Event Monitoring and Abstraction Tools

Peter C. Bates

COINS Technical Report 89–17
March 14, 1989

Laboratory for Cooperative Distributed and Parallel Computing
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This research supported in part by the National Science Foundation under grants MCS-8306327, DCR-8318776, and DCR-8500332.

Event Monitoring and Abstraction Tools

Peter C. Bates

Laboratory for Cooperative Distributed and Parallel Computing
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

A collection of tools is described that allows the user of ***Event Based Behavioral Abstraction*** to monitor and control an executing system. The use of the tools and how they are interconnected is the focus here; rather than the details of their operation. Also described here is the Lisp-like extension language that serves both as an inter-tool communications protocol and a programming language that permits users to create sophisticated responses to monitoring activities.

Contents

List of Figures

1. Overview

Event Monitoring and Abstraction is the main operational activity of the *Event Based Behavioral Abstraction* (*EBBA*) debugging paradigm. A collection of tools has been created that allows a user to employ *EBBA* in order to understand a system with a view to correcting its behavior. The toolset (figure 1) contains components for creating and maintaining viewpoints and models (Model Builder and Librarian), abstracting behavior models from actual behavior (Event Recognizer), analyzing the status of abstraction tasks (Behavior Monitor), and a collection of user interfaces (User) and graphical information gathering displays.

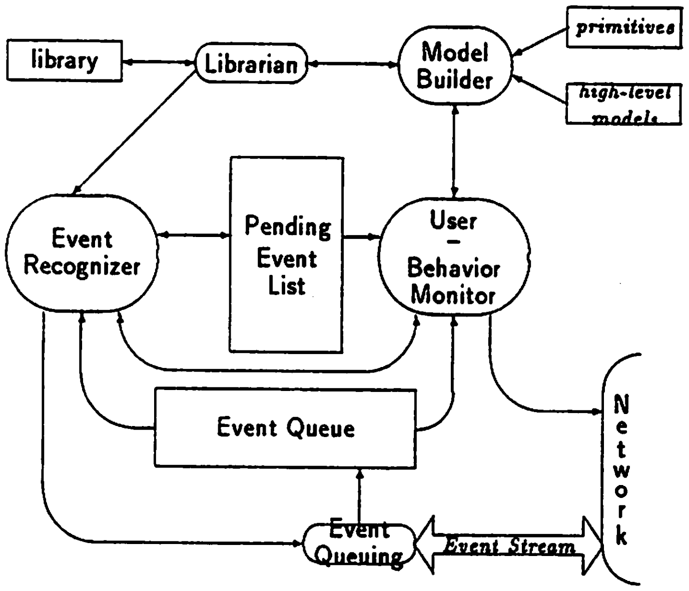

Figure 1: Basic toolset components

This report describes the use of toolset components that are involved in the creation, gathering, and abstraction of events. This description encompasses individual toolset components as well as their integration as a distributed program. Also described here is the Lisp-like extension language, *elll*, that serves to bind together all toolset components. The *elll* is a communication protocol and interpreted language that permits toolset components to be structured as a message-based system and provides a mechanism for users to extend the basic functionality provided by individual components.

The remainder of this section defines what events are and provides an overview of the interaction of toolset components involved in the event abstraction process. Following sections provide a more detailed description of the components and how they are used together to provide a functional system. A significant portion of these sections details *elll* messages and what their interpretation will effect. The last part of section 2 details how to add new basic functionality to the *elll*.

1.1 First, the Glue

Event exchange is the medium used for all system understanding in *EBBA*. Events are used to represent some significant system transition or interaction of system components. An event is recorded by a tuple that consists of an event class name and a list of attribute values. The class places an event into a general catagory while the attribute values serve to characterize and differentiate specific occurrences of events from the class. All of the events in a class have the same number, types, and ordering of their attributes. The *shape* of an event–its class together with its attribute arrangement–is derived from its *EDL* modelling language description [1].

Every event class can be represented with a template having the general form:

(event-class a_1 a_2... time location).

The attributes, $a_1, a_2,$...correspond to targets ~~of value binding expressions described in with~~ clauses of the *EDL* modelling language description. Each event class has a distinct template determined by its shape. The *time* and *location* attributes, characteristic of all events, are implicitly the last two attributes of any event tuple. *time* and *location* represent the system time the event instance was created and where it originates in the system, respectively.

As a system executes, it creates *instances* of its event classes to evidence its behavior. The class of an event instance is determined by where in the executing system the instance is detected. In the current grand scheme of things, event instances are created by annotations (probes) added to the program text. All instances of a single event class are (probably) created by the same instrumentation probe in the monitored system. The values bound to the attribute fields of the instance are determined from the system state that is available when the instance is recorded. The time attribute for any event is a somewhat slippery object. Since the time assigned to an instance is taken while the instance is being recorded, the instance time actually follows the time the behavior it represents is deemed to have "happened", but before the next bit of behavior has begun. One further complication to relying on time is that quantum rundown or other preemptive system behavior during event instance generation could cause events that have actually occurred in one order to acquire timestamps that appear to change this order.

For example the concurrent Mandelbrot Set Generator [2] creates a characteristic set of primitive events, one of which has the following *EDL* description:

- *worker announces its intention to fill a patch*
`event` *e_TakePatch* `is`

```
   primitive "e_TakePatch"
with
   id :  integer;
   patchidx :  integer
end
```

The event tuple described by the *e_TakePatch* description has the template:

(e_TakePatch *id patchidx time location*)

e_TakePatch is the event class. Instances of the class are distinguished by the values bound to the *id*, *patchidx*, *time*, and *location* attributes. Some instances of this class might be

```
(e_TakePatch 3 26 0:00.17 "min.3")
(e_TakePatch 2 47 0:01.03 "min.2")
```

The *Event Stream* (Figure 1) is a merging of the event instances from all of the event generators contained in a particular set of cooperating components. The instances inserted into the stream by individual components are interleaved arbitrarily with those generated by other components. Nothing can be determined regarding the temporal relations of events created on different processors. The clocks that time stamp individual events on different nodes are at best close and the statistical access to the physical communication medium confuses internode relations further.

The event Librarian (Figure 1) is responsible for maintaining viewpoints on a system. A *viewpoint* is defined in terms of a set of primitive events. Within a viewpoint, high-level behavior models may be defined in terms of the basic primitive event set. The librarian contains facilities for adding new primitive events to the current viewpoint, and for merging many libraries into a single viewpoint. An interface to the Model Builder helps tool users to add high-level models to the viewpoint maintained by a library.

1.2 Event Sources/Event Sinks

The tools that are directly involved in event instance manipulation and exchange (to differentiate from model creation and maintenance) can be classified as event *sources* and event *sinks* (see figure 2). An event source is a program that has been instrumented to generate low-level (primitive) events for consideration by a sink. An event source is constructed by annotating a program that is to be monitored with calls to event instance formatting and transmission routines. This is currently managed manually, but work is in progress that would automate this activity. Other techniques might also be employed, but the net effect is that sources somehow obtain event instance records and send them to one or more event sinks.

