Execution Monitoring Reference

$Revision: 1.27 $

A. M. Sloane

Department of Computing
Division of Information and Communication Sciences
Macquarie University
Sydney, NSW 2109
Australia

Table of Contents

This manual describes how to use the Noosa monitoring system to diagnose problems in Eli-generated programs or simply to observe their behaviour. While this manual provides general information about the system and the types of information that can be obtained, the reader is directed to the Noosa online help for more specific information on how to operate the user interface. (See the Help menus in most windows.)

1 Using Noosa

This chapter describes how to invoke the Noosa system and run your program under the control of Noosa.

1.1 Invoking Noosa

Noosa is invoked from within Eli using any of the `:mon`, or `:mongdb` products in conjunction with the `+monitor` parameter. See Products and Parameters Reference Manual, for details on how to use these products. Because Noosa is based on an X11 window system toolkit, you must be running an X session when Noosa is invoked.

Invoking Noosa will produce a window containing two main areas: an input text window and a transcript window. Noosa has a fairly conventional menu-based interface. See the Help menus in most windows for general information about how to use Noosa and to get specific descriptions of all menu commands.

The most important menu command to know at this stage is Quit in the main Noosa menu since it gets you out of Noosa when your monitoring session is over. You will then be able to resume your interactive Eli session.

1.2 Controlling Your Program

The text entry area just under the menu bar in the main Noosa window contains the name of your program and command-line arguments. Normally you shouldn't need to worry about the name of your program. It will be set by Eli and will refer to a file in your Eli cache.

The program arguments will be as specified using the `+arg` parameter when you invoked Noosa (see Products and Parameters Reference Manual). If you didn't specify +arg then the program will not be given any command-line arguments when run. The arguments can be edited in the text entry; there is no need to exit Noosa and re-enter with a new `+arg` setting if the argument values must be changed. You can specify as many `+arg` parameters as you like; their values will all be passed as command-line arguments.

Use the Execution menu to control the execution of your program by running it (Run command), continuing from a stoppage (Continue) or killing the process entirely (Kill). For convenience the same menu can be obtained by pressing the middle button in either the input text window or the transcript window. These commonly used commands are also available via the keyboard shortcuts `ALT-R`, `ALT-C`, and `ALT-K`, respectively.

1.3 User Initialisation

When Noosa begins execution it loads user initialisation files called '`.noosarc`' from the user's home directory and the current directory in that order (if they exist).

A '`.noosarc`' can contain arbitrary Tcl/Tk code to initialise the Noosa system. A complete description of the Tcl language and Tk toolkit is beyond the scope of this manual. See the Tcl/Tk online manual pages or any of the available books for details.

The '`.noosarc`' interface is presently mostly undocumented. Future versions of this manual will describe in detail how Noosa can be configured using a '`.noosarc`' file.

One facility that *is* documented is the ability to use a '`.noosarc`' file to autoload event handlers. The Handlers window allows you to save the current state of your handlers in a file. Suppose that you save them in '`myhandlers.hnd`'. That file can loaded on startup by placing the following lines in a '`.noosarc`'.

```
source myhandlers.hnd
```

Noosa also loads any files of type `tcl` that are present in your Eli specifications. You can provide any extra monitoring support you need in these files. See Section 3.1.6 [Browsing non-standard types], page 18, for information on how to provide Tcl support for browsing your own data types.

1.4 Changing files from within Noosa

It is often useful to be able to edit files while monitoring your program. For example, you might want to change the test data being used, or you might want to fix bugs in your specifications before you forget about them. The Files command in the Windows menu brings up a window from which you can open arbitrary files and edit them.

If you change your specifications while monitoring, Eli will only notice the changes if you have the *VerifyLevel* variable in Eli set to 2. (See ⟨undefined⟩ [Variables], page ⟨undefined⟩, for more information on influencing Eli with variables.)

1.5 X resources used by Noosa

Noosa is written using the Tk X11 toolkit. Thus you can set any X11 resources that Tk supports. These include settings for the fonts used in various types of window, the colours used to highlight various regions, and so on. For complete documentation of the resources that Tk supports, see the Tk documentation.

To illustrate the use of Tk resources, suppose that you want to change the fonts used for text and entry windows, and alter the colours used for the selection (Noosa default: red background and yellow foreground). The following settings might be used in your '`.Xdefaults`' file.

```
Noosa*Text.font:         -adobe-courier-bold-r-*-*-12-*-*-*-*-*-*-*
Noosa*Entry.font:        -adobe-courier-bold-r-*-*-12-*-*-*-*-*-*-*
Noosa*selectBackground:  blue
Noosa*selectForeground:  green
```

To make common situations easier, Noosa also supports some specific resources.

