Ready-Queue. The process instance with entry number equal to the first parameter will be moved to a position where it in a subsequent List-Ready-Queue command will be assigned the entry number given as the second parameter.

The Rearrange-Ready-Queue command should be used with care when process priorities are used. The ready queue is then sorted in priority order (see also "Scheduling" on page 2577 in chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*), which means that a rearrangement might disturb this order. Such a disturbance does not harm, except when a new process should be inserted into the ready queue. In this situation the insert point might not be the proper one. The following insert algorithm is used: Search from the end of the ready queue until a process with higher or equal priority (lower or equal priority value) is found, then insert the new process after the found process. If no process with higher or equal priority is found, then insert the new process first in the ready queue.

REF-Address-Notation

Parameters:

```
(None)
```

REF values (pointers introduced using the generator Ref) are printed as addresses, using the HEX value for the address. The Null value is printed as `Null`. On input, both this syntax and the Value-Notation (see command REF-Value-Notation) can be used.

The same applies for Own and ORef values.

REF-Value-Notation

Parameters:

(None)

REF values (pointers introduced using the generator Ref) are printed as NEW(<the value the pointer refers to>). This is the default syntax for REF values. It means that complete lists or graphs will be printed. Example:

```
NEW( (. 1, NEW( (. 2, Null .) ) .) )
```

To avoid problems in cyclic graphs a special syntax is used if a pointer refers to an address already presented in the output. OLD n, where n is a digit, means a reference to the n:th NEW in the printed value. Example:

```
NEW( (. 1, NEW( (. 2, OLD 1 .) ) .) )
```

The Null value is printed as Null. On input both this syntax and the Address-Notation (see command REF-Address-Notation) can be used.

The same applies for Own and ORef values.

Remove-All-Breakpoints

Parameters:

(None)

This command removes all breakpoints.

Remove-At

Parameters:

<SDT reference>

This command removes all breakpoints at the symbol specified by the SDT reference.

Remove-Breakpoint

Parameters:

<Entry number>

This command removes the breakpoint with the specified entry number. The entry number given by List-Breakpoints should be used.

Remove-Signal-Instance

Parameters:

<Entry number>

The signal instance with the given entry number in the input port of the process instance given by the current scope is removed (see the Set-Scope command for an explanation of scope). The entry number given by List-Input-Port should be used. If this signal was selected for the next transition, the process instance will execute an implicit next-state action.

> **Note:**
>
> Entry numbers are just positions in the input port. The removal of a signal changes the entry numbers of the remaining signals.

Reset-GR-Trace

Parameters:

```
<Optional unit name>
```

The GR trace value of the given unit is reset to undefined. If no unit is specified the GR trace value of the system is reset to undefined. As there always has to be a GR trace value defined for the system, Reset-GR-Trace on the system is considered to be equal to setting the GR trace value to 0. For more information about optional unit names, see the command Set-Trace.

Reset-MSC-Trace

Parameters:

```
<Optional unit name>
```

The MSC trace value of the given unit is reset to undefined. If no unit is specified the MSC trace value of the system is reset to undefined. As there always has to be a MSC trace value defined for the system, Reset-MSC-Trace on the system is considered to be equal to setting the trace value to 0. For more information about optional unit names, see the command Set-MSC-Trace.

Reset-Timer

Parameters:

```
<Timer name> <Timer parameters>
```

The result of the command is exactly the same as if the process instance given by the current scope had executed a reset action. If the reset action causes a timer signal to be removed and this signal was selected for the next transition, the process instance will execute an implicit nextstate action.

Reset-Trace

Parameters:

```
<Optional unit name>
```

The trace value of the given unit is reset to undefined. If no unit is specified the trace value of the system is reset to undefined. As there always has to be a trace value defined for the system, Reset-Trace on the system is considered to be equal to setting the trace value to 0. For more information about optional unit names, see the command Set-Trace.

Restore-State

Parameters:

```
<File name>
```

This command will restore the state of the current simulation to a state given in a file created with the Save-State command. The file name is given as parameter.

The write and read functions for the sorts are used to save and restore the variable values. So if a user has added a sort via the ADT directive, these write and read functions must be consistent.

It is recommended to restart the simulator before giving the Restore-State command. The command is not allowed when the execution is within a transition.

If the SDL system contains the Any construct, the sequence of random numbers will not be the same after the restore.

Only the sdl execution state is restored, no monitor settings are restored. E.g. Show-Previous-Symbol, Show-Next-Symbol does not work directly after a restore. The MSC trace is not restored. Coverage is not restored either.

The ADT package pidlist.pr is not supported, since the PId synonyms that are initialized are not valid after the restore.

Extern C and C++ variables will not be saved and restored since the simulator has no knowledge of them.

Charstar, Voidstar and Voidstarstar variables are not handled.

Since all variables are saved, dangling Ref and ORef variables will cause problems.

Scope

Parameters:

```
(None)
```

This command prints the current scope. See the command Set-Scope for a description of scope.

SDL-Value-Notation

Parameters:

```
(None)
```

The value notation used in all outputs of values is set to SDL value notation. This is the default value notation. See also the command ASN1-Value-Notation.

Save-Breakpoints

Parameters:

```
<File name>
```

This command will save all breakpoints in a file, with the name specified as parameter (in the work directory). It is written as a text file with commands, so the breakpoints can be restored with the Include-File command.

Save-State

Parameters:

```
<File name>
```

This command will save the state of the current simulation in a file, with the name specified as parameter (in the work directory). It is written as a text file with special commands, so the state can be restored with the Restore-State command.

Set-GR-Trace

Parameters:

```
<Optional unit name> <Trace value>
```

The GR trace value is assigned to the specified unit (system, block, process, or process instance). If no unit is specified the GR trace value is assigned to the system. The initial GR trace value of the system is 0, i.e. no GR trace, while it is undefined for all other units. For more information about optional unit names, see the command Set-Trace.

For a description of the possible GR trace values, see "GR Traces" on page 2118.

Set-MSC-Trace

Parameters:

```
<Optional unit name> <Trace value>
```

This command enables the trace of SDL events that take place during the simulation and which can be transformed into events in a Message Sequence Chart. Typically, the events that can be transformed are sending and consumption of signals, and creation and termination of processes.

> **Note:**
>
> Setting the MSC trace value with this command **does not** start the actual logging of MSC events. To do this, the command Start-Interactive-MSC-Log or Start-Batch-MSC-Log is used.

The scope of the trace can be delimited by specifying an optional unit name and a trace value. The general considerations for specifying the unit name for the command Set-Trace are also applicable for Set-MSC-Trace.

Optional Unit Name

- The trace value is assigned to the specified unit (system, block, process, or process instance).
- When specifying a unit name, the scope of trace affects **all process instances** contained in the **underlying SDL structure**.
- Initially, the scope of trace is set on the entire SDL system. So, by default, all events will generate Message Sequence Chart traces. If the unit name is omitted, the trace value is assigned to the system.

Trace Value

- Specifying a trace value of 0 means no trace for the currently specified unit name.
- Specifying a trace value of 1 means trace only if both involved units have a trace value greater than 0.
- Specifying a trace value of 2 means always trace, independent of the trace value of the other involved unit.
- Specifying a trace value of 3 means trace on the block level.

The initial MSC trace value of the system is 1. For more information on MSC trace values, see "Message Sequence Chart Traces" on page 2119.

> **Note:**
>
> Once the logging of Message Sequence Chart traces has been started (using any of the commands Start-Interactive-MSC-Log or Start-Batch-MSC-Log) **modifying the scope of trace may cause unpredictable results**, such as Message Sequence Charts with an unexpected appearance.

Set-Scope

Parameters:

```
<PId value> <Optional service name>
```

This command sets the scope to the specified process, at the bottom procedure call. If the process contains services, one of the services can be given as parameter to the command. A scope is a reference to a process instance, a reference to a service instance if the process contains services, and possibly a reference to a procedure instance called from this process/service. The scope is used for a number of other commands for examining and changing the local properties of a process instance. The scope is automatically set by the execution commands, when entering the monitor, to the next process in turn to execute.

The command Scope prints out the current scope; i.e. the name of the process instance and possibly the service instance and the called procedure instance. See also the commands Stack, Define-MSC-Trace-Channels and Up.

Set-Timer

Parameters:

```
<Timer name> <Timer parameters> <Time value>
```

The result of the command is exactly the same as if the process instance given by the current scope had executed a set action. If the set action causes a timer signal to be removed and this signal was selected for the next transition, the process instance will execute an implicit nextstate action.

Set-Trace

Parameters:

```
<Optional unit name> <Trace value>
```

The trace value is assigned to the specified unit (system, block, process, or process instance). If no unit is specified the trace value is assigned to the system. The initial trace value for the system is 4, while it is undefined for all other units. For a description of the possible trace values, see "Trace Limit Table" on page 2116.

There might, in some cases, be problems in identifying a specific unit. If more than one unit in the system match the, possibly abbreviated, unit name, the first unit found when searched from the system level will be assigned. To make sure the correct unit is assigned, the unit's diagram type, e.g. "process" (unabbreviated), can be introduced before the unit name. If there still are problems, for instance due to the fact that there are several processes with the same name, a qualifier should be inserted immediately before the unit name. An example can be found in "Specifying Unit Names" on page 2186 in chapter 51, *Simulating a System*.