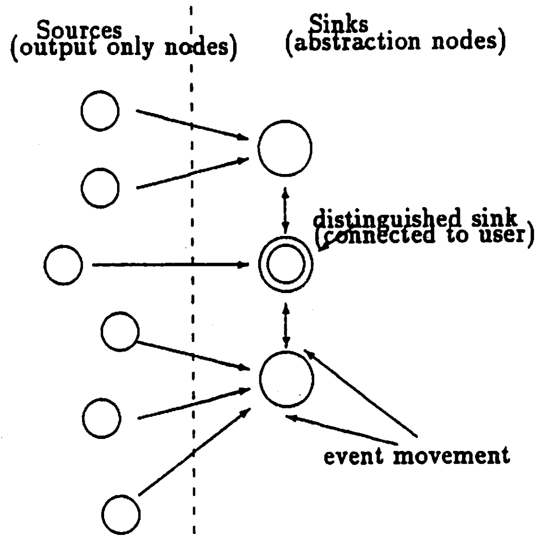

Figure 2: Event Source/Sink relations

An event sink is a component that is capable of receiving and possibly abstracting events. Some event sinks simply accept event messages and present them or some characterization to a tool user. An event sink that is involved in event abstraction is also a potential event source that sends representative instances of the events it has observed to other event sinks.

2. Extension Lisp-Like Language and Inter-Debugger Protocol

Toolset components exchange messages to describe their status and to request services of other components. These messages are coded as S-expressions using the syntax of the extension language, *elll*. All communication among components comprising a toolset is by exchange of *elll* expressions. Event instances resemble *elll* expressions. Requests to components for action are *elll* expressions. Information bearing messages passed among components are *elll* expressions.

elll is normally bound to a toolset component and used as a service by that component. The basic functionality of *elll* is determined by its wired-in functions. The basic functionality can be changed by adding more wired-in functions to the basic *elll* interpreter or by wired-in functions provided by the component that is bound to the interpreter. The basic functionality of these wired-in functions can be extended by defining programs and functions using the *elll* function definition routines.

When an expression is received by a component it is passed to the *elll* interpreter where it is parsed and interpreted to obtain a value. Interpretation of an expression is carried out in the context defined by the message receiver. This carries the implicit possibility that different receivers may use the same expression for different purposes. This simplifies the message passing architecture and facilitates component interchangability. Users can enter *elll* expressions to directly control the activity of a component if the component allows direct entry of messages from type-in windows. Likewise, user interfaces, perhaps based on mouse inputs and graphical display output, format messages as *elll* expressions to be executed by the component they represent.

elll expressions are parenthesized lists of symbols, values, and parenthesized lists. This gives *elll* a Lisp-like syntax but interpretation of individual expression arguments has certain context-sensitive aspects. All symbols must be declared (defined) as either a function, variable, or event class before any attempt is made to evaluate the symbol to obtain a value.

The extension language makes a clear distinction between names that represent executable functions and names that are value bearing variables. This is directly reflected in the structure of an *elll* expression

$$(\textit{function-name}\ arg_1\ arg_2 \ldots arg_n)$$

function-name must be bound to some executable procedure before evaluation of the expression is attempted. The list arg_1, $arg_2 \ldots$ may be constant values, variable names, or parenthesized function invocations.

Execution of *elll* programs and functions has two phases: a tree building phase where the text of an expression is parsed and translated into an internal form; followed by an execution phase that evaluates the program to return a value. *elll* programs are represented as a tree of object structures, each of which represents a function invocation or atomic value. Eventually all program trees arrive at wired-in procedures that actually perform functions on the values defined by the current execution context. Whenever a function executes it fills in a global "register" with the most recently obtained value, and returns an interpreter status value. These are accessed by component routines through wired-in *elll* functions and global variables.

2.1 Lexical and Syntactic Elements

elll expressions (or statements) are composed from a small number of lexical elements (tokens). An expression is free-form, any number of space characters may appear between tokens. Space characters are the usual blank and tab plus all of the text formatting characters such as newline, form feed, vertical tab, etc.. Structure constructors can result in vectors of atomic typed data, event tuples, and programs. However, general record structures are not supported as a type. One effect of this restriction is that there is no real capability for the normal Lisp program-as-data view.

2.1.1 Comments

A comment may be inserted anywhere a space character is permitted. Comment text begins with a "`;`" (semicolon) character and ends with the next newline character (`\n` for those C hackers out there). Thus, comments do not span lines. For example

```
; run-model
; Function to get a model running without questions
(defun run-model (model tag queuepos)
         ; create the recognition context
      (recognize tag model)
         ; add standard responses
      (add-rcb tag 'RecyclePendingEvent nil)
      ...
      )
```

2.1.2 Parenthesized List

A parenthesized list introduces a function invocation. The list begins with an open parenthesis character "`(`" and ends with a matching right parenthesis "`)`". The general form of a parenthesized list (function invocation) is:

(*function-name* $item_1 \ldots item_n$)

function-name must name a function symbol for either a wired-in or an *elll*-defined function. The function is applied with the remainder of the list as actual procedure arguments. Each *item* may be any of the *elll* lexical elements, subject to the interpretation constraints of the named function. In the above example, `defun`, `recognize`, and `add-rcb` are executable function names.

2.1.3 String

A string is a contiguous sequence of characters enclosed between pairs of """ marks. Any printable character may be embedded in a string. Single " characters may be inserted into a string by preceding the " character with an escape character (\), i.e. a string

```
"...x\"y..."
```

is represented internally as

```
...x"y....
```

Non-printing characters may be embedded in strings using the `\000` form where `\000` is the octal value of the desired character. Since strings are stored internally as nul (character 0) terminated sequences of characters, inserting the character `\0` might terminate the string prematurely. Some special forms are available for commonly used special characters:

`\\` -inserts a `\` character

`\n` -inserts a newline (`ctrl-j`) character

`\b` -backspace (`ctrl-h`) character

`\r` -carriage return (`ctrl-m`) character

`\t` -tabbing (`ctrl-i`) character

`\e` -escape (`esc`) character

A string must terminate before the occurrence of the next newline character in the *elll* text. There is currently no simple mechanism for entering multiline strings. Also, comment text (beginning with a ; character) cannot be inserted into a string.

2.1.4 Numbers

Numbers represent unsigned or negatively signed whole decimal values, floating point numbers, or bitstrings specified using any of hexadecimal, octal, or binary radixes. Also possible are numbers specified as single characters. Positive, whole integer values are entered as a string of contiguous decimal digits. Negative decimal numbers may be entered with a minus sign (- character) preceding the digit string. (There is no explicit +*number* form available.) Numbers entered as single characters are specified by enclosing the character between "'" marks, e.g. `'a'` or `'$'`.

Floating point numbers are indicated by their containing an explicit fractional part or scaling exponent part, or both. As with whole integers, an optional '-' (minus sign) preceding the string indicates a negative valued number. The following are acceptable forms for floating point numbers:

```
dd.dd
dd.ddEdd or dd.ddE+dd
dd.ddE-dd
ddEdd or ddE+dd
ddE-dd
```

Where dd is a string of contiguous decimal (0-9) digits.