`Noosa.width`
: The width in characters of text windows except file windows (see below) (default: 80).

`Noosa.inputHeight`
`Noosa.transHeight`
: The height in lines of the input and transcript windows (default: 15 and 15).

`Noosa.handHeight`
: The height in lines of the text part of the handlers window (default: 20).

`Noosa.fileWidth`
`Noosa.fileHeight`
: The width in characters and the height in lines of the file editing windows (defaults: 80 and 30).

`Noosa.treeWidth`
`Noosa.treeHeight`
: The width and height in pixels of the tree windows (see also below) (defaults: 400 and 300).

`Noosa.valueColour`

The colour used to highlight values in the transcript window that can be opened (default: blue).

`Noosa.nodeColour`

The colour used to highlight nodes in the abstract tree displays (default: red).

Note that in each case the width and height of a window is the actual display area of the window, not including any borders.

Noosa also allows fine-grained control over the sizes of the various tree displays. The following resources control the sizes of the four different types of tree display. If these resources are not set, the values of `Noosa.treeWidth` and `Noosa.treeHeight` are used.

`treeFullWidth`, `treeFullHeight`, `treeSrcWidth`, `treeSrcHeight`, `treeCompWidth`, `treeCompHeight`, `treeIncrWidth`, and `treeIncrHeight`.

2 Information

This chapter briefly describes the type of information that Noosa can provide about the execution of your program and how to go about getting it.

2.1 Input Text Display

Usually the initial input to an Eli-generated program is specified on the command-line of the processor (see Command Line Processing) and subsequent input (if any) is given by the input text itself (perhaps via `include` directives or similar mechanisms).

Noosa displays the input text as seen by your program in the top part of the main window. (The current Noosa system does not fully support the monitoring of programs when their input is standard input.) The input text is shown *exactly* as your program sees it. In particular, it appears as one contiguous piece of text rather than (say) a set of files included into other files. Note also that when the program stops, the input text displayed is the text that has been seen by the program at that point. Text encountered later on (perhaps by later `include` directives) will be displayed when it is encountered.

In various settings Noosa will display input text coordinates (e.g., when you ask to see the lexical tokens recognised). There are two formats used to display coordinates:

```
12,3
9,1-12,80
```

The first form indicates a single coordinate (column three of line twelve); the second indicates a range of coordinates (column one of line nine through to column eighty of line twelve, inclusive).

In the transcript window coordinates or coordinate ranges will be underlined. Clicking the left mouse button on a displayed coordinate (or range) causes Noosa to highlight the coordinate (or range) in the input text. This enables you to conveniently match Noosa output to input text.

As mentioned above, the input to your program may come from more than one text file. The coordinates used by Noosa are *cumulative* in that they reflect the overall input text, not the individual text files. To find out from which file a location comes, select the location in the input text window and execute the Describe coord command from the Examine menu (also available on the right button in the main windows).

2.2 Messages

Eli-generated programs take text as input, analyse that text, and perhaps produce some text as output. During analysis, messages may be produced for a variety of reasons. Eli provides a module to help generate messages (see Library Reference Manual).

If your program generates any messages they will be displayed in the transcript window. The coordinate of the message will be shown with the severity and the message text.

2.3 String Table

Most Eli-generated programs need to manipulate text strings. To avoid the overhead of copying strings around during execution, a string table can be used. Eli has a module that implements a string table allowing integers to be used to represent strings (see Library Reference Manual).

Noosa allows you to see the contents of the string table using the Strings command from the Examine menu. Each string in the table will be displayed with its index. The String command can be used to display a single particular string. Select the numeric string index with the mouse then execute String. This mode of use is particularly useful if the string index has already been displayed by Noosa in some other setting (e.g., as the intrinsic attribute of a token).

2.4 Lexical Structure

Eli-generated programs that perform lexical analysis can do so using the support of an automatically-generated lexical analyser (see Lexical Analysis). Noosa lets you examine the behaviour of the generated analyser on your program's input text.

Your program will generate a stream of tokens. Selecting an input text coordinate (or range of coordinates) in the input text window and executing the Token command from the Examine menu will cause Noosa to display the tokens recognised that overlap that coordinate (or range).

The following information is displayed for each token: input text coordinate range, numeric token code used internally by the analyser, length in characters, the intrinsic attribute value of the token, the input text (lexeme) matched by the token, and, for non-literal tokens, the name of the non-literal as specified in your type-'`gla`' specifications (see Lexical Analysis).