To specify a PId value, a colon and an instance number must follow directly after the process name; see also "Input and Output of Data Types" on page 2069.

Show-Breakpoint

Parameters:

```
<Entry number>
```

This command is only applicable for symbol breakpoints defined with the command Breakpoint-At. The breakpoint with the specified entry number is listed, and the symbol with the breakpoint will be selected in an SDL Editor showing the source GR document. The entry number given by List-Breakpoints should be used.

Show-C-Line-Number

Parameters:

```
(None)
```

This command prints the `.c` file name and line number where the execution of the current SDL transition is suspended. The Text Editor is opened with the cursor positioned on that line in the C source file. The given line number will reference a statement with the following structure:

```
XBETWEEN_SYMBOLS(...)
```

The case label just following this statement is the place where the execution will be resumed when an execute command is given in the monitor.

The Show-C-Line-Number command is mainly thought to be used when the execution is interrupted within an SDL transition. If the command is issued between two transitions, the `XBETWEEN_SYMBOLS` statement immediately before the last Nextstate or Stop operation will be referenced in the printout, together with a warning that this is not the current position.

> **Note:**
>
> Changes made in the monitor, as for example Rearrange-Ready-Queue, will not affect the printout by this command.

Show-Coverage-Viewer

Parameters:

`(None)`

This command starts the Coverage Viewer tool with the current test coverage loaded.

Show-Next-Symbol

Parameters:

`(None)`

The symbol in turn to be executed will be selected in an SDL Editor showing the source GR document. If the simulation is not connected to the SDL suite or the SDL source document is in SDL/PR, a GR symbol reference or a PR reference will instead be displayed on the screen. The details about GR symbol references are presented in "Dynamic Errors" on page 2121. A PR reference is a file name and a line number. See also "Syntax" on page 911 in chapter 19, *SDT References*.

Note that between the execution of two transitions, i.e. after a Nextstate or Stop operation, no next symbol can be shown. Note also that changes made in the monitor, as for example changing the ready queue using the command Rearrange-Ready-Queue, will not affect the symbol selected by Show-Next-Symbol.

This command uses the same mechanisms to select symbols in the SDL Editor as the GR trace facility, so the same general characteristics as presented in "GR Traces" on page 2118 are also valid for this command.

Show-Previous-Symbol

Parameters:

```
(None)
```

The last executed symbol will be selected in an SDL Editor displaying the source GR document. If the simulation is not connected to the SDL suite or the SDL source document is in SDL/PR, then a GR symbol reference or a PR reference will instead be displayed on the screen. See the command Show-Next-Symbol for more information.

Show-Versions

Parameters:

```
(None)
```

The versions of the SDL to C Compiler and the runtime kernel that generated the currently executing program are presented.

Signal-Log

Parameters:

```
<Unit name> <File name>
```

Starts logging of signals to a specified file. See also the commands Close-Signal-Log and List-Signal-Log.

The unit name parameter should be either a channel, a signal route, a system, a block, a process type, or a process instance.

- If the unit is a channel or a signal route then all signals sent through the channel or signal route will be logged on the specified file.
- If the unit is a process type or a process instance then all signals sent to or from any instance of the process type or to or from the specified process instance will be logged.
- If the unit is a block or a system then all signals that have relevance for the block or system will be logged, i.e. signals sent within the block or system or signals sent to or from the block or system.

Note that the process “env” is a legal unit in the Signal-Log command. By specifying “env” as unit parameter, the signal interface between the system and the environment is logged. For more information about the unit name parameter, see the command Set-Trace.

Stack

Parameters:

`(None)`

The procedure call stack for the PId/service defined by the scope is printed. For each entry in the stack, the type of instance (procedure/process/service), the instance name and the current state is printed. See also the commands Set-Scope, Define-MSC-Trace-Channels and Up.

Start-Batch-MSC-Log

Parameters:

`<Symbol level> <File name>`

This command starts the logging of SDL events which can be translated into the corresponding MSC events in a log file (see "Mapping Between SDL and MSC" on page 2119). See also the commands Start-Interactive-MSC-Log and Stop-MSC-Log.

The results will be stored in a log file, whose contents are specific for this purpose. That log file can later on be read by a Message Sequence Chart Editor, where its contents are interpreted as a Message Sequence Chart and displayed as such.

The symbol level parameter determines if states and actions should be included in the MSC log. For a description of the possible symbol level values, see "Level of Symbol Logging" on page 2120.

The file name parameter to this command can be any valid file name, although it is recommended to use a file name with the suffix `.mpr`, since this is the default suffix used when reading a log file into a Message Sequence Chart Editor.

Start-Env

Parameters:

`(None)`

When the SDL to C Compiler is used to generate applications, interface functions towards the environment of the SDL system must be provided. The `InEnv` function, which is used to enter signals into the SDL system, is frequently called from the main loop in the process scheduler. During debugging, the polling of the `InEnv` function can turned on and turned off.

The Start-Env command turns on the polling of `InEnv`. At start up of the program polling is turned off. See also the commands Stop-Env and Call-Env.

> **Note:**
>
> This command and the commands Stop-Env and Call-Env are only available when the SDL to C Compiler is used to generate applications.

Start-Interactive-MSC-Log

Parameters:

```
<Symbol level>
```

This command starts the logging of SDL events which can be translated into the corresponding MSC events in a Message Sequence Chart Editor (see "Mapping Between SDL and MSC" on page 2119). See also the commands Start-Batch-MSC-Log and Stop-MSC-Log.

The symbol level parameter determines if states and actions should be included in the MSC log. For a description of the possible symbol level values, see "Level of Symbol Logging" on page 2120.

When the command is issued, the following takes place:

1. An instance of the Message Sequence Chart Editor is started. Each currently existing (SDL) process instance is displayed as an instance, with its instance head and its instance axis.

2. Following the execution of the simulation, each SDL event which is possible to map to an MSC event and which is within the scope of MSC trace which is currently defined will automatically be displayed in the MSC Editor. Each event will cause the insertion point in the MSC Editor to be moved downwards with one step, which provides a feeling of absolute order between events.

Some drawing conventions and default layouting algorithms are used when drawing the automatically generated Message Sequence Chart. These conventions are described in *"Drawing Conventions" on page 1683 in chapter 40, Using Diagram Editors*.

Start-SDL-Env

Parameters:

```
(None)
```

A simulator can communicate with other simulations using SDL signals and the SDL suite communication mechanism. Signals sent to the environment in one simulation can enter as signals from the environment in another simulator.

This command is used to tell a simulator to start sending signals that are designated to the environment via the SDL suite communication mechanism and to start looking for incoming signals (polling) from the SDL suite communication mechanism. See also the commands Stop-SDL-Env and Call-SDL-Env.

This facility is at program start up turned off, and should only be turned on when a simulator should be able to communicate with other simulators (or applications).

Note:

The command must be given in **both communicating simulators**.

Start-ITEX-Com

Parameters:

`(None)`

This command starts the communication with the TTCN suite.

Start-SimUI

Parameters:

`(None)`

This command starts a Simulator UI and connects the running simulator to it. After this it is only possible to give commands in the UI.

Start-UI

Parameters:

`(None)`

This command attempts to start the program `sdtenv` in the start directory. The started program is assumed to connect itself to the SDL suite communication mechanism and can then communicate with the simulation program.

Note:

The command Start-SDL-Env should be given in the simulation program to make it communicate.

Step-Statement

Parameters:

```
<Optional number of statements>
```

This command is used to step statement for statement through SDL transitions. A statement is the same as a symbol, except that a task symbol may contain several assignment statements; compare with the command Step-Symbol.

Step-Statement will step into procedure calls; compare with the Next-Statement command.

After making the step(s) the monitor is entered, making it possible to, for example, examine the temporary status of the actual process instance.

> **Note:**
>
> The right hand side of an assignment may contain a value returning procedure call.

Step-Symbol

Parameters:

```
<Optional number of symbols>
```

This command is used to step symbol for symbol through SDL transitions. A symbol may contain several statements; compare with the Step-Statement command.

Step-Symbol will step into procedure calls; compare with the Next-Symbol command.

Using the optional parameter, a specified number of symbols can be stepped through. Step-Symbol will, however, never step from within one transition into another transition.

After making the step(s) the monitor is entered, making it possible to, for example, examine the temporary status of the actual process instance.

> **Note:**
>
> Join is not considered a symbol by this command.

Stop

Parameters:

```
<PId value> <Optional service name>
```

The specified process instance is stopped. If the process contains services, then either the process can be stopped by giving no service parameter, or one of the services can be stopped. The result of the command is exactly the same as if the process instance or service instance had executed a stop action.

Stop-Env

Parameters:

```
(None)
```

This command turns off the polling of the InEnv function; see the Start-Env command for more details.

> **Note:**
>
> This command and the commands Start-Env and Call-Env are only available when the SDL to C Compiler is used to generate applications.

Stop-MSC-Log

Parameters:

```
(None)
```

This command stops the logging of Message Sequence Chart events (in interactive mode as well as in batch mode). In the case of a batch mode logging, the log file will be closed. See the commands Start-Interactive-MSC-Log and Start-Batch-MSC-Log for more details.

Following this command, it is possible to log the rest of the session on a new file.

Stop-SDL-Env

Parameters:

```
(None)
```

The command turns off the communication mechanism described for the Start-SDL-Env command.