Bitstrings are entered as unsigned radix-specific strings of characters. The general form is

`0<`*radix*`>DDDD`

where <*radix*> is one of `H`, `O`, or `B` (or `h`, `o`, `b`) indicating hexadecimal, octal, or binary conversion radix respectively. The value string `DDDD` that follows the radix specification must contain only valid characters for the indicated radix

hexadecimal: `0-9, a, b, c, d, e, f (A, B, C, D, E, F)`

octal: `0-7`

binary: `0,1`

Thus, some acceptable radix specific bitstrings might be

```
0x4e7      0xf3A
0o62243    0O77737
0B011011   0b10101
```

2.1.5 Names (Symbols)

Any other contiguous set of characters is accumulated into a symbol or name string. A name string terminates at the first space character, parenthesis (open or close), or comment character (`;`). As name symbols are parsed they are looked up in the name table. If previously defined, a reference to the name is placed into the parsed expression. If not defined previously a new name table entry is created and this reference is placed into the parsed expression. Until a name is explicitly (using *declare-global*) or implicitly declared (function or lambda block parameter), or assigned a value (using *(setq ...)*), it may not be referenced to return a value.

2.2 Wired-in Names (functions and variables), common across *elll* interpreters

Wired-in functions are the routines that are linked to the implementation of the *elll* interpreter in order to provide the fundamental operations defined by *elll* and the components that interpret functions. All components that contain *elll* interpreters provide a common set of functions, described below. Each component might also provide some component-specific wired-in functions. The wired-in functions group into value-returning arithmetic and logical operations, control functions, declarative and structure constructors, and miscellaneous environmental access operations.

2.2.1 Wired-in Variables

Wired-in variables provide a way for *elll* routines and function calls to access values maintained by program components. Wired-in variables also represent commonly used values. The variable name is created by the component that maintains the value, via a call to the `elllDefineVariable()` interpreter function. Available wired-in variables are:

`nil` - supplies null string, empty list, or number 0 as needed.

`InterpreterStatus` - contains the most recently set interpreter status value.

2.2.2 Arithmetic Functions

Arithmetic functions accept a list of numeric values, compute their function, and return the result. The value types can be integers specified as whole numbers or bit strings, floating point values, or text strings that contain a valid number. The result type will be integer if all the numeric values are integers or floating point if any of the operand values are floating point. Available arithmetic operators are:

(+ a_1 $a_2 \dots a_n$) - adds numbers $a_1 + a_2 + \dots + a_n$ and returns the sum,

(- a_1 $a_2 \dots a_n$) - subtracts numbers $a_1 - a_2 - \dots - a_n$ and returns the resultant difference,

(* a_1 $a_2 \dots a_n$) - multiplies $a_1 * a_2 * \dots * a_n$ and returns product

(% a_1 $a_2 \dots a_n$) - returns remainder of integer division, a_1 modulo a_2 modulo $\dots a_n$

(/ a_1 $a_2 \dots a_n$) - returns the value obtained from a_1 divided by a_2 divided by $\dots a_n$

2.2.3 Bitwise Numeric Functions

The bitwise logical functions perform their operation on 32 bit operands. Operands that are string types representing valid whole numbers are converted to 32 bit numbers before use. The result value from all bitwise logical operations is a 32 bit integer.

(& a_1 $a_2 \dots a_n$) - This function evaluates each a_i in turn, performing the bitwise and of the returned a_i value and the previous result. If the result becomes zero at some a_i, no more of the $a_{i+1} \dots a_n$ are evaluated, otherwise the final result is the bitwise "and" of $a_1 \dots a_n$

(| a_1 $a_2 \dots a_n$) - Evaluate $a_1 \dots a_n$ in turn and return the bitwise "inclusive-or" of $a_1 \dots a_n$. All of $a_1 \dots a_n$ are evaluated.

(^ a_1 $a_2 \dots a_n$) - Evaluate each of $a_1 \dots a_n$ and return the bitwise "exclusive-or" of $a_1 \dots a_n$

($\sim$ a_1) - bitwise complement of a_1

(<< a_1 a_2) - shift a_1 left by a_2 bits, zero filled

(>> a_1 a_2) - shift a_1 right by a_2 bits, zero filled

2.2.4 String Manipulation Functions

The string functions manipulate arbitrary length character strings of 8-bit characters. The nul character (\0) is considered to end a character string. In functions concerned with the position or index of a character in a string, the index or position of the first character is one (1).

(concat *str*$_1$ *str*$_2$...*str*$_n$) - returns a string that results from appending strings *str*$_2$ to *str*$_1$, *str*$_3$ to that, *str*$_4$ to that, etc. .

(substr *str pos n*) - copies *n* characters from *str* with first at *pos*, and returns this result. If the value of *pos* is negative, the source string is indexed from its end. If the effective starting position is after the end or before the beginning ($|pos| > length(str)$), the result is a null string (after) or a substring beginning at the first character position, (before).

(length *str*) - counts the characters in *str*, not including the nul-terminator character.

2.2.5 Relational Functions

The relational operators return an integer valued false (0) or true (1) value depending on the relation of their operands. All of the relational operators operate on integer, double length floating point, or string values. Implicit type conversion takes place to the stronger of the operands, where the ordering is double, integer, string, with double the strongest. A non-zero relational expression operand value is considered "true"; zero-valued expressions are "false".

(< a_1 a_2) - true if a_1 less than a_2

(<= a_1 a_2) - true if a_1 less than or equal a_2

(>= a_1 a_2) - true if a_1 greater than or equal a_2

(> a_1 a_2) - true if a_1 greater than a_2

(= a_1 a_2) - true if a_1 equal to a_2

(!= a_1 a_2) - true if a_1 not equal to a_2

(! a_1) - true if a_1 is false (zero valued), false if a_1 is true (non-zero).

(&& a_1 $a_2 \ldots a_n$) - logical "and" connective, true if all of a_1 and a_2 and $\ldots a_n$ are true. False if any of $a_1 \ldots a_n$ is false. This function short circuits. Each a_i is executed in turn until either all return true or the current a_i returns false.

(|| a_1 $a_2 \ldots a_n$) - logical "or" connective, true if any $a_1 \ldots a_n$ are true. This function short circuits. Each a_i is evaluated until the current a_i returns a true value.

(^^ a_1 a_2) - logical "exclusive-or", true if a_1 or a_2 is true but not if a_1 and a_2 are true.

2.2.6 Declarative Functions

These functions declare a name to be one of the acceptable kinds, thus permitting its use in an interpretation context. In general, there are only variable names and function names. Use of a variable name invokes a typed internal function that accesses the value currently bound to that name. Name symbols are also implicitly declared as variables if they appear in either the formal parameter list of an extension function or as a local name in a (progn ...) scoping construct.

Function names may be bound to *elll* interpreter-defined functions (wired-in to *elll*) or to user defined extension functions. Although a function name may be the same as a variable name, defining one kind does not imply definition or usability as the other (event names are an exception). Wired-in functions and event names may *not* be redefined.

Event names may be used in either a variable or function context. When used as a variable an event name returns its internal library-defined identifier. When used as a function name the remaining items in its expression are evaluated as attributes and an event tuple is returned.