2.5 Phrase Structure

Programs that need to determine the phrase structure of their input can do so within Eli using automatically-generated parsers (see Syntactic Analysis).

The Noosa Phrase command (in the Examine menu) lets you look at the phrase structure that is recognised by your parser. Selecting an input text coordinate in the input text window and executing Phrase will produce a list of all the production instances recognised by your program that overlap the selected coordinate.

The instances are listed from most general to most specific, so the first one is always the root production of the grammar. Each production instance is displayed with the input text coordinate range for that instance. On the right-hand side of each production the symbol corresponding to the left-hand side of the following production is highlighted. (Note that in some cases chain production elimination is performed by Eli-generated parsers. This may mean that the highlighted symbol on the right-hand side of a production instance is not the same symbol as the left-hand side of the next production instance.)

Examination of the phrase structure with Noosa will work if you are using either the PGS or COLA parser generating systems available within Eli (see Products and Parameters Reference Manual).

2.6 Trees and Attribute Values

If your processor contains attribution, Eli will automatically construct an abstract tree for the input text. Noosa has facilities for examining this tree and any other trees computed by your processor.

Tree display can be enabled using the Trees item in the Windows menu. Windows containing the selected trees will appear next time the program is run. There are four options in the Trees menu: Just Source, Separate Computed, Source and Computed, and Incremental. Any combination of these options can be used.

The first three options draw trees in a traditional tree manner with the root at the top and children under their parents. The Incremental option draws the root at the left and children to the right of their parents. The former style always uses a nice layout but always draws the whole the tree (but see below); the latter initially just draws the root of the tree but allows nodes to be selectively expanded (see the online help for details).

The Just Source option causes the source tree built by your processor's parser to be displayed. The Separate Computed option will cause each computed tree to be displayed in a separate window as soon as they are complete. This option is most suitable if you have a few largish computed trees. The Source and Computed option shows the entire tree (including computed trees joined at the appropriate places) so it is more suitable if you have many smaller computed trees. Finally, the Incremental display allows access to the entire tree that has been computed so far.

The tree displays can be saved as Postscript via the Tree menu. You can elect to save just the visible portion of the tree or the whole tree.

In any of the tree displays it is possible to select nodes with the left button. The abstract grammar production derived at that node will be displayed in the transcript window and the input text extent of the node will be highlighted in the input text window.

Also, the right button can be used on a symbol (rule name) to display a menu listing the attributes (attributes and terminal values) of that occurrence of the symbol (rule). Each attribute or terminal has a pull right menu with which you can indicate whether you want to see its value (with optional stopping of execution) or ignore it (the default). Using this facility you can check that your attribution is working correctly. Note that values will only be displayed when they are next calculated, so you will need to run the program again after selecting some values for display.

Note: The current version of Noosa is not able to deal properly with chain attributes. Chain attributes will show up in the attribute menu as a pair of regular attributes with `_pre` and `_post` appended to the attribute name. It is possible to select these attributes for display. However, in the current system, not all will be displayed because of limitations in the generated processor code.

Noosa has a simple mechanism for displaying the values of attributes in the transcript window. Values are displayed preceded by their type name. If a value can be browsed (or "opened") it will be underlined and browsing is performed by clicking on the value with the left button.

Eli currently has support to allow the following types of value to be browsed.

Tree nodes (Node, NODEPTR)
: Clicking on a tree node value causes your abstract tree display(s) (if any) to highlight that node. A NODEPTR value is a run-time pointer to a tree node. Clicking on one of these values will select the corresponding node in a tree display if it is there.

PTG nodes (PTGNode)
: Opening a PTG node causes the system to run the function `PTGOutFile` on the node and display the resulting output in the transcript window. Note that due to side-effects

in PTG functions or redirected output, the text displayed may not be same as the text finally output by your processor.

Environments (Environment)

Opening an environment produces in the transcript a list of the name-key pairs in that environment. If the environment is nested within another environment then the parent environment is printed so that it can be browsed as well.

Bindings (Binding)

Opening a binding will produce the identifier that is bound (`IdnOf`), the key to which it has been bound (`KeyOf`), and the environment containing the binding (`EnvOf`).

Definition table keys (DefTableKey)

Opening a definition table key will produce a list of the current properties of that key and the values of those properties.

OIL types and typesets (tOilType, tOilTypeSet)

These types are used for operator identification. Opening an OIL type shows the type name (a definition table key). Opening an OIL typeset shows the elements of the set and their associated costs.

