Execution Monitoring Reference

$Revision: 1.10 $

A. M. Sloane

Department of Computer Science
James Cook University
Townsville, QLD, 4811
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` and `CmdHelp` commands.)

1 Using Noosa

This chapter describes how to invoke the Noosa system, run your program under the control of Noosa and how to use Noosa to find out information about the program execution.

1.1 Invoking Noosa

Noosa is invoked from within Eli using any of the `:mon`, `:mongdb`, or `mondbx` 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 the system is invoked. Invoking Noosa will produce a window containing an input text window at the top, a command area in the middle and a transcript window at the bottom.

To execute a command within Noosa, just click the middle mouse button on its name. When you are first starting to use Noosa it is best to use the Help command to find out about the user interface. It will direct you to the CmdHelp command which lets you find out about individual commands.

The most important command to know at this stage is Quit 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 The Noosa Mouse

The three mouse buttons are used extensively by Noosa to provide a convenient way to display information and browse through it. Apart from the obvious use of the mouse to scroll around the Noosa text windows, the buttons have a number of special functions.

The left button operates in a standard fashion to select text. Press and drag selects the dragged text. A double-click selects a word and a triple click selects a line.

The middle button is used to execute commands. A command is executed simply by clicking the middle button on its name. This name may be displayed anywhere; it doesn't have to appear in the centre command window, although that is the usual case.

Some commands are toggles. In this case executing the command toggles an piece of internal Noosa state (e.g., to turn tracing on and off). When a toggle command is off its name will be displayed with a line through it. Toggling the command will remove the line.

The right button is used for searching. The simplest usage searches for text by clicking the right button on a word (non-white space). The following occurrence of that text in the same window (if any) will be highlighted.

The right button also searches for coordinates. Clicking on a coordinate (or range of coordinates) will cause the text corresponding to the coordinate (or range) to be highlighted in the input text window (see Section 2.1 [Input Text Display], page 7).

It is possible to override the coordinate searching behaviour by first selecting the text you want to search for with the left button. If the right button is clicked inside an existing selecting then that selection is searched for (not the current word or coordinate). Thus to find the next occurrence of a coordinate in the window you must select it with the left button and then click inside that selection with the right button.

The default Noosa bindings assume you have a three button mouse. In some environments (e.g., on laptop machines) one button mice are more common. For this reason, Noosa supports the following mouse aliases. Holding down the Control key while using the left button is equivalent to using the middle button. Also, holding down the Alt key while using the left button is equivalent to using the right button. These modifiers work even when you have a three button mouse.

1.3 Controlling Your Program

The text entry area near the middle of the Noosa window contains the name of your program and any 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.

Normally only those commands which are applicable will be shown in the command area. Commands will come and go as monitoring proceeds. A number of commands are used to control the execution of your program.

The `Run` command causes your program to execute from the beginning with the specified program arguments. If there is already an instance of the program running it is killed first.

The `Cont` button causes your program to continue execution after stopping. The program will always stop at the end of its execution. Whether it stops at any other times will depend on whether you have set any breakpoints or not (see Section 2.7 [Breakpoints], page 10).

The `Kill` button causes the currently running or stopped instance of your program to be killed.

1.4 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 undocumented. Future versions of this manual will describe in detail how Noosa can be configured using a '`.noosarc`' file.

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 (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).

Clicking the right mouse button on a displayed coordinate causes Noosa to select 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 Coord command.

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 String command. Each string in the table will be displayed with its index. The String command can also 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 let 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) and executing the Token command 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 type used internally by the analyser, length in characters, the intrinsic attribute value of the token, and the input text matched by the token (lexeme) (see Lexical Analysis). A future version will also display the token name as specified in your type-‘`gla`’ specification.

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 lets you examine the phrase structure that is recognised by your parser. Selecting an input text coordinate 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). Noosa cannot currently be used with the COLA `+direct` parameter (see Cola Option Tool).

2.6 Abstract Tree

The Attributes command enables you to get a display of the abstract tree that is to be used by your program. If Attributes is enabled Noosa will produce a new window when the program is run and it creates the abstract tree. The new window will contain a graphical depiction of the abstract tree. Each node in the tree will contain the grammar symbol represented by that node and the name of the rule that derives the children.