(declare-event $e_1 \ldots e_n$)

Declares that symbols $e_1 \ldots e_n$ are to be treated as events. Each e_i has its definition searched for in the currently open event library and its description is read (but not its recognizer) into the event definition cache. An entry is made for the event class in both the global variable table and the function table. Subsequent use of the event name in a variable context returns the event library internal identifier. If invoked as a function, the actual parameters must agree in number and kind with the formal argument list from the event definition.

Global variables maintained by *elll* routines must be declared before they are used. These variables are given values using the `(setq ...)` *elll* function.

(declare-global $v_1 \ldots v_n$)

creates persistent global variable names $v_1 \ldots v_n$ with initial values 0 or `NULL`. These names are accessible to *elll* expressions to serve as operand values. Variables (or other value bearing items) contained in a component can be made visible to extension language routines by binding a global name in the *elll* domain to the storage holding the value.

The most powerful use of the extension language comes from the ability to define functions and routines that add functionality not wired into the protocol. The form

(defun *fn* (a_1 $de_1 \ldots a_c$ de_c) $lv_1 \ldots lv_m$ exp_1 $\ldots exp_n$)

declares a function, *fn*, with c arguments $a_1 \ldots a_c$. Each of the $lv_1 \ldots lv_m$ are local variables given empty bindings when the function is invoked. When the function is called with actual parameter arguments $p_1 \ldots p_k$ each actual parameter p_i is bound to the corresponding formal parameter a_i. If the number of actual parameters is greater than c, extra actual parameters are discarded. If the actual parameter count is less than c, each a_i that has no matching p_i is assigned the value resulting from the evaluation of its corresponding default expression, de_i. The de_i for formal parameter a_i is optional for the function definition. If no de_i is specified, and no actual parameter is available, an empty variable binding is attached to a_i when the function is entered. Execution of each exp_i takes place in the context of the global variable environment, the variables defined by formal parameters, and local variables.

2.2.7 Control Operations

The control operations implement simple conditional iterative and selection constructs. Each control function returns a value in the result register and returns an evaluator status.

(progn $var_1 \ldots var_m$ $exp_1 \ldots exp_n$) - creates empty local bindings for the list of variables $var_1 \ldots var_n$ and then evaluates each expression $exp_1 \ldots exp_n$ in turn. The returned result is the value of the last expression exp_i evaluated. Following execution of all the exp_i's, the bindings assigned to $var_1 \ldots var_n$ are released. Use this construct as a compound statement with local variables might be used.

(if be_1 $true_1 ... be_n$ $true_n$ *default*) - evaluate each selection expression (be_i) until the result of be_i is non-zero; then execute the corresponding $true_i$. Zero result from all be_i evaluations will cause the *default* statement to be evaluated. The result is the value returned by the executed statement. If no default is supplied, the if form simply exits. The status returned is STATUS_Success if no errors occurred during evaluation; STATUS_Failure otherwise.

(while *be* $exp_1 \ldots exp_n$) - The conditional expression *be* is evaluated. If evaluation returns a non-zero result, each expression $exp_1 \ldots exp_n$ is executed. The test-execute loop is repeated until *be* returns a zero value. The result is the value returned by the last exp_i to execute.

2.2.8 Environment Access Functions

(getenv *environment-variable*) - returns the string associated with the argument environment variable. If the environment variable is not defined, nil is returned.

(gethostname) - returns the host name associated with the local node.

(current-time) - returns a 26 character string containing the local day, time, and date. The form is the following:

```
Thu Mar 12 09:28:09 1987
```

(time-stamp) - creates a time value by accessing the current local system time, returns a time-typed expression value in the *EBBA* time format.

(location-stamp) - returns as a string the component location for events created in this process. The location string is set by a call to the libebba function, ebbaSetOrigin().

2.3 Adding New Wired-in Operations

Wired-in operations are useful for complex functions exported by a component or in situations where some limited capabilities of *elll* must be overcome. Adding a new wired-in procedure or function is not hard, just involved. Wired-in variables are a bit simpler. A wired-in procedure must be bound to the image that the basic *elll* interpreter is part of. There is currently no way to dynamically load executable procedures so the wired-in procedure must be compiled and linked when the component is built. A newly added wired-in procedure must be made known to the interpreter at runtime so that it's name is accessible in the *elll* interpreter domain.

2.3.1 Execution Environment of Wired Operations

elll programs are represented as a tree of `ProgramNode` structures. Each tree node represents a function invocation or an atomic value. Each `ProgramNode` has an attached structure that describes how to execute the node to obtain a value. A name-procedure binding structure implements the actual mechanism for obtaining values in the extension language. For function calling nodes the program node holds pointers to the actual argument list for the function invocation and a reference to the code that executes the function.

An executable procedure can currently be one of two types described by a name-procedure binding descriptor:

`ProcBound` - a wired-in procedure written in the implementation language (C, or compatible) and bound to the *elll* interpreter,

`LispBound` - a user defined function (program) written in the extension language in terms of wired-in and other lisp functions.

Bound procedures of `ProcBound` type are declared by the component that supports their functionality by calling `elllDefineProc()`. Arguments to `elllDefineProc()` include the name of the procedure to be bound, a reference to the executable procedure, and the type of binding the name-procedure pair represents.

`elllDefineProc()` searches for an already defined version of the name; if not found a new name-procedure binding structure is created. A reference to the name is added to the procedure name table. If the search through the names table finds a previous definition for the name the new procedure body binding will replace the old one only if the old binding is not already `ProcBound` (wired-in). `LispBound` bindings can be replaced at any time, but after a name-procedure binding is `ProcBound` it can no longer be replaced.

Constant values are represented in a program as a single `ProgramNode` structure. The descriptor attached to the program node indicates a procedure that can extract the value from the `ProgramNode`. For each atomic data type (integer, double, time, string, location) supported by *elll* there is a corresponding predefined name-procedure descriptor.

Variables are similar to constant values. The parsed program node points to the variable descriptor of the referenced name and refers to a wired-in procedure that can extract the value from its storage.

All extension language programs are presented to the *elll* interpreter as coded text strings. To create an executable version of an *elll* program the `elllParse()` routine is called with a single argument, the name of a routine which to be called to obtain the next character from the input text.

`ProgramNode *elllParse (void (*`*nextchar*`)())`
: Will parse a text representation of an *elll* expression and return the resulting program tree. *nextchar* is a callback routine used by the parser to obtain the next character of the text being parsed. The routine should reply with `-1` to signal the end of the text.

`Status elllExecProgram (ProgramNode *`*tree*`)`
: The *elll* program rooted at *tree* is executed. Its final return value is found in the `ResultValue` global and the interpreter status is returned as a function result.

`elllExecuteFile (char *`*filename*`)`
: Opens the file indicated by *filename*, parses and executes each expression contained therein.

A `ProgramNode`-represented expression is executed by sending the root of the expression tree to the `elllExecProgram()` routine. The results of the most recent `ProgramNode` execution are stored in the global result register structure `ResultValue`. `elllExecProgram()` saves the old executing node and defines a new current execution context. `elllExecProgram()` calls an internal interpreter dispatcher with the name-procedure binding given by the current `ProgramNode` to invoke the appropriate interpreter code that implements the procedure. If the current executing program node refers to an *elll* program the interpreter execution stack will be pushed and `elllExecProgram()` called back with the `ProgramNode` that is the root of the called *elll* program.