Tree parser nodes (TPNode)

Opening a tree parser node will produce the node name and a list of its children.

2.7 Breakpoints and events

Noosa follows the progress of your program using *events*. When a significant thing happens during execution the program will generate an event to signal that fact to the monitoring system. Event instances have *parameters* which allow them to provide arbitrary information to the monitoring system.

Breakpoints in the Noosa system are conceptually similar to breakpoints in source-level debuggers, but operate at the level of events rather than source code locations, functions or variables. They are implemented by attaching handlers to event types.

The Handlers command in the Windows menu creates a dialog window through which you can enter handlers for the different types of events that your program may produce during execution. A list of relevant event types is displayed and handlers can be entered, edited, deleted etc. (See the Help menu in the Handlers dialog for more information.) Handlers can also be saved to files and autoloaded, see See Section 1.3 [User Initialisation], page 3.

Handlers are expressed using the Tool Command Language (Tcl). (A complete description of TCL is beyond the scope of this manual. See the Tcl online manual pages or any book on Tcl/Tk for details.) Handlers can contain arbitrary Tcl code and may refer to the event parameters as Tcl variables. To cause execution to stop as the result of handler execution, have the handler call the Tcl `n_break` command.

For example, the following handler causes execution to stop if the string `printf` is stored into the string table. This handler would be attached to the `string_stored` event.

```
if {$string == "printf"} {
    n_break
}
```

There is no requirement that a handler actually cause execution to stop. It may just display information and allow execution to continue. Within a handler, the builtin Noosa command `n_say` may be used to display information in the Noosa transcript window.

For example, the following handler causes the lexeme of every token on line three of the input to be displayed. This handler would be attached to the `token` event type thereby making the `linebeg` and `lexeme` parameters available.

```
if {$linebeg == 3} {
    n_say "lexeme is $lexeme\n"
}
```

All Eli-generated programs prepared for monitoring automatically generate a single event instance of type `init` at the beginning of execution, and one of type `finit` at the end of execution. This can be useful if you want to collect some information using handlers during execution and display a summary at the end using a handler on the `finit` event type.

2.8 Frequency Profiles

Frequency profiles provide information about the frequency of events generated by your program. When execution stops, a summary of events generated up to that point will be produced. The summary contains the name of each event type generated and the count of the number of times events of that type were generated by a particular component of the program.

Frequency profiles are enabled and disabled by the Frequency profile checkbutton in the Profile menu. By default they are disabled. The Zero frequencies command can be used to set all of the frequencies to zero. This can be useful if you only want to collect frequencies from a particular point during the execution.

2.9 Time Profiles

Time profiles provide information about the CPU time spent in components of your program. For each component the CPU time in seconds is given with the percentage of total CPU time due to that component.

Time profiles are enabled and disabled by the Time profile command in the Profile menu. By default they are disabled. The Reset times command can be used to set all of the times to zero.

Time profiles are obtained by generating an `enter` event each time execution enters the code for a component, and a `leave` event when execution leaves again. Consequently, time is only allocated to components which have appropriate monitoring support. Currently, the main components within Eli have this support, but not all components do. Also, due to the short running time of most Eli-generated programs on test input, the times reported in a time profile are likely to vary considerably from run to run due to the granularity of the timing mechanisms. Consequently, time profiles should only be relied on when using large inputs or running time is larger for some other reason.

2.10 Tracing Events

Sometimes it is useful to see the event stream generated by your program. The Event trace command in the Profile menu provides this capability. When tracing is enabled Noosa will display the event type and parameters of every event generated by the program until it stops.

The Set event filter command allows subsets of events to be selected using a regular expression. A dialog box allows you to set a new expression or clear an old one. Executing the Set event filter command will cause subsequent tracing to display an event only if the event information matches the regular expression. The default regular expression is `.*` meaning all events are displayed.

3 Implementation

This chapter describes some of the implementation of Noosa in detail. Eli users who just want to perform monitoring with existing monitors do *not* need to read this chapter. It is intended for Eli developers or advanced users who want to extend the capabilities of Noosa.

3.1 Monitoring Interfaces

Noosa needs to obtain information from the running program. It uses the program's *monitoring interface* to do it. A program's monitoring interface is the union of all of the monitoring interfaces of the components making up that program. The contents of the monitoring interface for a component depend on the nature of the component and the information that it wants to make available to the monitoring system.