Up

Parameters:

```
(None)
```

Moves the scope one step up in the procedure call stack. Up from a service leads to the process containing the service. See also the commands Set-Scope, Stack and Define-MSC-Trace-Channels.

Traces

When a process instance executes actions within a transition, trace information describing the current action might be printed on the screen. The amount of information printed can be selected using the trace commands in the monitor system. A typical trace from a transition containing a few actions is given below.

```
*** 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 transitions can also be traced in the GR source diagrams. This is discussed in detail under "GR Traces" on page 2118.

Transition Trace

A trace value, which is a non-negative integer, can be assigned to process instances, to process types, blocks, and to the system. The commands associated with traces are Set-Trace, Reset-Trace and List-Trace-Values.

When a process instance starts a transition, the trace value that governs the amount of trace to be printed is computed according to the following algorithm:

1. If a trace value is defined for the process instance executing the transition, that value is used.
2. If not, and a trace value is defined for the process type, that value is used.
3. Otherwise, if a trace value is defined for the block enclosing the process, that value is used.
4. If still no trace value is found, the block structure is followed outwards until a unit is reached which has a trace value defined. The system will always have a trace value.

The hereby computed value is compared with the trace limit for each action executed during the transition. The trace information is printed only

if the trace value is greater or equal to the trace limit for the action. Trace information at trace level 1 is treated specially, see below.

In the table below the trace limits for all the information that can be part of a trace is presented.

Note:

The **trace messages** are **produced** when **actions** defined in the SDL diagrams are **executed**. These messages are not used when monitor commands like Set-Timer, Nextstate, or Stop are entered.

Trace Limit Table

Action	**Trace limit**
Transition start	2
Parameters of signal in input	6
Output to environment, signal name and receiver	1 (see below)
Output, signal name and receiver	3
Output caused immediate null transition	5
Parameters of signal in output	6
Task	4
Decision, value of expression in decision	4
Procedure start, procedure return	3
Parameters to procedures	6
Create, successful or unsuccessful	3
Parameters to create	6
Set, timer name, time	3
Time less than Now in set, changed to Now	5
Parameters in set	6
Set caused an implicit reset action	5

Action	Trace limit
Reset, timer name	3
Reset caused timer or signal to be removed	5
Parameters to reset	6
Output of timer signal, timer name, receiver, time	2
Output of timer signal caused a null transition	5
Parameters to timer signal	6
Nextstate, state name	3
Null transitions at a nextstate	5
Parameters to signals in null transitions at nextstate	6
Stop	3
Signals discarded at stop	5
Parameters to signals discarded at stop	6
Timers discarded at stop	5
Parameters to timers discarded at stop	6
Export	4

The trace limit table can be summarized as follows:

Trace Limit Table Summary	
0	No trace
1	Trace of signals to environment (environment as seen from the specified unit)
2	Trace of transition start and timer outputs
3	As 2 + trace of important SDL actions
4	As 3 + trace of other SDL actions as well
5	As 4 + result of actions (example: discarded signals)
6	As 5 + Parameters of signals, timers, create actions

Trace information at trace level 1, i.e. trace of signals sent to the environment of the specified unit, is treated in a special way. This information is not printed if the trace value is greater than 1, instead the normal trace of outputs is used.

GR Traces

GR trace is a way to follow the execution of transitions in the GR source diagrams by selecting SDL symbols. It should normally only be used for a small number of processes to limit the amount of information displayed. The GR trace value determines to what degree the execution will be traced, i.e. how often SDL symbols will be selected in the GR diagrams (see below). After a nextstate or stop operation, i.e. between two transitions, the nextstate or stop symbol is still selected.

The commands associated with GR traces are Set-GR-Trace, Reset-GR-Trace and List-GR-Trace-Values.

The GR tracing will take place in a single SDL Editor window, which will show the appropriate GR diagram as the execution progresses. If no SDL Editor is opened, a new editor is started.

Three trace values are possible:

- 0 - No GR trace.
- 1 - When the monitor is entered next, show the next symbol to be executed. No symbols are selected during execution when the monitor is inactive. Thus, the SDL Editor window may not show the diagram containing the symbol currently executed.
- 2 - Follow the execution and show each symbol as it is executed. If the execution is continued into another diagram, this diagram is loaded into the SDL Editor window.

Message Sequence Chart Traces

The commands associated with MSC traces are Set-MSC-Trace, Reset-MSC-Trace and List-MSC-Trace-Values.

Mapping Between SDL and MSC

When executing an SDL system, some of the SDL events can be transformed into a corresponding symbol in a Message Sequence Chart. The mapping rules which govern how SDL events are transformed into MSC symbols, lines and textual elements are described in "Mapping Between SDL and MSC" on page 1682 in chapter 40, *Using Diagram Editors*.

The ITU definition of the MSC language introduces the *instance* concept. An instance is mapped to any instance of an SDL process.

Scope of Trace for Generation of Message Sequence Charts

The scope of MSC trace is the process instances that have a calculated MSC trace value > 0.

Assume that we have set the scope of MSC trace to a part of the SDL system, for instance a block and the underlying structure in terms of processes. When an event takes place in the SDL system, one of three possible situations is possible. Let us assume for the sake of simplicity that the event is the sending of a signal from one process to another process. The three cases and their behavior are as follows:

- First case: The two units (the sending process and the receiving process) are both within the scope of MSC trace. It is easy to transform this into the sending and consuming of a message, and the result will be a trace in a Message Sequence Chart.
- Second case: None of the units is in the scope of MSC trace. This will not result in any trace in a Message Sequence Chart.
- Third case: One and (one only) of the units is in the scope of MSC trace. This case is slightly more complicated. Sending and receiving messages to and from units may be of interest outside the scope of trace.

  The concept *conditional trace* is introduced: A conditional trace will indicate that a message has been sent or received, and that the

receiver or sender is beyond the scope of trace. See also "The Void Instance" on page 1686 in chapter 40, *Using Diagram Editors*.

The MSC trace levels 1 and 2 are used to specify if conditional trace should be presented or not; see the table below. MSC trace level 3 specifies a block level trace.

MSC Trace for Sender	MSC Trace for Receiver	Result
0	0	No trace
	1	No trace
	2	Conditional trace
1	0	No trace
	1	Trace
	2	Trace
2	0	Conditional trace
	1	Trace
	2	Trace

Level of Symbol Logging

The *symbol level* determines the amount of information that should be part of the MSC log. Three symbol levels are possible:

- 0 - Events for signals and timers plus create and stop.
- 1 - As level 0, plus condition symbols for each nextstate.
- 2 - As level 1, plus action symbols for task, decision, call, and return.

Initial Trace Values

At the start of a simulation, the trace values, the GR trace values and the MSC trace values for all units except the system are undefined. The system trace is 4, the system GR trace is 0, and the system MSC trace is 1.

Dynamic Errors

Violations of the dynamic rules of SDL will cause dynamic errors during the execution of a simulation program. A dynamic error is presented to the user on the screen as an error message. See the example below.

Example 317: Dynamic Error Printout ––––––––––––––––––––––––

```
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,/usr/tom/demon.spr(1),122(30,55),1)
TRACE BACK
  Process        : Demon
  Block          : DemonBlock
  System         : Demongame
```

––

The symbol reference given in the `TRANSITION` message should be interpreted as follows:

Item	Text in Example 317	Interpretation
1	`SDL`	Reference to SDL/GR object
2	`/usr/tom/demon.spr`	Reference to the file
3	`(1)`	Page name
4	`122`	The object identity (an unique number assigned by the SDL Editor)
5	`30`	The x-coordinate of the object in mm. The origin of coordinates is the upper left corner of the page.
6	`55`	The y-coordinate of the object in mm
7	`1`	Line number within symbol

By entering the monitor command Show-Previous-Symbol, the symbol that caused the error is displayed.

For more information on references, see chapter 19, *SDT References*.

Dynamic Errors Found by a Simulation Program

```
Error in SDL array index in sort <sort>:
<value> is out of range.
```

Violation of the index range given in an array.

```
Error in assignment in sort <sort>:
<value> is out of range.
```

Violation of the range conditions given in a syntype.

```
Error in SDL Decision: Value is <value>
Entering decision error state
```

The value of the expression in the decision did not match any of the possibilities (answers).

```
Error in SDL Import. Attempt to import from NULL

Error in SDL Import. Attempt to import from stopped
process instance

Error in SDL Import. Attempt to import from the
environment

Error in SDL Import. No process exports this
variable

Error in SDL Import. The specified process does not
export this variable
```

Error in an import statement. Supplementary information about remote variable and exporting processes is also given.

```
Warning in SDL Output of signal <signal>.
No path to receiver, signal discarded
```

An attempt was made to output a signal to a PId expression. There exists, however, no path of channels and signal routes between the sender and the receiver that can convey the signal.

```
Warning in SDL Output of signal <signal>.
No possible receiver found, signal discarded
```

An attempt was made to output a signal without specifying a to PId expression. When all paths or all paths mentioned in a via clause had been examined no possible receiver was found.

```
Warning in SDL Output of signal <signal>.
Signal sent to NULL, signal discarded
```

An attempt was made to output a signal to a PId expression that was null.

```
Warning in SDL Output of signal <signal>.
Signal sent to stopped process instance
```

An attempt was made to output a signal to a PId expression that referred to a process instance which has performed a stop action.