Eventually all program trees arrive at wired-in procedures that actually perform functions of the values defined by the current execution context. Since there is only a single, global, value return structure (`ResultValue`, mentioned earlier) to hold the most recently obtained value, wired-in functions that must execute several `ProgramNode`'s to obtain a final value (e.g. a binary operator) must arrange to keep the intermediate evaluation results somewhere during multiple `ProgramNode` evaluations.

Actual procedure argument list values are available to the wired-in function through the calls described below.

`elllEvalArg(int `*n*`)`
: Evalutates the n^{th} argument in the current actual procedure argument vector and stores the value into the global result register.

`char *elllStringArg(int `*n*`)`
: evaluates the n^{th} argument of the current actual procedure argument vector and attempts to

return its value as a text string

`int elllNumericArg(int` *n*`)`
: Evaluates the n^{th} argument and attemts to return its value as an integer.

Globally declared variables represented by names in the *elll* environment may be accessed through the interpreter function `elllAccessGlobal()`. This mechanism would be used to access variables whose values are not explicitly passed through an argument list.

`elllAccessGlobal(char *`*variable-name*`, int *`*size*`, caddr_t *`*value*`)`
: *variable-name* is a nul terminated string that has the name of the variable to be accessed. *value* is given a pointer to the value container. *size* should be a pointer to an `int` that will receive the number of bytes in the value.

A properly formed value should be returned to the *elll* interpreter (so that it might be used in subsequent function calls). The `elllStoreValue()` and `elllCreateLispObj()` routines provide mechanisms to do this.

`elllStoreValue(int` *binding*`, int` *count*`, caddr_t *`*value*`)`
: creates an *elll* expression value from the supplied scalar or structured value and stores it into the value result register. *Binding* is the kind of object that the Lisp expression result object is to be, one of: `ExprVoid`, `ExprInteger`, `ExprDouble`, `ExprTime`, `ExprString`, `ExprLocation`, `ExprEvent`. *count* is the value to be stored if the *binding* is `ExprInteger`. For all other bindings, *count* is the length of the value referenced by the *value* argument.

`ProgramNode *elllCreateLispObj(int` *binding*`, int` *count*`, caddr_t` *value*`)`
: Creates and returns an *elll* constant valued object (`ProgramNode`) in filled with the appropriate information.

2.3.2 Wired-in Procedures

In general, to make new wired-in procedures known to the *elll* interpreter the wired-in interpreter routine `elllDeclareProc()` routine is called to make necessary the name-procedure binding:

`elllDeclareProc(char *`*name*`, void (*`*routine*`)())`
: *name* is the nul-terminated string text for symbol that invokes the function in an *elll* function call. *routine* is the address of the wired-in code that the *elll* interpreteter is to call.

For example the declaration and definition of the wired-in procedure `RecognizeEvent()` uses the following definitions and calls:

```
static void RecognizeEvent();       -forward declare the routine
                    ...
elllDeclareProc ("recognize", RecognizeEvent);       -tell elll about it
```

```
        ...
static void RecognizeEvent()        -the actual routine definition
      { code for function }
```

The first line simply declares the name so that the C compiler allows its use in the `elllDeclareProc()` call. The `elllDeclareProc()` call informs *elll* that function invocations of `(recognize ...)` are executed by `RecognizeEvent` procedure. The third line is the place at which the actual routine definition must be supplied. No arguments are recognized since the wired-in procedure is not directly called, but invoked indirectly. Actual parameter values are obtained by the wired-in procedure code as stated in Section 2.3.1.

2.3.3 Adding Wired-in Variables

Wired-in variables can be added to the global variable name space with a simple declaration, below. The component that maintains the variable should keep the value in stable storage, i.e. either some statically declared program variable or a heap storage location that can not be freed.

Variable declaration and value assignment is a multiple step process. A binding must be explicitly created by calling the *elll* function `(declare-global ...)`, implicitly by entering a lambda or progn binding block, or argument binding to an *elll* defined function. A binding created in any of these ways is a default integer type with value zero (0). Variable names can be created (added to the variable name table) implicitly by their appearance as an argument in a function invocation. In this context, the parser recognizes the function argument position of a name and creates a name table entry that is added to interpreter variable tables. Initially there is no binding associated with a name declared in this fashion. One must be created through a call to `(declare-global ...)`. Values can be attached to bindings through the `setq` *elll* function, default value assignment upon entering a lambda block, or by actual/formal parameter binding.

`elllDefineVariable(char *`*name*`, caddr_t `*address*`, enum `*binding*`)`
: Creates a variable descriptor for *name* and adds it to the global variable list in the *elll* interpreter domain. *address* references the storage container where the value is kept by the component that declares the name. *binding* determines the type of the value referenced by the variable.

3. Linking to a Stream - Event Sources

Programs that are instrumented to supply primitive events need to locate an appropriate stream for their events and then deliver meaningful events to that stream. The *libEBBA* component of the toolset contains a collection of routines that handle event stream connection, event formatting, and event posting to a sink. When an instrumented program begins to execute, the stream connection service must be invoked to look for an event sink that will handle the kinds of events generated by the source. During its execution, as the instrumented program encounters its event generating annotations, the event posting routines determine if the event to be created is wanted by the sink and, if so, format the event instance record before shipping it to the sink.

3.1 Locating and Connecting to an Event Stream

An event stream is identified by a triple *(location, component, library)*. The *location* part identifies the network host where the sink might be located. The *component* part indicates the type of event stream service expected by the source. The *library* field indicates which event library contains descriptions for the events created at the source. The named event library is found at the host indicated by the location field, there is currently no controlled provision for network access to an event library[1].

A text representation of an event stream triple is used by components to establish connections, supply default values for unspecified parts of a triple, or to pass to other components. The form of the text representation of a connection triple consists of the values for the triple fields separated by colon (`:`) characters:

location: *component*: *library*

For example the string "`grinch:EventMonitor:libpps`" specifies a connection to the Event Monitor on the node named "`grinch`" that is monitoring events from the "`libpps`" event library.

Two routines are available to manipulate the connection tuple text format.

```
ebbaUnpackConnection(connection, location, component, library)
    char *connection, **location, **component, **library;
```

examines the supplied connection string and determines which components are present in the string. `ebbaUnpackConnection()` then applies default values to supply any missing components. Copies of each string are made and a reference to each is copied back to the caller. Missing connection string parts default from left to right as in the following table:

Form	Supplied	Defaulted Components
a	*location*=a	*component, library*
a:b	*location*=a, *component*=b	*library*
a:	*location*=a	*component, library*
a::b	*location*=a, *library*=b	*component*
::a	*library*=a	*location, component*

[1]NFS could access the required files

To obtain default values, `ebbaUnpackConnection()` interrogates the `EVENTSTREAM` environment variable, unpacks the resultant string, and fills in missing parts of the original connection string. If the `EVENTSTREAM` environment variable is not defined or is incomplete `ebbaUnpackConnection()` will supply the local host name for the location, "`EventMonitor`" for the component, and "`default`" for the library. A completely defaulted string (`connection == NULL` or "`:`", "`::`", or"`:::`") will return

(*localhost*:`EventMonitor:default`).

where *localhost* is determined by the *gethostname(2)* service. The complementary routine

```
ebbaPackConnection(connection, location, component, library)
   char **connection, *location, *component, *library;
```

simply formats a connection from supplied component parts in such a way that a corresponding unpack will take the string apart properly.