Monitoring interfaces are described by type-'`dapto`' files. (See Section 3.6 [Dapto Grammar], page 24, for the syntax of the Dapto language.) Dapto files contain the information described in the following. Examples are taken from the monitoring interface for the string table module in Eli (see the file '`pkg/Adt/csm.dapto`' in the Eli distribution).

In the following discussion, two pre-defined data types: `int` and `str` are used. These correspond to the C data types `int` and `char *`, respectively.

3.1.1 Aspects

All elements of a monitoring interface are grouped together into *aspects* (similar to a module). The names of aspects are used to enable the monitoring system to decide what components are present in the program. Some monitoring commands are only applicable to programs which provide the aspects on which the monitor depends. For example, the Phrase command can only be used on programs that contain parsers. See Section 3.3 [Database], page 21, for more details on this mechanism.

An aspect syntactically encloses the interface elements which it contains.

```
aspect string;
     Interface elements of the string aspect
end;
```

3.1.2 Event Types

Event types are described in a monitoring interface by giving their names plus the names and types of their parameters. We also enforce the inclusion of documentation strings for each of these entities to enable the user interface to provide readable descriptions of events where necessary.

The string table monitoring interface contains one event, `string_stored`, which is generated whenever a string is inserted into the table. Consequently we have the following event description in the monitoring interface:

```
event string_stored* "Storage of a new string in the string table"
    (int index "Index of new string", str string "New string");
```

Normally event types are assumed to be hidden from the user. If you want the events of a particular type to be visible to the user through the Handlers window, it is necessary to append a `*` to the name of the type, as is done in the example above.

3.1.3 Operations

Operation signatures are described in the monitoring interface by giving the name of the operation, its parameters (if any), its return type (if any), along with documentation strings. Currently the return type of an operation must be `str` or there must be no return type.

Here is the signature for the string table `get_string` and `set_string` operations:

```
operation get_string "Look up a string given its index"
    (int index "Index of the string to be looked up") : str

operation set_string "Change the value of a stored string"
    (int index "Index of string to be changed",
     str value "New value for string")
```

Operation implementations are given in C following the operation signature. Any legal C code can be used in an operation definition, except that C `return` statements should not be used and to return values from an operation you must use the following macros:

`DAPTO_RESULT_STR(char *s)`

Append the string s to the result to be returned by this operation.

`DAPTO_RESULT_INT(int i)`
: Append the integer i as a string to the result to be returned by this operation.

`DAPTO_RESULT_LONG(long l)`
: Append the long integer l as a string to the result to be returned by this operation.

`DAPTO_RESULT_PTR(void *v)`
: Append the arbitrary pointer v to the result to be returned by this operation. The value will be passed as a long integer and won't be interpreted by Noosa. To be useful, this value must later be passed back to another part of the monitoring interface where it can be used as a pointer again.

`DAPTO_RETURN`
: Return the current result as the value of this operation.

Use of the `DAPTO_RESULT` macros sets up a value that is returned when the end of the operation is reached. To return from the middle of an operation use the `DAPTO_RETURN` macro with no arguments.

For example, the following is the full definition of the `get_string` operation:

```
operation get_string "Look up a string given its index"
    (int index "Index of the string to be looked up") : str
{
    if ((index < 0) || (index >= numstr)) {
        DAPTO_RESULT_STR ("*** Illegal string table index ***");
    } else {
        char *s = string[index];
        if (s == (char *) 0) {
            DAPTO_RESULT_STR ("*** No string at this index ***");
        } else {
            DAPTO_RESULT_STR (s);
        }
    }
}
```

The `DAPTO_RESULT` macros for integer, long and pointer values should only be used with arguments whose addresses can be taken. For other values (e.g., return values from function calls or the values of expressions) there are analogous macros whose names are formed by appending `VAL` to the macro name. For example, the first of the following calls will not compile; the second must be used.

```
DAPTO_RESULT_INT (i + 1);
DAPTO_RESULT_INTVAL (i + 1);
```

The `VAL` forms of the macros can always be used, but they incur the cost of an extra copy compared to the non-`VAL` form.

3.1.4 Header Files

When writing the operation and translation parts of a monitoring interface it is often necessary to refer to C entities exported by other modules. To enable the implementation of the monitoring interface to access these other interfaces it is necessary to include them in the monitoring interface description. Interfaces are included by simply naming the header files which contain them.

The string table monitoring interface uses some standard C library functions, C string functions and entities made available by the string table module. Consequently the interface also includes the following lines:

```
<stdlib.h>
<string.h>
"csm.h"
```

3.1.5 Non-standard types

By default, Dapto can handle the built-in types `int` and `str`. If you want to pass a value of some other type to an operation or receive such a value as an event parameter you need to tell the system about it. If you don't do anything then the values will be passed as the string "unknown".