```
Error in SDL View. Attempt to view from NULL

Error in SDL View. Attempt to view from stopped
process instance

Error in SDL View. Attempt to view from the
environment

Error in SDL View: The specified process does not
reveal this variable

Error in SDL View: No process reveals this variable
```

Error in a view statement. Supplementary information about viewed variable and viewing process is also given.

```
Error in SDL Create: Process <process>
More static instances then maximum number of
instances.
```

Obvious!

```
Illegal #UNION tag value for assignment to component
Name.
Tag value is xxx.
```

Attempt to assign a value to a non-active UNION component.

```
Illegal #UNION tag value for access to component
Name.
Tag value is xxx.
```

Attempt to access a non-active UNION component.

```
Error when accessing component Name. Component is
not Present.
```

Attempt to access a non-active optional struct component.

```
Component Name is not active.Present is xxx.
```

Attempt to access a non-active choice component.

```
Dereferencing of NULL pointer.
Pointer assigned new data area at address: HEX(xxx)
```

Attempt to de-reference a Null pointer defined using the Ref generator.

```
User specified error: SDL error expression found
```

Error introduced by the user, by inserting the error expression defined in SDL.

Errors Found in Operators

The errors that can be found in operators defined in the predefined data types are listed below.

```
Error in SDL Operator: Extract! in sort Charstring,
Index out of bounds

Error in SDL Operator: Modify! in sort Charstring,
Character NUL not allowed.

Error in SDL Operator: MkString in sort Charstring,
Character NUL not allowed.

Error in SDL Operator: First in sort Charstring,
Charstring length is 0

Error in SDL Operator: Fix in sort Integer, Integer
overflow

Error in SDL Operator: Last in sort Charstring,
Charstring length is 0

Error in SDL Operator: Substring in sort Charstring,
Charstring length is 0

Error in SDL Operator: Substring in sort Charstring,
Length of substring < 0

Error in SDL Operator: Substring in sort Charstring,
Start index is <= 0

Error in SDL Operator: Substring in sort Charstring,
Start index + length of substring > length of
charstring

Error in SDL Operator: / in sort Integer, Attempt to
divide by 0.

Error in SDL Operator: / in sort Real, Attempt to
divide by 0.0.
```

```
Error in SDL Operator: Rem in sort Integer,
Second operand is 0

Error in SDL Operator: Mod in sort Integer,
Second operand is 0

Error in SDL Operator: Modify! in sort Bit_String,
Index out of bounds.

Error in SDL Operator: Extract! in sort Bit_String,
Index out of bounds.

Error in SDL Operator: First in sort Bit_String,
Bit_String length is zero.

Error in SDL Operator: Last in sort Bit_String,
Bit_String length is zero.

Error in SDL Operator: SubString in sort Bit_String,
Bit_String length is zero.

Error in SDL Operator: SubString in sort Bit_String,
Start is less than zero.

Error in SDL Operator: SubString in sort Bit_String,
SubLength is less than or equal to zero.

Error in SDL Operator: SubString in sort Bit_String,
Start + Substring length is greater than string
length.

Error in SDL Operator: BitStr in sort Bit_String,
Illegal character in Charstring (not 0 or 1).

Error in SDL Operator: HexStr in sort Bit_String,
Illegal character in Charstring (not digit or a-f).

Error in SDL Operator: Modify! in sort Octet,
Index out of bounds.

Error in SDL Operator: Extract! in sort Octet,
Index out of bounds.

Error in SDL Operator: Division in sort Octet,
Octet division with 0.

Error in SDL Operator: Mod operator in sort Octet,
Right operand is 0.

Error in SDL Operator: Rem operator in sort Octet,
Right operand is 0.

Error in SDL Operator: BitStr in sort Octet,
An Octet should consist of not more than 8
characters 0 or 1.
```

```
Error in SDL Operator: HexStr in sort Octet,
An Octet should consist of 2 HEX values.

Error in SDL Operator: HexStr in sort Octet,
Illegal character in Charstring (not digit or a-f).

Error in SDL Operator: Modify! in sort Octet_String,
Index out of bounds.

Error in SDL Operator: Extract! in sort
Octet_String,
Index out of bounds.

Error in SDL Operator: BitStr in sort Octet_String,
An Octet_String should consist of 0 and 1.

Error in SDL Operator: HexStr in sort Octet_String,
Illegal character in Charstring (not digit or a-f).
```

Apart from these error message each instantiation of generator String will introduce similar error messages as for the Charstring sort. Note that `Object_Identifier` is a predefined sort which is an instantiation of String.

Action on Dynamic Errors

After a dynamic error, the execution of the simulation program is continued until the current symbol is ended. The interactive monitor will then become active. The following actions will be taken:

- If the error was an output error the signal will not be sent.
- If the error was a decision error the process instance will immediately be placed in an *decision error state*. It can only be removed from this state by using the Nextstate command in the monitor. The input port will not be affected when the decision error state is entered. All signals sent to a process instance in a decision error state will be saved in the input port.
- If the error occurred during the creation of static process instances, i.e. the initial number of instances is greater that the maximum number of instances, an error message is given and the number of instances specified by initial number of instances are created.
- If the error occurred during an import or view action a data area of the correct size containing zero in all positions is returned.

- If the error was found in a range condition check during an assignment, the variable to the left of the assignment operator will be assigned the computed value, although it is out of bounds.
- If the error was found during a range check of an array index the index value will be changed to be the lowest value of the index type. This means that the corresponding C array will never be indexed out of its bounds.
- If the error occurred during selection of an optional component in a struct or when selecting a component in a #UNION or Choice, an error message is given and the operation is perform anyhow.
- If a Null pointer (Ref) is de-referenced, an new data area of correct size is allocated containing zeros. This data area is assigned to the pointer. After the error message the statement containing the de-referencing is performed.
- If the error occurred within an expression the operator that found the error returns a default value and the evaluation of the expression is continued. The default values returned depend on the result type of the operator and are given in section *"Default Values" on page 2604 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler*.

Assertions

A user can define his own run-time errors or assertions. Basically the run-time library only provides an appropriate C function that can be called to print out an error message. This function is assumed to be used in #CODE directives in TASKs according to the following example:

Example 318: Assertion in C Code ––––––––––––––––––––––––––

```
TASK  '' /*#CODE
#ifdef XASSERT
  if (#(I) < #(K))
    xAssertError("I is less than K");
#endif
*/ ;
```

––

The `xAssertError` function, which has the following prototype:

```
extern void xAssertError ( char *Descr )
```

takes a string describing the assertion as parameter and will produce an SDL run-time error similar to the normal run-time errors. The function is only available if the compilation switch XASSERT is defined. For the standard libraries this is true for all libraries except the *Application Library*.

Run-time Prompting

For some special SDL constructs, an SDL simulator is not able to continue executing without prompting the user for necessary input. The execution continues when the user has entered an allowed value.

Decision with Any

If an SDL DECISION is encountered with an ANY question, the user will be prompted which path to execute:

```
Decision with ANY
1 (go path)
1 ? (show path)
2 (go path)
2 ? (show path)
3 (go path)
3 ? (show path)
Enter path :
```

At this point, the user must enter a path value (integer) within the allowed range. If GR trace is enabled, the user may enter a path value followed by a '?' to see the corresponding path in an SDL Editor.

Informal Decision

If an SDL DECISION is encountered with informal text, the user will be prompted which path to execute:

```
Informal decision: 'Question1'
'answer1'  1
'answer2'  2
'ELSE'  3
Enter path :
```

At this point, the user must enter a path value (integer) within the allowed range. If GR trace is enabled, the user may enter a path value followed by a '?' to see the corresponding path in an SDL Editor.

Unimplemented Operator

If an expression is encountered involving an operator that is not implemented, the user will be prompted for the value to be returned depending on the parameters:

```
Operator op in sort s is called
Parameter 1: 50
Enter value(s) :
```

At this point, the user must enter a value in the resulting sort to be the returned value of the operator when called with the listed parameter values.

Graphical User Interface

This section describes the appearance and functionality of the graphical user interface to the simulator monitor (SimUI). Some user interface descriptions general to all tools in Telelogic Tau can be found in chapter 1, *User Interface and Basic Operations*. These general descriptions are not repeated in this chapter.

Starting the SimUI

A new SimUI is started by selecting *SDL > Simulator UI* from the *Tools* menu in the Organizer. When the SimUI is started, a number of definition files are read, controlling the contents of the main window and some status windows. See "Definition Files" on page 2152 for more information.

No simulator is started automatically by the SimUI in this way. The user must start a simulator by selecting *Open* from the *File* menu, as stated in the text area of the main window, or by using the *Open* quick button.

A simple way to generate a simulator, start the SimUI and open the simulator is to click the *Simulate* quick button in the Organizer.

When a simulator is started, a file selection dialog may be opened if the SDL system contains external synonyms. For more information, see "Supplying Values of External Synonyms" on page 2172 in chapter 51, *Simulating a System*.

The Main Window

The main window provides a text area (which displays output from the monitor system), an input line (used for entering and displaying textual command line input to the monitor system and the SimUI) and a button area with button modules (with buttons for execution of monitor and SimUI commands).

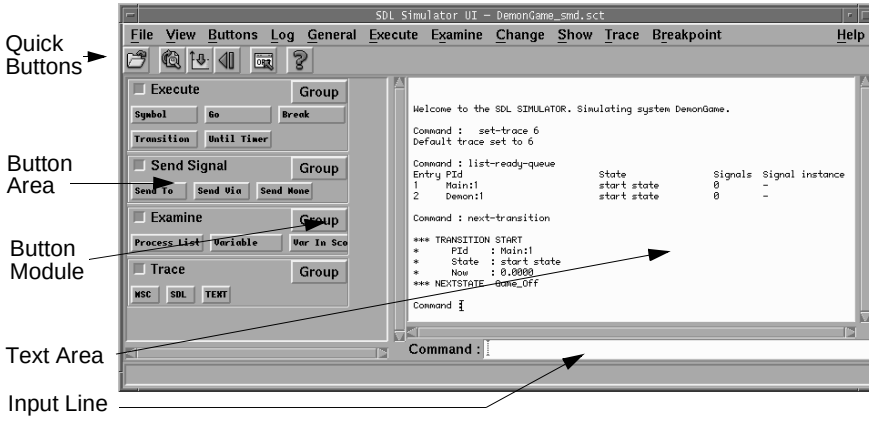

Figure 451: The main window

The Text Area

The *text area* displays all text output from the monitor system, including user prompts, error messages and command results.

Commands cannot be entered in this area, but a command given on the input line or through the use of the command buttons is echoed after the displayed prompt:

```
Command :
```

The Input Line

The *input line* is used for entering and editing commands from the keyboard. For information on available monitor commands, see "Monitor Commands" on page 2077. There are also special SimUI commands that are not sent on to the simulator monitor, see "SimUI Commands" on page 2156.

The 100 latest commands entered on the input line are saved in a history list. The history list can be traversed by using the `<Up>` and `<Down>` keys on the input line.

When `<Return>` is pressed anywhere on the input line, the complete string is saved in the history list and is moved to the text area. The command is then executed.

The input line also has an associated popup menu with the choices *Undo*, *Command*, *Save*, *Go to Source* and *Add to watchwindow*:

- Both *Undo* and *Command* open a dialog with all commands entered so far. In the *Undo* case, all the commands to redo can be selected, and the default is to redo all but the last one, which is equivalent to an Undo of the last command. In the *Command* case, only one of the commands to re-execute can be selected. For more information, see "Undo/Redo Commands" on page 2179 in chapter 51, *Simulating a System*.
- *Save* will save all monitor commands issued by the user so far to a command file. It opens a file selection dialog, in which the name of a command file is selected (**on UNIX** preferably with the suffix `.com`).
- *Go to Source* will go to the SDT reference selected in the text area.
- *Add to watchwindow* will add the variable selected in the text area to the watch window.

Parameter Dialogs

If a command entered on the input line requires additional user input (i.e. parameter values), the information will automatically be asked for in a dialog:

- Parameter values that are file names are selected in File Selection Dialogs.
- Parameter values of enumeration type are presented in lists, from which the value can be selected (see Figure 452).
- Other parameter values are prompted for in simple text input dialogs. In these dialogs, the button *Default value* will enter a "null" value for the parameter in the input field.

Each parameter dialog has an *OK* button for confirming the value and a *Cancel* button for cancelling the command altogether. Some parameters have a default value that does not have to be specified. In this case, an empty value or '-' is accepted for the default value.

Enumeration Type Parameter Dialogs

Additional functionality is available in the dialog for enumeration type parameters:

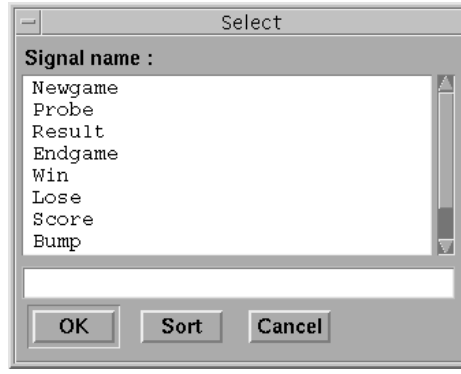

Figure 452: A typical selection dialog

In this dialog, the value can also be entered or edited on the text input line below the list in the dialog. The *Sort* button sorts all values in alphabetical order.

Name completion with the space character is provided. When you press `<Space>` after an initial string, the first value **starting with** the string will be selected (if any). Another `<Space>` will select the next value, etc. When there are no more matches, a space character will be added after the string you initially entered.

A slightly different name completion is provided with the '?' character. When you press '?' after an initial string, the first value **containing** the string will be selected (if any). Another '?' will select the next value. When there are no more matches, a '?' character will be added after the string you initially entered.

For those commands that take an optional variable component, the component must be entered on the text input line after the selected variable name, since it will not be asked for in a dialog. If a '?' is entered instead of a variable component, an additional dialog is opened in which the component can be selected. See the discussion of '?' in "Examine-Variable" on page 2084.

Signal Parameter Dialogs

If a command takes an output signal as parameter, and that signal have parameters, the signal parameters are asked for in a separate dialog. In this dialog, all parameters are listed with their default ("null") values, and they can all be edited in the same dialog. For more information, see "Selecting Signal Parameters" on page 2178 in chapter 51, *Simulating a System*.

Quick Buttons

The simulator has the following quick buttons:

- **Open simulator**: Open an existing simulator executable file.
- **Restart simulator**: Restart the current simulator from the initial system state.
- **Rerun script**: Rerun last executed script.
- **Undo command(s)**: Undo one or more commands.
- **Show Organizer**: Show the Organizer main window.
- **Show Help**: Show help on Simulator user interface.

The Button Area

The *button area* is used for entering monitor or SimUI commands by clicking the left mouse button on a command button. Each button corresponds to a specific command. The buttons are divided into groups, roughly corresponding to the different types of commands listed in chapter 51, *Simulating a System*. Each group is shown as a *module* in the button area. Any number of button modules may reside in the button area. If the modules do not fit in the button area, a vertical scroll bar is added.

The definition of the buttons and button groups are stored in a *button definition file* (see "Definition Files" on page 2152). New buttons can be added and existing ones deleted or redefined by using the *Group* menu in a button module.

To examine a button's definition without executing the command, the left mouse button is pressed on the button and the mouse pointer is

moved outside the button. The command definition then appears in the status bar, but the command is not executed.

If a button's definition contains parameters, the parameter values are prompted for in dialog boxes before the command is executed, in the same way as described for commands entered from the input line. See "Parameter Dialogs" on page 2132.

When no more parameter values are requested, the string shown on the input line saved in the history list and is moved to the text area. The command is then executed.

A Button Module

A *button module* looks like this:

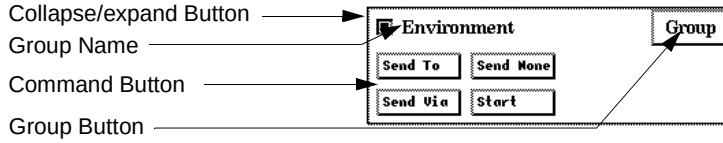

Figure 453: A button module

Each module consists of a title bar and a number of command buttons arranged in rows and columns. The title bar displays:

- A *collapse/expand* toggle button. Clicking on this button collapses the module so that only the title bar is visible, and expands the module back to its current size so that the buttons become visible.
- The *group name* of the button module.
- A *Group* button, providing a menu with commands affecting the buttons in the module.

The *Group* button contains the following menu items:

Add

This menu choice adds a new button to the button module. A dialog prompts for the button label and the command to be executed when the button is pressed. The new button is added to the end of the module. Several buttons may be added with the dialog by using the *Apply* button instead of the *OK* button.

For the syntax of a button definition, see the subsection "Button and Menu Definitions" on page 2153.

> **Note:**
>
> **On UNIX**, if several buttons have **the same button label**, it is always **the first button** found that will be deleted or modified, independently of the selection.

Edit

This menu choice edits the label and definition of a button. The button to edit is selected in a dialog. When a button has been selected, the label and definition can be edited using a dialog.

> **Note:**
>
> **On UNIX**, if several buttons have **the same button label**, it is always **the first button** found that will be edited, independently of the selection.

Delete

This menu choice deletes one or more buttons from the button module. The buttons to be deleted are selected in a dialog.

> **Note:**
>
> **On UNIX**, if several buttons have **the same button label**, it is always **the first button** found that will be deleted, independently of the selection.

Rename Group

This menu choice edits the name of the current button module in a dialog.

Delete Group

This menu choice deletes the current module from the button area. A dialog asks for confirmation.

The Default Button Modules

The following tables list the default buttons in the button modules and the corresponding monitor command. See "Monitor Commands" on page 2077 for more information.

Note:

The buttons in the button modules are specified in the button definition file. If the default button file is not used, the button modules may be different than described here. See "Button and Menu Definitions" on page 2153 for more information.

The *Execute* Module

Button	Monitor command
Symbol	Step-Symbol
Transition	Next-Transition
Go	Go
Until Timer	Proceed-To-Timer
Break	Pressing `<Return>`

The *Send Signal* Module

Button	Monitor command
Send To	Output-To
Send Via	Output-Via
Send None	Output-None

The *Examine* Module

Button	Monitor command
Process List	List-Process -
Variable	Examine-Variable (
Var In Scope	Examine-Variable

The *Trace* Module

Button	Monitor command
MSC	Start-Interactive-MSC-Log 1
SDL	Set-GR-Trace 1
TEXT	Set-Trace 6

The Menu Bar

This section describes the menu bar of the SimUI's main window and all the available menu choices. However, the *Help* menu is described in "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*. Simulator commands are described in "Monitor Commands" on page 2077.

The menu bar contains the following menus:

- *File Menu*
- *View Menu*
- *Buttons Menu*
- *Log Menu*
- *General Menu*
- *Execute Menu*
- *Examine Menu*
- *Change Menu*
- *Show Menu*
- *Trace Menu*
- *Breakpoint Menu*
- *Help Menu*
  (See "Help Menu" on page 15 in chapter 1, *User Interface and Basic Operations*.)

File Menu

The File menu contains the following menu choices:

- *Open*
  (See "Open" on page 9 in chapter 1, *User Interface and Basic Operations*.)
- *Restart*
- *Exit*
  (See "Exit" on page 15 in chapter 1, *User Interface and Basic Operations*.)

Restart

Restart the currently opened simulator. After user confirmation, the currently running simulator is stopped, the *Watch* window is updated, and the text area is cleared from previously executed commands.

The command is dimmed if no simulator has been opened.

View Menu

The *View* menu contains the following menu choices:

- *Watch Window*
- *Command Window*
- *Open All*
- *Close All*
- *Clear Text Area*

Watch Window

Opens the Watch window displaying variable values in the simulation. If this window is already opened, the menu item is dimmed. See "Watch Window" on page 2150 for more information about this window.

Command Window

Opens the Command window in which arbitrary monitor commands will be executed. If this window is already opened, the menu item is dimmed. See "Command Window" on page 2147 for more information about this window.

Open All

Opens the Watch and Command windows.

Close All

Closes the Watch and Command windows.

Clear Text Area

Clears the text area in the main window without affecting the simulation.

Buttons Menu

The Buttons menu contains the following menu choices:

- *Load*
- *Append*
- *Save*
- *Save As*
- *Expand Groups*
- *Collapse Groups*
- *Add Group*

For more information on the SimUI's button definition file mentioned in the menu commands below, see "Definition Files" on page 2152.

Load

Reads in a new button definition file that overrides the current button definitions. All buttons and modules currently in the button area are deleted and replaced by the buttons and modules defined in the new file. A File Selection Dialog is opened for specifying the file to load.

Append

Appends the contents of a new button definition file into the current button definitions. Buttons with new labels are added to the button area, while buttons with already existing labels in the same module will be duplicated (possibly with different definitions). A File Selection Dialog is opened for specifying the file to append.

Save

Saves the current button and module definitions in the button definition file under its current file name.

Save As

Saves the current button and module definitions in a new button definition file. A File Selection Dialog is opened for specifying the new file name.

Expand Groups

Expands all modules in the button area.

Collapse Groups

Collapses all modules in the button area.

Add Group

Adds one or more new button module after the last module in the button area. A dialog box is opened for specifying the name of the new module. Several modules may be added with the dialog by using the *Apply* button instead of the *OK* button.

Log Menu

The *Log* menu manages three different log variants:

- The *input history* contains all textual commands sent to the SimUI.
- The *command history* contains all textual commands sent to the simulator.
- The *complete log* contains all textual output presented during a simulator session, this includes both user commands and simulator output.

The difference between the input history and the command history is that the SimUI processes commands before forwarding them to the simulator. The input history contains the raw user input, while the command history contains pure simulator commands. For instance:

- *Macros*: The input history contains