The tree can be browsed by scrolling around the display using the scrollbars or dragging with the middle button down. Also, you can click on a node with the right button to get the input text corresponding to the node highlighted in the main Noosa window. The node coordinates and production text will also be displayed in the transcript window.

Correspondences in the other direction can be examined using the Node command (in the abstract tree display window). Select an input text coordinate and execute Node. The most specific (closest to the leaves) abstract tree node that derives that coordinate will then be highlighted.

2.7 Breakpoints

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 *attributes* 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 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, disabled (and re-enabled) or deleted. (See the Help command in the Handlers dialog for more information.)

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 attributes as Tcl variables. To cause execution to stop as the result of handler execution have the handler call the Tcl break command at the top level (i.e., not inside a loop).

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"} {
    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. The builtin Noosa command `n_send` must be used to send the `n_say` command to Noosa itself; otherwise, the `n_say` is executed in the context of your program.

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` attributes available.

```
if {$linebeg == 3} {
    n_send 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 Freq command. By default, they are disabled which is indicated by a line through the Freq command name. Also, the FreqZero 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 command. By default, they are disabled which is indicated by a line through the Time command name. Also, the TimeZero 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 Trace command provides this capability. When tracing is enabled Noosa will display the event type and attributes of every event generated by the program until it stops.

The TraceExp command allows subsets of events to be selected using a regular expression. First, enter the regular expression in a window somewhere (in the transcript window is fine) and select it. Executing the TraceExp command at that point will cause subsequent tracing to only display an event 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 3.5 [Dapto Grammar], page 19, 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 3.2.1 [Database], page 16, 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 attributes. 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 generic monitors, 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:

`RESULT(char *s)`

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

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

`RESULTPTR(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.

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

Use of `RESULT` and `RESULTINT` 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 `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
{
    char *s = string[index];
    if (s == (char *) 0)
        RESULT ("*** No string at this index ***");
    else
        RESULT (s);
}
```

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"
```

3.2 Implementing Monitoring Interfaces

A type-'`dapto`' file defines the monitoring interface of a component. (See Section 3.5 [Dapto Grammar], page 19, 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 `generate_`*X* that has parameters corresponding to the attributes 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 3.2.1 [Database], page 16, 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`'.

3.2.1 Monitoring Database

A monitoring database is generated by Dapto from a monitoring interface description (see Section 3.2 [Implementing Interfaces], page 16). The concatenatation 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 attributes, plus a flag which is 1 if the event is visible to the user.

3.3 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 attributes 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 an:

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

pair.

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 attribute values:

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

When the component is compiled by Eli with the `-DMONITOR` compiler option, this monitoring support will be included.

3.4 Supporting Profiling

Noosa contains two generic monitors for producing profiles (see ⟨undefined⟩ [Frequency profiles], page ⟨undefined⟩ and see ⟨undefined⟩ [Time profiles], page ⟨undefined⟩). To support these monitors 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
  generate_enter ("string");
#endif
```

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

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

The event attribute ("string" in this case) is used by the profile monitors to identify the component.

3.5 Dapto Grammar

The following context-free grammar defines the syntax of the Dapto language. Identifier definitions are required to be unique within a specification and within event and operation blocks.

spec: *aspects*.

aspects: *aspect_stmt* / *aspects aspect_stmt*.
aspect_stmt: '`aspect`' *iddef* '`;`' *sigs* '`end`' '`;`'.

sigs: *sig* / *sigs sig*.
sig: *event_sig* / *operation_sig* / *str* / *translation_sig*.

event_sig: '`event`' *iddef export str event_block* '`;`'.
event_block: '`(`' *optattrs* '`)`'.
export: '`*`' / /* empty */.
optattrs: /* empty */ / *attrs*.
attrs: *attr* / *attrs* '`,`' *attr*.
attr: *iduse iddef str*.

operation_sig: '`operation`' *iddef str operation_block text* /
  '`operation`' *iddef str operation_block* '`:`' *iduse text*.
operation_block: '`(`' *optparams* '`)`'.
optparams: /* empty */ / *params*.
params: *param* / *params* '`,`' *param*.
param: *iduse iddef str*.

translation_sig: '`translation`' *iddef iduse str text*.

iddef: *ident*.
iduse: *ident*.

Index