Even if you do not add new operations or events involving non-standard types you probably want to provide proper monitoring support for them anyway. The reason is that other parts of the system may need to report values of these types to Noosa. Most notably, the attribute evaluator generates events whenever attributes are evaluated. If you want to be able to monitor attributes of non-standard types then you must add proper monitoring support for these types or the attribute values will be reported as "unknown".

The rest of this section explains what you need to do to monitor values of a non-standard type. It talks about the monitoring interface and associated support. The next section describes how you might go about displaying values in the Noosa transcript window for user browsing.

The following information is based on the monitoring support for environment values in the current Eli system. The environment module has the following monitoring interface containing a couple of events and an operation (see the file 'pkg/Name/envmod.dapto' in the Eli distribution).

```
aspect envmod;

"envmod.h"

event env_created* "An environment value has been created"
    (Environment env "The environment that was created",
     Environment parent "The parent environment (if any)");

event binding_made* "A binding has been made in an environment"
    (Environment env "The environment in which the binding was made",
     int idn "The identifier that was bound",
     DefTableKey key "The key to which the identifier was bound");

operation get_scope_info
    "Return the parent environment of an environment and its idn-key bindings"
    (Environment env "The environment to be searched") : str
{
    Scope s;

    DAPTO_RESULT_PTR (env->parent);
    for (s = env->relate; s != NoScope; s = s->nxt) {
        DAPTO_RESULT_INT (s->idn);
        DAPTO_RESULT_PTR (s->key);
    }
}

end;
```

As is conventional in a monitoring interface, the events are used to notify Noosa of important changes to the environment values as they occur. The operation is used to allow Noosa to get the complete contents of an environment. Providing both events and operations in this style is a good idea because the events allow fine-grained control via breakpoints and handlers while the operation can be used to implement value browsing.

Note that the operation implementation can use any C code it likes to determine the appropriate information and return it to Noosa. In this case we use the fields provided by the environment module to return the parent environment and all of the integer-key pairs.

Since `Environment` and `DefTableKey` values are passed as event and operation parameters we need to tell Dapto how to pass them. In the following we just talk about environment values. Support for definition table keys is similar.

When Dapto generates the event generation code for an event parameter of unknown type it attempts to use a macro of the form `DAPTO_RESULTx` where *x* is the name of the parameter type. Thus to get the value passed correctly you need to define this macro. Usually the definition is placed in the header file that defines the type itself. E.g., '`envmod.h`' contains the following definition.

```
#define DAPTO_RESULTEnvironment(e) DAPTO_RESULT_PTR (e)
```

which says that an environment value should be sent from the running program to Noosa as a pointer (since it is a pointer).

Similarly, to permit values of this type to be sent from Noosa to the running program (as operation parameters) you need to define a macro whose name is `DAPTO_ARGx`. For example, for environments we define the following macro.

```
#define DAPTO_ARGEnvironment(e)    DAPTO_ARG_PTR (e, Environment)
```

which says that it should be received as a pointer. In the definition of the macro, the second parameter is the type of the value. It is used to cast the received value to the appropriate type.

3.1.6 Browsing non-standard types

Once you have Noosa and the running program correctly passing values of a non-standard type back and forth, you usually want to see those values in the Noosa transcript. If the values are structured, you will also want to add browsing support for them.

Adding browsing support for a non-standard type involves writing Tcl code that will be invoked whenever a value of this type is browsed. The procedure can be automatically loaded into Noosa by placing its definition in a startup file (see Section 1.3 [User Initialisation], page 3). Alternatively, it can be placed in a file of type `tcl` and included in your specifications. At startup Noosa will load all files of this type.

The Noosa transcript is a general text display area, so you can use `n_say` to display whatever you like (it always displays at the end). As a special case if you display something of the form `t:v` where *t* is the name of a type which has browsing support, then the value *v* will also be browsable. In general it's a good idea to arrange for values to be prefixed by their type in this way even if no browsing support is currently available. The type provides a valuable clue to the user and if browsing support is added later it will available here without you having to do anything.