```
Output-to MySignal ($maxvalue) Main
```

while the command history contains

```
Output-to MySignal (37329) Main
```

- *Check*: This SimUI command, together with its expected output section, is saved in the input history but not in the command history.
- *Execute input script*: This SimUI command is saved in the input history, while the command history contains the executed commands.

The *Log* menu contains the following menu choices:

- *Save Input History*
- *Save Command History*
- *Clear Input History*
- *Clear Command History*
- *Start/Stop Complete Log*
- *Log Status*

Clear Input History

SimUI command: `clear-input-history`

Normally, the input history is cleared when the simulator is restarted. You use this menu choice to clear the input history without restarting the simulator.

Save Input History

SimUI command: `save-input-history`

A file selection dialog appears, where you specify the file to save the input history in. The default file extension for input history files, i.e. input scripts, is `*.cui`.

Clear Command History

SimUI command: `clear-command-history`

Normally, the command history is cleared when the simulator is restarted. You use this menu choice to clear the command history without restarting the simulator.

Save Command History

SimUI command: `save-command-history`

A File Selection Dialog appears, where you specify the file to save the command history in. The default file extension for command history files, i.e. command scripts, is `*.com`.

Start/Stop Complete Log

Start Complete Log starts the logging of complete monitor interaction, i.e. the complete contents of the text area will be logged to a file. A file selection dialog is opened for specifying the log file. If an already existing log file is selected, the user is asked whether to overwrite this file or to append the log to it.

Stop Complete Log stops the complete logging to the file. The appropriate menu choice is displayed depending upon the current state of the log.

Log Status

Displays the status of the command and complete logs in a dialog box. If a log is active, the name of the log file is shown.

Additional Simulator Menus

In addition to the standard SimUI menus, a few special simulator menus are included in the menu bar. The menu choices in these menus simply execute a monitor command, i.e. they are functionally equivalent to buttons in the button modules. If the monitor command requires parameters, they are prompted for using dialogs in the same way as the command buttons.

The following tables list the default menu choices and the corresponding monitor command. See "Monitor Commands" on page 2077 for more information.

> **Note:**
>
> The additional menus in the SimUI are specified in the button definition file. If the default button file is not used, the button modules may be different than described here. See "Button and Menu Definitions" on page 2153 for more information.

General Menu

Menu choice	Monitor command
Command	? (Interactive Context Sensitive Help)
Start SDL Env	Start-SDL-Env
Version	Show-Versions
News	News

Execute Menu

Menu choice	Monitor command
Go	Go
Over Symbol	Next-Symbol
Into Stmt	Step-Statement
Over Stmt	Next-Statement
Finish	Finish
Until Time	Proceed-Until
Until Trace	Next-Visible-Transition
Until Timer	Proceed-To-Timer
Input Script	`execute-input-script`
Command Script	Include-File
Stop Sim	Exit

Examine Menu

Menu choice	Monitor command
Ready Q	List-Ready-Queue
Now	Now
Process List	List-Process
Input Port	List-Input-Port

Menu choice	Monitor command
Signal	Examine-Signal-Instance
Timer List	List-Timer
Variable	Examine-Variable (
Call Stack	Stack
Set Scope	Set-Scope

Change Menu

Menu choice	Monitor command
Ready Q	Rearrange-Ready-Queue
State	Nextstate
Create Process	Create
Stop Process	Stop
Input Port	Rearrange-Input-Port
Del Signal	Remove-Signal-Instance
Set Timer	Set-Timer
Reset Timer	Reset-Timer
Variable	Assign-Value
Synonym File	`set-synonym-file <file name>`

Show Menu

Menu choice	Monitor command
Next Symbol	Show-Next-Symbol
Prev Symbol	Show-Previous-Symbol
C Line	Show-C-Line-Number
Coverage	Show-Coverage-Viewer

Trace Menu

Menu choice	Monitor command
Text Level : Set	Set-Trace
- : Show	List-Trace-Values
SDL Level : Set	Set-GR-Trace
- : Show	List-GR-Trace-Values
MSC Level : Set	Set-MSC-Trace
- : Show	List-MSC-Trace-Values
MSC Trace : Start	Start-Interactive-MSC-Log
- : Start Batch	Start-Batch-MSC-Log
- : Stop	Stop-MSC-Log

Breakpoint Menu

Menu choice	Monitor command
Transition	Breakpoint-Transition
Output	Breakpoint-Output
Variable	Breakpoint-Variable
Symbol	Breakpoint-At
Connect sdle	Connect-To-Editor
Remove	Remove-All-Breakpoints
Show	List-Breakpoints

Command Window

The *Command window* is an optionally visible window used for displaying the results of executed commands. The Command window is opened from the SimUI's *View* menu.

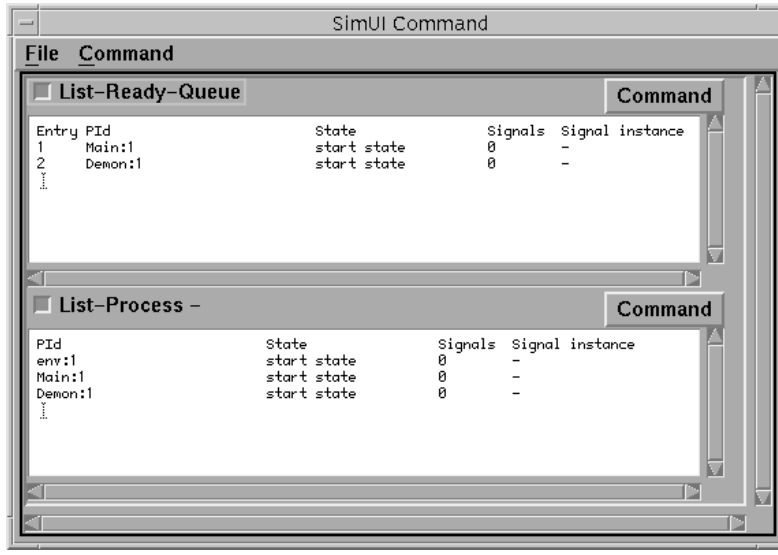

Figure 454: The Command window

The Command window is updated automatically whenever the monitor becomes active and after each monitor command. See "Activating the Monitor" on page 2063 for information on when the monitor becomes active. The window can also be updated manually with a menu command.

Any number of commands can be defined to be executed in the Command window. Each command is executed in a scrollable module in the window.

New commands can be added to the window and existing commands can be changed. The commands to execute are by default List-Process and List-Ready-Queue. The set of commands to execute are stored in a *command definition file* (see "Definition Files" on page 2152). The default command definition file can be changed with the Preference Manager.

Command Modules

A *command module* looks like this:

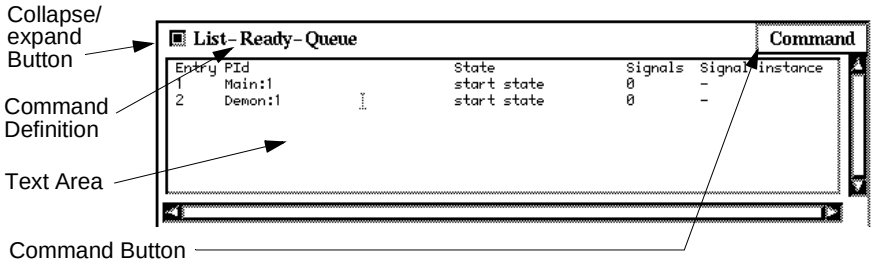

Figure 455: A command module

Each module consists of a *title bar* and a scrollable *text area* for the command output. The title bar displays:

- A *collapse/expand* toggle button. Clicking this button collapses the module so that only the title bar is visible, and expands the module back to its current size so that the text area becomes visible. When a module is expanded, the text area is automatically updated.
- The *command* that is executed.
- A *Command* button, providing a menu with commands affecting the module.

The *Command* button contains the following menu items:

Edit

Opens a dialog for editing the command executed.

Delete

Deletes the command and command module from the Command window.

Size

Sets the size of the text area. A dialog is opened where the number of text rows can be set using a slider.

File Menu

The File menu contains the following menu choices:

- *Load*
- *Append*
- *Save*
  (See "Save" on page 11 in chapter 1, *User Interface and Basic Operations*.)
- *Save As*
  (See "Save As" on page 12 in chapter 1, *User Interface and Basic Operations*.)
- *Close*
  (See "Close" on page 14 in chapter 1, *User Interface and Basic Operations*.)

Load

Reads in a new command definition file that overrides the current command definitions. All commands currently in the Command window are deleted and replaced by the commands defined in the new file. A File Selection Dialog is opened for specifying the file to load.

Append

Appends the contents of a new command definition file into the current command definitions. New commands are added to the command window, but already existing commands are not affected. A File Selection Dialog is opened for specifying the file to append.

Note:

Command names in a command definition file are **case sensitive**. If a command in an appended file already exists in the Command window, but with different case, the command will be duplicated.

Command Menu

The Command menu contains the following menu choices:

Command