Delivering events to an event sink requires that an explicit IPC connection to the sink be established. The *libEBBA* routine `ebbaConnect()` handles all of the details of creating the connection to an event stream.

```
ebbaConnect (connection, origin)
   char *connection, *origin;
```

uses the supplied `connection` string to locate the event sink required by the source. `ebbaConnect()` calls the the unpack service to unpack the argument connection string (`*connection`) and supply defaults in the normal fashion. The network address for the host given by the location part of the specified connection descriptor is obtained from the *gethostbyname(2)* service. The component part of the connection is used to obtain the port number expected for the service (the only protocol employed is TCP). An attempt is made to connect to the event sink. A host that is unavailable or non-existant causes the connect to fail with a message and non-zero returned value.

The `origin` argument is used to establish the value attached to the *location* attribute of each event tuple that originates at this component. This is set by calling the `ebbaSetOrigin()` routine.

If successful, the library part of the connection descriptor is sent to the sink in a hello message that requests that the named library is used for further dialog and identifies the source. Upon successful completion of the `ebbaConnect()`, the source has a connection to the requested sink that may be used to send events and exchange *elll* messages.

3.2 Posting Events to the Stream

Once a connection to an event sink is established the instrumented program can send event instance records to the event sink. What is generally required for this process are the annotations embedded in the program text, the event formatting and delivery routines found in *libEBBA*, and the descriptors for individual event templates. Means to automate this entire process are currently under development but not yet available.

A primitive event instance is assembled and sent to an event sink by calling the `ebbaPostEvent()` routine

```
void
ebbaPostEvent (descriptor, a1, a2, ...)
     evtTevent_format *descriptor;
     caddr_t a1, a2, ...;
```

The descriptor is a summary of the shape of a primitive event. It contains all the information needed to format an instance of the event class. The formatting routines have no à priori knowledge of any primitive events. The model builder (*EDL*) assists event source definition by creating, from the descriptions supplied for a library, a collection of primitive event descriptors. Currently these are coded as specially formatted `#define` statements that may be `#include`'d in C programs. Use of an event descriptor in a `ebbaPostEvent()` call will cause the corresponding descriptor to be loaded into the data area of the program.

The descriptor is defined by the C typedef

```
typedef struct _evtTevent_format {
     u_long library_id; /* its library number (Unused) */
     u_long event_number; /* its event number */
     short attribute_count; /* number of attributes */
     char format_string[1]; /* nul-terminated for encoding procedures */
     } evtTevent_format;
```

This is the minimum amount of information needed to create primitive event instances using `ebbaPostEvent()` and allow control over the monitored component. Figure 3 sketches use of these instrumentation parts for a portion of the Sequent Parallel Programming library. The routine is compiled in the normal manner. But since it calls `ebbaPostEvent()` (and some other component must call `ebbaConnect()`) the executable module must be linked with the instrumentation library `libebba.a`. For example the invoking the C compiler as

```
cc -o mandel *.o -lX -lpps_ebba -lebba
```

links all of the `.o` files in the current directory, the X interface library (`libX.a`), the instrumented parallel processing function library (`libpps_ebba.a`), and the *libEBBA* instrumentation library (`libebba.a`).

As the program executes and encounters the event posting annotations, `ebbaPostEvent()` encodes the instance tuples according to the event descriptors described by `e_BarrierEntry` and

```
#include "barrier.h" -gets the event descriptors
    ...
s_wait_barrier(b)
   sbarrier_t *b;
{
   ebbaPostEvent (e_BarrierEntry, getpid (), b->useno, b->limit, b->count);
   S_WAIT_BARRIER(b);
   ebbaPostEvent (e_BarrierExit, getpid (), b->useno, b->limit, b->count);
}
```

Figure 3: Accessing and sending events using `ebbaPostEvent()`

`e_BarrierExit` and the supplied list of attributes. Upon entering the `ebbaPostEvent()` routine the *time* attribute is read from the system clock and added to the attribute list. The *location* attribute is recovered from the `ebbaConnect()` call and added likewise to the attribute list. The tuple formatting routine is invoked and the resulting instance description is sent to the connected event stream.

3.3 Miscellaneous libEBBA Routines

3.3.1 Time routines

Time is stored internally as a `struct timeval` (see `<sys/time.h>`). Externally, time is represented textually in the form

[*hours*:]*minutes*:*seconds*[.*hundredths*]

Time typed values use this form to express constant time values that generally represent time differences or relative times. The *hours* and *hundredths* part of the time value are optional. Fifteen seconds is expressed `0:15`; eight hours is expressed `8:0:0`.

Two routines are supplied to change the form of at time value from one to the other:

`ebbaCvtAscToTime (struct timeval` **time*, `char` **buf*)
: change the ascii representation string for a time value found in *buf* to a binary value referenced by **time*.

`ebbaCvtTimeToAsc (char` **buf*, `struct timeval *time)`
: change a binary time value (**time*) to ascii in the acceptable format in the **buf*.

3.3.2 Encoding event tuples as strings

ebbaEncode*xxx*() is a special version of *printf(3s)* that formats event tuples in the form used by all the components. It includes the following alternate formatting characters:

%s format will include double quote marks (") around the output string,

%#s outputs the string without the quote marks,

%t inserts the ascii time value according to:

- if the argument list item corresponding to the %t format character is 0 (or NULL), the current time of day is obtained and formatted,
- if the argument list item is non-zero, the value is taken as a time value and formatted into the string.

%h inserts a location string obtained as follows:

- the value of the ebbaGetOrigin() call if the corresponding argument list item is NULL,
- the value in the argument list, if non-NULL.

in both cases, the output string is surrounded by double quote marks (").

There are three versions of the encode tuple routine. However, all have a common calling form:

```
ebbaEncodexxx (FILE *iop, char *format, va_list args, int (*rtn)())
```

The tuple encoding routines all take a file buffer descriptor (*iop) as an argument and will fill the buffer with the formatting results. Encoding the event instance into the buffer is guided by the *format string. Printable objects are obtained from the args vector for each format item encountered in the format string. If the buffer fills, the encoding routine will call the supplied callback procedure (*rtn) for disposition of the full buffer. The tuple coding routines expect that the called user procedure will modify the file descriptor to reflect its changed status.

The three versions of the routine differ in how they interpret the args vector and the %h and %t formatting specifications. So, call the one you want.

ebbaEncodeTuple (iop, format, args, rtn)
: always takes its %t and %h values from the environment, i.e. the encoding routine does not use an argument from the args vector. Time (%t) is taken from when the encode is entered; location (%h) from the ebbaSetOrigin() call. The args are scalar values or string pointers

ebbaEncodeObjects (iop, format, args, rtn)
: expects each of its args items to be pointers to an *elll* internal object descriptor. This routine makes the NULL/non-NULL distinction for the %t and %h format characters.

ebbaEncode (iop, format, args, rtn)
: expects each of its args items to be plain old scalar data types.