Here is a slightly simplified version of the Tcl support used by Eli to support browsing of environment values.

```
set n(Environment,desc) "Identifier scoping environment"

proc n_Environment_say {env} {
    n_say "Environment:0x[n_dectohex $env]"
}

proc n_Environment_open {text env} {
    n_say "$text"
    if {$env == 0} {
        n_say "\n  NoEnv\n"
    } else {
        set env [n_hextodec $env]
        set r [n_send get_scope_info $env]
        if {[lindex $r 0] != 0} {
            n_say " (parent: "
            n_Environment_say [lindex $r 0]
            n_say ")"
        }
        set r [lreplace $r 0 0]
        n_say "\n"
        set c 0
        foreach {i j} $r {
            n_say "  "
            n_say_val DefTableKey $j
            set s [n_send get_string $i]
            n_say " $s\n"
            incr c
        }
        if {$c == 0} {
            n_say "  No bindings\n"
        }
    }
}
```

The first `set` command sets a documentation string that will be used to display an information message at the bottom of the Noosa window whenever the user moves the mouse over a value of this type in the transcript window. In general, for a type *x* you need to set the array element `n(x,desc)` in the global scope.

The procedure `n_Environment_say` is used by Noosa to display values of this type. Since Environment values are pointers, the code displays them in hex to facilitate cross-referencing with values displayed by a source-level debugger. The Noosa library procedure `n_dectohex` is used to obtain the hexadecimal representation of the value. If `n_Environment_say` did not exist, values would be displayed in the style *t:v* where *t* is the type and *v* is the value in decimal.

The procedure `n_Environment_open` is invoked whenever the user clicks on a value of this type in the transcript window. In general, the procedure name must be `n_x_open` where *x* is the type name. The existence of this procedure is taken by Noosa as an indication that values of type *x* should be browsable. The procedure gets two parameters; the first is the complete text that the user clicked on (which includes the type name) and the second is the value part of that text. In this case the second parameter will be the environment value of interest.

The implementation of this procedure first displays the clicked-on text to identify the subsequent output because the browsable value may be a long distance from the bottom of the transcript where the output will be displayed. A null environment is displayed in a standard way to match the user's view of the module.

Non-null environments are converted by `n_hextodec` into decimal before being passed to the `get_scope_info` operation defined in the environment module monitoring interface (see Section 3.1.5 [Non-standard types], page 16). This operation gets the parent environment and the integer-key pairs as a Tcl list. The Noosa procedure `n_send` is used to invoke the operation with the environment value as the sole parameter.

When the `get_scope_info` operation returns, the `n_Environment_open` procedure goes on to display various information in the Noosa transcript window. Strings are displayed using `n_say`. The parent environment (if there is one) is displayed using `n_Environment_say` so that it is displayed in a style consistent with other environments.

All of the integer-key pairs in the environment are displayed. The routine `n_say_val` is used to display the keys. It is passed the type of the value and the value itself. `n_say_val` separates the decision about how to display keys from other code. `n_say_val` just dispatches to `n_DefTableKey_say` if it exists.

Note that we don't display the integers as-is, we use the `get_string` operation from the string storage module to convert them to strings which is generally more helpful. Note: arguably this is a bug since it's possible to use the environment module with integers that are not string table indexes.

3.2 Implementing Monitoring Interfaces

A type-'`dapto`' file defines the monitoring interface of a component. (See Section 3.6 [Dapto Grammar], page 24, for the syntax of the Dapto language.) The `dapto` program turns these interfaces into code that can be incorporated into a program that we want to be able to monitor. Dapto does two main things:

1. Generates a type-'`c`' file and a type-'`h`' file containing an implementation of the monitoring interface given by its input type-'`dapto`' file.

   The type-'`c`' file will contain routines to enable the monitoring system to invoke data operations and receive the results. The mechanisms by which this happens are beyond the scope of this manual.

   Also contained in the type-'`c`' file will be one function definition for each event type defined in the monitoring interface. For each event type X there will be a function `_dapto_`X that has parameters corresponding to the parameters of X. (See ⟨undefined⟩ [Monitoring support], page ⟨undefined⟩, for details on how to use this function.)

   The type-'`h`' file generated by dapto will contain the externally visible interface of the type-'`c`' file.
2. Generates a type-'`db`' file containing a *monitoring database* with information about the monitoring interface. This file is a TCL script that sets up data structures for use by the monitoring system. It is used to let the monitoring system know which aspects are provided by the monitoring interface and which events are contained in those aspects. See Section 3.3 [Database], page 21, for more information on how the database is used.

The names of the generated files depend on the name of the input file; '`csm.dapto`' will produce '`csm_dapto.c`', '`csm_dapto.h`' and '`csm_dapto.db`'.

3.3 Monitoring Database

A monitoring database is generated by Dapto from a monitoring interface description (see Section 3.2 [Implementing Interfaces], page 21). The concatenation of the monitoring databases for all of the components present in a program comprises the monitoring database for the program.