Adds a new command to the Command window. A dialog box prompts for the new command. The new command module is added to the bottom of the window. For the syntax of a command definition, see "Command Definitions" on page 2154.

Update All

Updates all command modules by executing the defined commands.

Watch Window

The *Watch window* is an optionally visible window used for displaying values of variables defined in the currently running simulation. The Watch window is opened from the SimUI's *View* menu.

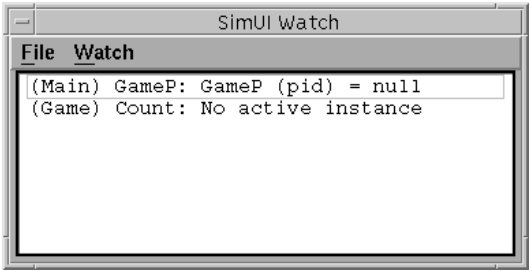

Figure 456: The Watch window

The Watch window is updated automatically after each monitor command. The window can also be updated manually with a menu command.

The variables to display are selected with a menu command. The set of selected variables are stored in a *variable definition file* (see "Definition Files" on page 2152).

File Menu

The File menu contains the following menu choices:

- *Load*
- *Append*
- *Save*
  (See "Save" on page 11 in chapter 1, *User Interface and Basic Operations*.)
- *Save As*
  (See "Save As" on page 12 in chapter 1, *User Interface and Basic Operations*.)
- *Close*
  (See "Close" on page 14 in chapter 1, *User Interface and Basic Operations*.)

Load

Reads in a new variable definition file that overrides the current variable definitions. All variables currently in the Watch window are deleted and replaced by the variables defined in the new file. A File Selection Dialog is opened for specifying the file to load.

Append

Appends new variable definitions to the current variable definitions file. The variables in the file are added to the Watch window. A File Selection Dialog is opened for specifying the file to merge. Duplicate variable definitions are permissible.

Watch Menu

The Watch menu contains the following menu choices:

- *Update All*
- *Add*
- *Edit*
- *Delete*
- *Delete All*

Update All

Updates the Watch window by showing the current value of all displayed variables.

Add

Adds one or more variables to the Watch window. A dialog is opened for specifying the variable to be added. For the syntax of a variable definition, see "Variable Definitions" on page 2155. Several modules may be added with the dialog by using the *Apply* button instead of the *OK* button.

Edit

Edits a variable specification in the Watch window. The variable whose specification is to be changed is selected in a dialog. When a variable has been selected, the specification can be edited using a dialog.

Delete

Deletes one or more variables from the Watch window. The variables to delete are selected in a dialog.

Delete All

Deletes all variables from the Watch window.

Definition Files

In the SimUI, the following types of information are stored on files:

- **Button definitions**

  i.e. definitions of button groups and button commands in the main window's button area.

- **Menu definitions**

  i.e. definitions of additional menus and menu commands in the main window's menu bar.

- **Command definitions**

  i.e. definitions of commands to be executed in the Command window.

- **Variable definitions**

  i.e. definitions of variables to display in the Watch window.

At start-up of the SimUI, the files to read are determined in the following way:

1. The file names are defined with the Preference Manager. If a file name is not defined there, the default file name `def.btns`, `def.cmds` and `def.vars` is used, respectively.

2. If the file names does not contain a directory path, the files are searched for in the following directories and the following order:

   – the current directory
   – the user's home directory
   – the installation directory

Once a file has been found, it is read and the contents of the corresponding window are set up. If a file cannot be found, the corresponding window area becomes empty.

Common File Syntax

Each of three text files can contain comment lines starting with the '#' character. Empty lines are not discarded.

> **Note:**
> When a file is read, **no checks are made upon the relevance** or correctness of the definitions contained in the file.

Button and Menu Definitions

The *button and menu definitions* are stored in a button definition file with the default extension `.btns`. The button definitions are divided into groups where each group defines a button module in the main window's button area, or a menu in the main window's menu bar.

Syntax

In the file, a button group has the following syntax:

```
:[COLLAPSED] <group name>
<button label>
<definition>
<button label>
<definition>
  . . .
```

The <group name> is the string shown in the title bar of the button module. If the group name is prefixed with the string `COLLAPSED`, the button module is initially collapsed. The <button label> is the label of the button in the button module. The <definition> is the monitor command that will be executed when the button is pressed. The syntax of a button definition is the same as when entering a command textually to the monitor.

A menu has the following syntax:

```
:MENU <menu name>
<menu choice>
<definition>
<menu choice>
<definition>
 . . .
```

The <menu name> is the name of the menu shown in the menu bar. The <menu choice> is the name of the menu choice in the menu. The <definition> is the monitor command that will be executed when the menu choice is selected.

In the monitor command definition, a '?' as parameter will not work, but a hyphen '-' can be used to signify the default value. Missing parameters at the end of the command will open dialogs for those parameters.

Defining Multiple Commands

In addition, a button definition may consist of several monitor commands, if they are separated by a space and a semicolon, i.e. ";". The commands are then executed immediately after each other.

Example 319: Multiple Commands in Simulator ––––––––––––––––––

```
next-transition ; out-via probe -
```

This button definition executes the next transition and then sends the signal Probe from the environment.

––

Command Definitions

The command definitions are stored in a command definition file with the default extension `.cmds`. Each command definition defines a command module in the Command window. The file has the following syntax:

```
<command>
<command>
   . . .
```

The <command> is the monitor command that will be executed in the command module. The syntax of a command definition is the same as when entering a command textually to the monitor. All parameter values must be specified explicitly or with the default value '-', i.e. '?' is not allowed, and no parameters may be missing. Command names are case sensitive.

> **Note:**
>
> Care should be taken when deciding what command to execute; commands belonging to the Execute group **should be avoided**.

Variable Definitions

The variable definitions are stored in a variable definition file with the default extension `.vars`. Each variable definition defines a variable to display in the Watch window. The file has the following syntax:

```
(<process>) <variable>
(<process>) <variable>
   . . .
```

The <process> is the name of the process, possibly augmented with an instance number separated with a space or colon. The <variable> is the name of the variable in that process. Names are **not** case sensitive.

Example 320: Variable Definitions in Simulator ––––––––––––––––––

```
(Main) Count
(game:1) guess
```

––

Macros

The SimUI has a built-in macro facility. A macro can represent a simulator command or a part of a simulator command. Even though SimUI macro commands are processed by the SimUI, it is possible to enter SimUI macro commands almost as if they were normal simulator commands:

- Type in the SimUI macro command. Note though, that SimUI commands cannot be abbreviated in the same way as simulator commands.
- Define a button or a menu for the SimUI macro command.

These commands are available for macros:

- Use **add-macro** to define a new macro:

  ```
  add-macro <macro name> <macro value>
  ```

  Example:

  ```
  add-macro maxvalue 37329
  ```

- Use **list-macros** to see all defined macros and their values:

  ```
  list-macros
  ```

  Example:

  ```
  Command: list-macros
  Number of macros defined: 1
  Macro name: maxvalue
  Macro value: 37329
  ```

- Use **$** to get the value of a defined macro:

  ```
  $<macro name>
  ```

  Example: The SimUI input:

  ```
  Output-to MySignal ($maxvalue) Main
  ```

  is sent to the simulator as

  ```
  Output-to MySignal (37329) Main
  ```

- Use **remove-macro** to remove an already defined macro:

  ```
  remove-macro <macro name>
  ```

  Example:

  ```
  remove-macro maxvalue
  ```

SimUI Commands

The SimUI examines entered textual commands before sending them to the simulator. If they are SimUI commands they are not sent to the simulator.

Here is a list of all SimUI commands and what they do:

- `add-macro`

  Adds a new SimUI macro. For details, see "Macros" on page 2155.

- `check`

  Tells the SimUI that the textual simulator output from the next command should be checked.

  – When commands are entered manually and a check command is issued, the SimUI saves the simulator output for the next command in an expected output section in the input history.

  – When commands are read from file by executing an input script, and a check command is encountered, the SimUI compares the simulator output for the next command with the expected output from the input script. The SimUI reports if the check passed or failed.

- `clear-command-history`

  Clears the simulator command history without restarting the simulator. This command is useful if you want to save a command history that does not start from the beginning of a simulator session.

- `clear-input-history`

  Clears the SimUI input history without restarting the simulator. This command is useful if you want to save an input history that does not start from the beginning of a simulator session.

- `execute-input-script`

  Executes the commands found in an input script, i.e. an input history saved to file.

  – If a relative file name is used, then it is interpreted as relative to the Organizer source directory.

  – If parameters are given as a space separated list after the file name, then these parameters can be accessed with $1, $2, ... in the called script. It is also possible to access all parameters with $*, the number of parameters with $# and the name of the calling script with $0.

- `list-macros`

Produces a list of all defined macros and their values. For more details, see "Macros" on page 2155.

- `remove-macro`

  Removes an already existing SimUI macro. For more details, see "Macros" on page 2155.

- `save-command-history <file name>`

  Saves the simulator command history in a command script file (`*.com`). If no <file name> is given, a file selection dialog appears, where you can specify a file name.

- `save-input-history`

  Saves the SimUI input history in an input script file (`*.cui`). If no <file name> is given, a file selection dialog appears, where you can specify a file name.

- `set-synonym-file <file name>`

  Sets the synonym file to use the next time the simulator is restarted from the simulator UI. If no <file name> is given, a file selection dialog appears, where you can specify a file name. Note that the synonym file can also be set from the menu choice SimUI>Change>Synonym File.

Regression Testing

By using the simulator for regression testing, it is possible to check that an enhanced version of an SDL system does not break the old functionality, that is still expected to work for the same SDL system.

With regression testing in the simulator, you can for instance:

- Send in signals and check that the correct signals are sent back out again.
- Check that a variable has the correct value at a specific moment.

The basic idea is to save and replay the input history, and check the textual simulator output for selected commands. An input history saved to a file is called an input script. A group of input scripts can be collected in the Organizer together with the SDL system. The Organizer provides the possibility to run a group of input scripts (or test cases). This results in a textual summary where it is easy to see any test case failures. To

examine a test case failure in detail, it is possible to run a single test case with the SimUI command `execute-input-script` (see "SimUI Commands" on page 2156).

Note that it is also possible to express test cases with the MSC notation. MSC test cases are converted to input scripts before being run in the simulator.

Here is a small example, to see how it works:

Example 321 ————————————————————————————

In System DemonGame (that is provided as an example in the SDL suite installation), you can (after some initialization) send in a Result signal and expect to get a Score signal back.

To make a test case testing that we get a Score signal back:

1. Create a simulator for System DemonGame.
2. Execute the following simulator commands (you do not have to type in the comments):

```
- Output-To NewGame Main /* Send in signal NewGame
  to start a new game. */
- Next-Transition /* Execute the start transition
  for process Main. */
- Next-Transition /* Execute the start transition
  for process Demon. */
- Next-Transition /* Execute the transition receiv-
  ing signal NewGame and creating process Game */
- Output-To Result Game /* Send in a signal to get
  the current score. */
- break-output - env - - - - - - /* Set a breakpoint
  on signals sent to the environment. */
- Set-Trace System DemonGame 0 /* Minimize textual
  trace. */
- check /* Check the output from the next command. */
- go /* Execute as far as possible. The simulator
  will stop and report that the Score signal has been
  sent to the environment. */
```

3. The go command that we are checking produces the following textual simulator output:

```
Breakpoint matched by output of Score
```

4. Save the input history in an input script, with the SimUI command `save-input-history`. The last part of the input script will look like this:

```
check
go
expected-output start
Breakpoint matched by output of Score
expected-output end
```

5. Add a symbol for the input script in the Organizer, with *Edit > Add New*, plain text, do not show in editor.
6. Connect the symbol to the input script file with *Edit > Connect*.
7. Now, make a test: Deselect everything in the Organizer (to make sure that both the SDL system and the input script will be considered) and choose *Tools > Simulator Test*. After a dialog, a fresh simulator will be created and the test case run. The result is presented in the Organizer Log.
8. To force a test case failure:
   - Edit the test case (by double-clicking on the Organizer symbol to get a text editor) by changing the expected output to be something different.
   - Save the test case.
   - Repeat the above test.
   - If everything works as expected, the test case fails.
9. To examine the details of a test case failure:
   - Restart the simulator.
   - Execute the test case (input script) in the SimUI with the `execute-input-script` command.
   - If everything works as expected, the expected output from the go command did not match the actual output.

Mapping Instances to Different Environments

When generating test scripts (.CUI scripts) from MSC:s, it is possible to select for which specific processes/blocks for which a cui script shall be generated if an MSC testcase describes several blocks.

Through an instance translation table, which is a plain text file in the Organizer with the extension `*.itt`, it is decided which instances in the MSC that should be regarded as environment. In this file it is for example stated that an instance A really is instance `env:2`.

```
Example *.itt file:
MyBlock env:2
MyProcess env:3
```

The name used for an instance in the MSC is placed first, followed by a space, and then comes the name of the environment instance to map the MSC instance to. Note that the only names that can be used for mapping are `env:2`, `env:3`, `env:4` etc and that no numbers can be left out when mapping, i.e. it is not allowed to only use `env:2` and `env:4` without using `env:3`.

Whenever an MSC is converted to a CUI test script, the first found (or closest found) translation table from the Organizer is used and all instances in the file are translated according to the table in the file. All instances starting with `env` (after translation, if there is a translation file) are outside the system, and all other instances are assumed to be inside the system.

Order of execution

In the case where the MSCs used as test scripts only have one environment instance, the order of execution is the same order as messages are connected to the environment instance in.

To determine the order things are done in when several instances are converted to environment instances, the y-coordinate of the connection of messages to environment instances is used. If the y-coordinate of sending in message A is above the y-coordinate of sending in message B, then message A is sent before message B. If two connections have the same y-coordinate, then the x-coordinate is used to determine the order. The left most connection is done first.

Only messages connected to an environment instance in one end and a system instance in the other end are used to drive the simulation. Messages between environment instances and messages within the system are ignored.

For action symbols (that can contain ordinary simulator commands or shortcuts for ordinary simulator commands), the upper connection point to the environment instance is used for determining the order of doing things in.

It is also possible to test in an MSC test script if a service response message is returned to the same environment instance (representing a specific external process) that sent in the service request message, in the

case where there are many environment instances. This is done with the *output-internal* simulator command that is generated by the MSC to CUI converter. An example:

```
output-internal SignalIn ProcessOne env:2
```

If the system responds to this message by sending back another message to sender, then the textual trace will contain information about what environment instance the signal was sent to:

```
* OUTPUT of SignalOut to env:2 from ProcessOne:1
```

In addition to the output-internal command, the create null env command is generated in the CUI script by the converter. This command will create the environments in the simulator that will be used for the instances that are mapped to an environment instance in the `.itt` file.

Note that when instances are mapped with an `.itt` file, process id mapping should not be used.

Restrictions

Restrictions on Monitor Input

The following restrictions apply to the monitor input:

- A parameter to a monitor command may not contain more than 5,000 characters.
- **On UNIX**, control characters of different types may terminate the simulation program. `<Ctrl+C>`, `<Ctrl+D>`, and `<Ctrl+Z>` are typical characters that might terminate a simulation program.

Restrictions on Dynamic Checks

There are a number of dynamic checks that are not performed at all or performed at the C level by the C runtime system. A C runtime error will of course lead to the simulation program being terminated. The following checks are not made at the SDL level.

- Several paths from a decision for a certain decision expression value. A simulation program will simply choose one of the existing paths.
- Overflow of integer and real values are checked at the C level if the actual C system performs these checks. (These checks are likely not to be performed.)

Chapter 50 The SDL Simulator