3.3.3 Event Instance Tuple *location* Value

The value supplied as the location for formatted event tuples (or for any call to the `ebbaEncode`*xxx* routines) when the argument corresponding to the `%h` is `NULL`.

`ebbaSetOrigin (char *`*loc*`)`
: Sets the default location string to the argument *loc*. If *loc* is `NULL`, a default string is obtained by interrogating the environment variable `EVENTLOCATION`. Failing that, the results of a *gethostname(2)* are used.

`ebbaGetOrigin (int *`*length*`)`
: Returns the value assigned as the default location string. If the string is undefined at the time of the call, `ebbaSetOrigin` will be called to set an origin from defaults. If the **length* argument is non-`NULL`, the length of the returned location string is also returned (at `length`).

4. End Points to Streams – Event Sinks

Event sinks can be as simple as a message receiver that obtains the records output by a source (or other sink) and prints them, or as complex as a complete event abstraction component. Becoming an event sink is a fairly involved process that requires name bindings in the communication domain that the toolset operates in. Due to the heavyweight nature of an event sink it is strongly recommended that the author of a new sink reuse existing code to paste a new style of event sink together.

There are two standard sinks available which are generally useful: a standalone event receiver that will listen to an event stream, and a complete event abstraction component. Both use a version of the event queuing module customized with appropriate compiler-time options. The standalone receiver contains a copy of the event queuing component and the *elll* interpreter. It is capable of listening to a large number of event sources (up to 64, `NOFILE` in `/sys/h/param.h`) and executing any event queue related commands. The standalone receiver is not capable of abstracting high-level events from the incoming event stream. The other available sink is a complete event abstraction (event monitor) component capable of abstracting events and passing these results on to other sinks.

4.1 Event Abstraction Component

The Event Monitor component listens to the event stream and attempts to match user defined behavior models to the contents of the incoming stream. The Event Monitor is composed from Event Queuing, Model Matching and instance management, Pending List Maintenance, and *elll* parts. Event Queuing is responsible for listening to the event stream, converting any received events into an internal form, and receiving and executing *elll* messages sent to the Event Recognizer. The

Pending List maintenance is responsible for creating model recognition contexts and decomposing hierarchical models for recognition requests. Model matching attempts to fit the event queue contents to the pending list models. *elll*, of course, supplies helpful functions and the execution model for *elll* messages.

4.1.1 Starting the EventMonitor Component

The Event Monitor component is located in the *EBBA* tools directory (`/usr/local/ebba` by default). Either add this directory name to the shell search path or create an alias to access the routine. The Event Monitor is started with some variation on the following command line:

```
EventMonitor -l libname [-o location] [-s startup] ...
```

This will start the Event Monitor component executing. As the routine struggles to life it invokes an initialization procedure that goes to each major component to execute its startup routine. As each is called a message is output to announce which startup is in progress. The startup message should be similar to

```
Init:...Eval,...Library,...Monitor,...EvtQ,...Interface, done!
```

Once the completed message is output, the Event Monitor is ready to accept recognition requests, events, and other control messages.

The order of initialization should be noted. If the X windows interface was originally configured into the Event Monitor, an attempt is made to access the server given by the `DISPLAY` environment variable. This access precedes the component initialization sequence. A failure here will cause the Event Monitor to exit with a diagnostic.

The first component to be set is the function evaluator since other components will declare *elll* functions that they export. The Event Library indicated by the `-l` *libname* parameter designates the initial event library to use to interpret the contents of the event stream. Presetting the event library interface loads all of the events currently known to the library and declares them to the *elll* component. If no library parameter was supplied, the EventMonitor would have exited before initialization began. If the designated event library (or its librarian) is unaccessible, the initialization fails. If supplied, the `-o` *location* parameter sets the *libEBBA* default location supplied for event tuples generated by this event recognizer. Without an explicit `-l` parameter, the `EVENTLOCATION` environment variable will supply this value; failing that the host name will be used.

Presetting the monitor component clears all of the pending event list structures and scalars associated with pattern matching. Also in the monitor preset, the *elll* functions exported by the monitor and pattern matching parts of the Event Monitor are declared.

Getting to the event queueing module will set the event queue structure and announce the Event Monitor as a server in the internet communications domain. The Event Queueing component accepts all event stream events and *elll* messages that are sent by other components. A common ailment occurs if a previously executing version of the event monitor is still running on the node or if some component is still attached to the previously executing event monitor's event stream. This is manifest as an extraordinarily long pause in the `EvtQ` startup as the queuing module attempts to acquire the event stream port. The user is warned with a message, but if not noticed and corrected, the startup will eventually time out with a diagnostic message.

If the X Windows interface is configured into the Event Monitor, the `Interface` preset phase will create several basic graphics displays used to display activity of the pending event list and event queueing components. The Pending Event List display has an interactive component that allows a user to interrogate the status of entries on the list.

The `-s` *startup-file* option supplies the name of file containing extension language expressions. Up to eight startup files may be specified, each preceded by a `-s` option. Generally, the startup file will contain any function, constant, or variable declarations needed by the user models, that are not hard-wired into the *elll* protocol. (RIGHT NOW it should also contain a `(declare-event` *event-name, event-name,* ...`)` for each event in the library. This will get them declared to the function evaluator cleanly.)

The final step of event monitor startup is to execute each of the startup files given by `-s` options by calling the *elll* interpreter. The startup files are executed in the order given on the command invocation line. The search path for files is `.:~/ebba/`.

4.1.2 *elll* Messages Interpreted by the Event Monitor

The Event Monitor export a substantial number of functions so that users (or other tools) may make recognition requests, control aspects of recognizer operation, and obtain information regarding the status of behavior abstraction operations. The primary function of the Event Monitor is to abstract event stream event instances into user behavior models. Each request for recognition of some behavior model results in creation of one or more descriptors on the pending event list. In order to allow users to interrogate the status of a pending event list entry, each is given a short (or long if you please) string *tag*. All requests to the event monitor for information regarding a descriptor are made in terms of the tag assigned to the descriptor. Users supply the tag when they make recognition requests. The recognizer will derive a tag for any models it requests to support the original recognition request. The form of derived tags is:

User-tag: *derived-tag*. *index*

where *User-tag* is the tag attached to the original high-level model request; *derived-tag* is the name of the constituent high-level event that is part of the high-level model; and *index* distinguishes between multiple requests for the same class of model by the higher-level model.

User interface to pattern matcher:

`(recognize "`*tag*`" (`*event-class* p_1 p_2 `...))` The event *event-class* has a recognition context created and placed on the pending event list. The parameters $p_1, p_2, \ldots$ should match the formal parameter list declared in the event model. The event descriptor is created in the suspended state, so will not participate in pattern matching until it is resumed.

`(delete-pe "`*tag*`")` Changes the status of the pending event located by *tag* to deleted and bumps the usage count of its pending event list slot. The next time the pattern matcher examines the pending event it will be removed from the pending event list. All of its children will notice that their parent pending event has been deleted and commit suicide.