The monitoring database is simply a TCL file which, when loaded by Noosa, provides information about the aspects and events of the monitoring interface. For example, the monitoring database for the string table monitoring interface (see Section 3.1 [Monitoring Interfaces], page 13) yields the following database (reformatted slightly):

```
lappend n(aspects) string
lappend n(events) \
    [list string_stored "Storage of a new string in the string table" \
        { index "Index of new string" string "New string" } 1]
```

The global TCL lists `n(aspects)` and `n(events)` are used to store the database information. `n(aspects)` contains a list of the all of the aspect names contained in the program. `n(events)` is a list of lists; each sub-list contains the name and documentation strings for a single event type and its parameters, plus a flag which is 1 if the event is visible to the user and 0 otherwise.

3.4 Adding Monitoring Support To A Component

Once you have a monitoring interface implementation for a component you must add monitoring support to the component itself. This support consists entirely of calls to the event generation routines for any events you have in your interface (see ⟨undefined⟩ [Monitoring interfaces], page ⟨undefined⟩ and see ⟨undefined⟩ [Implementing interfaces], page ⟨undefined⟩). If you have no events in your interface, the code of the component does not need to be changed.

Adding event generation to a component is a matter of adding calls to event generation routines at the appropriate places. The details of this will depend on the component, but the idea is to insert the calls at places where the action which the event represents can be said to have taken place. Any necessary event parameters should be passed to the event generation routine.

To enable a monitoring-free version of the component to be easily produced, the convention is that all additions purely for the purpose of monitoring be conditionalised by

```
#ifdef MONITOR
...
#endif
```

The following examples are based on monitoring support for the Eli string table component. The component must be modified to include the C interface to the monitoring interface:

```
#ifdef MONITOR
#include "csm_dapto.h"
#endif
```

Then we must identify places in the code where `string_stored` events must be generated. There is only one of these, at the end of the routine `stostr`, so we add the following code to generate the event with the appropriate parameter values:

```
#ifdef MONITOR
    _dapto_string_stored (numstr, string[numstr]);
#endif
```

When the component is compiled by Eli with the `-DMONITOR` compiler option (implied by the `+monitor` parameter), this monitoring support will be included.

3.5 Supporting Profiling

Noosa contains support for two kinds of profiles (see ⟨undefined⟩ [Frequency profiles], page ⟨undefined⟩ and see ⟨undefined⟩ [Time profiles], page ⟨undefined⟩). To support profiling of a component it is necessary to add extra event generation to a component. It is necessary to generate an `enter` event whenever execution enters the code of the component and a `leave` event whenever execution leaves the code of the component. These events have the following signatures:

```
event enter "Enter a program component"
    (str name "Name of component");
event leave "Leave a program component"
    (str name "Name of component");
```

For the string table component we would add the following code to the beginning of each string table routine:

```
#ifdef MONITOR
  _dapto_enter ("string");
#endif
```

and the following code at each exit point of each string table routine:

```
#ifdef MONITOR
  _dapto_leave ("string");
#endif
```

The event parameter ("string" in this case) is used by the profile monitoring code to identify the component.

3.6 Dapto Grammar

The following context-free grammar defines the syntax of the Dapto language. *ident* is an identifier in the C style. Identifier definitions are required to be unique within a specification and within event and operation blocks. *str* and *bstr* are strings delimited by double quotes and angled brackets, respectively. *text* is arbitrary text delimited by braces.

spec: aspects.

aspects: aspect˙stmt / aspects aspect˙stmt.
aspect˙stmt: ‘`aspect`’ iddef ‘`;`’ sigs ‘`end`’ ‘`;`’.

sigs: sig / sigs sig.
sig: event˙sig / operation˙sig / str / bstr.

event˙sig: ‘`event`’ iddef export str event˙block ‘`;`’.
event˙block: ‘`(`’ optattrs ‘`)`’.
export: ‘`*`’ / /* empty */.
optattrs: /* empty */ / attrs.
attrs: attr / attrs ‘`,`’ attr.
attr: typeid iddef str.

operation˙sig: ‘`operation`’ iddef str operation˙block text /
‘`operation`’ iddef str operation˙block ‘`:`’ typeid text.
operation˙block: ‘`(`’ optparams ‘`)`’.
optparams: /* empty */ / params.
params: param / params ‘`,`’ param.
param: typeid iddef str.

iddef: ident.
iduse: ident.
typeid: ident.

Index

S

T

U

V

W

X