`(suspend-pe "`*tag*`")` Changes the status of the pending event located by *tag* to suspended. The pending event will not take part in pattern matching activity until it is resumed. This does not affect the status of any children or its parent pending event descriptor.

`(resume-pe "`*tag*`")` Changes the status of the indicated pending event to `active`. The pending event is now able to accept input event stream events.

`(start-pe "`*tag*`")` Will change the status of the pending event given by *tag* to active, and also change the status of all of *tag*'s children to active. This function is usually invoked following creation of a new request (via `(recognize ...)`).

`(add-rcb "`*tag*`" '`*fn arg*`)` Creates a response control block that will invoke *fn* with argument *arg*. The RCB is attached to the end of the list of responses for the pending event attached to *tag*.

`(set-queue-position "`*tag*`"` *position*`)` The pending event located by *tag* has its current event queue position set to the entry indicated by *position*. Unfortunately this is currently the address of the event descriptor. Fortunatly, there are two variables defined that will supply common positions: `EventQueueBeginning` is the first event in the queue; `EventQueueTail` is the last event in the queue.

`(reset-PEL)` Clears the pending event list of all descriptors.

Event attribute fetch/store

`(fetch-local` *index*`)` *index* is used as an offset into the `LocalContext` event instance descriptor. The expression node found at that offset is stored into the *elll* result register. If the attribute slot given by *index* is `NULL`, the evaluator status is set to `STATUS_DontKnow`.

`(store-local` *index expression*`)` *expression* is evalutated and the resultant expression result node is stored into the `LocalContext` instance descriptor at the *index* slot.

`(fetch-attribute` *index name*`)` The event located by *index* of the `GlobalContext` instance descriptor has the attribute binding slot for *name* accessed and returned in the *elll* result register. If the slot is `NULL`, the expression evaluator status is set to `STATUS_DontKnow`.

Library/librarian communication

(`reload-event` *event-name*) The current definition for *event-name* (if it exists) is removed from the library interface, the library information file entry for *event-name* is read and a new version of *event-name* is read.

(`merge-library` *background-library merge-library*) The *merge-library* is read and its contents added to the current open background library. If *background-library* is non-`nil` or not the same as the current background library, the library interface shifts to use the new background library.

System control

(`quit-now`) Causes the Event Recognizer to broadcast a shutdown message to all of its clients and then exit gracefully.

(`clear-eventq`) Checks to see if there are any outstanding pending events that might be attached to event queue event instances, and if there are none, clears the queue pointers and returns all of the resources attached to the event instances.

(`go-idle`) Changes the operation mode of the event recognizer from its round-robin polling mode to blocking on the client connections.

(`become-active`) Changes the operation mode of the event recognizer from blocking on client connections to round-robin.

Routines that display information:

(`list-PEL`) Displays the status of all of the current entries on the pending event list.

(`list-completed`) Displays all interpretations of each pending event that has completed and been moved to the completed event list.

(`list-pe` "*tag*") Looks for the pending event with the attached *tag* and will display its current status.

(`display-pe-inputset` "*tag*") Lists all of the events under consideration by the pending event located by *tag*.

(`display-pe-partials` "*tag*") The pending event descriptor attached to *tag* is located and for each interpretation the current event set (guess at what will satisfy the model) is displayed.

(`list-eventq`) Lists all of the events on the event queue.

(`list-eventq-stats`) Simply prints the running summary statistics of event queue activity.

A. Error Messages and Recovery (ha!) Procedures

A.1 errors from libebba

`No service` *component*

The component specified by the final connection string was not found by *getservbyname(2)*. The component should be in the `/etc/services` file or configured into the appropriate Yellow Pages map.

`No host` *location* `for event stream service` (*system-error-message*)

There was no host found by *gethostbyname(2)* to match the location part of the connection string. Either the `/etc/hosts` file or the Yellow Pages map needs to be fixed.

`No socket for event stream` (*system-error-message*)

`ebbaConnect` was unable to establish an endpoint for the client to communicate through. *system error message* should provide a decent reason.

`Could not connect to event stream` (*system-error-message*)

The attempt to attach the client to the event stream failed. *system-error-message* should provide a cryptic reason for the failure.

A.2 errors from elll

`Too many procedure and event definitions`

Overflow of an internal table of procedure names. Nothing simple can be done to rectify this.

name `is already bound to a wired procedure!`

An attempt to create an external hard-wired procedure is about to redefine an existing hard-wired procedure or event name. The previous definition stands.

`Unexpected end-marker (parens mismatched?)`

Output by the token reader when it is trying to read a new *elll* token and it meets nothing but end of file (or end of string) on the stream.

`Improperly formed character constant.`

A character constant (`'c'`) does not have any of the acceptable forms.

`Unterminated string constant.`

An opening quote (") is not closed before the end of line (the next `\n`) character. It is implicitly terminated by the \n.

`List Syntax -- function symbol expected`

The first symbol in a list must be a function name.

`defun syntax -- missing function name`

A `(defun ...)` call is missing the function name for the function that is being defined.

`defun syntax -- incorrect arg list`

The argument list for the function being defined by the `(defun ...)` call is missing.

`Reference to an unbound variable:` "*variable-name*"

variable-name is used as an argument to a function but has yet to be declared.

"*function-name*" `has not been defined yet.`

function-name has not been defined via `DefMac()` or `(defun` *function-name*....

`Too` $\begin{array}{c}\texttt{few}\\\texttt{many}\end{array}$ `arguments to` "*function-name*", *n* `supplied,` *m* `required.`

function-name was called with the wrong number of arguments. *n* and *m* tell the whole story.

`Missing argument` *argument-index* `to` *function-name*

The `EvalArg()` function that accesses the i^{th} argument to the wired function *function-name* that was just invoked could not find the argument given by *argument-index* —too few arguments.

"*function-name*" `didn't return a value;` "*calling-function*" `was expecting it to for argument` *argument-index*.

`EvalArg()` evaluated the argument for *calling-function* given by *argument-index* and did not get a value back. *function-name* was the function invoked at the index.

`Insufficient args to lambda.`

A `(lambda ...` needs at least one argument.

`Insufficent arguments and no default initializers`

Call to a `defun`'ed function with two few arguments and the original `defun` does not have enough initializers to supply values for the missing arguments.

`Bad argument index: (arg` *argument-index*`)`

Internal error? attempt to access a non-existant argument for a function.

"*name*" `is bound to a wired procedure and cannot be rebound!`

Attempt to bind *name* to a wired-in function or event.

`setq first argument must be a declared variable name.`
: form of the `setq` function was improper.

`setq to unbound variable "`*name*`"`
: *name* is not declared as a variable and given an empty binding.

`InCompatible types to` *arithmetic-function*
: Tried to add apples to oranges.

`Division by zero, Mod by zero`
: The usual.

`There is no environment variable named` *name*
: `getenv` looked for, but did not find *name* in the execution environment.

`Can't read` *file-name*
: `Eval_ExecuteFile()` could not load *file-name* before attempting to execute it. Needless to say, *file-name* was not executed!

A.3 Errors from the Event Monitor

`Too many startup files (max =` *n*`)`
: Tooooo many `-s` options have been specified. *n* is the number that may be specified. This causes the Event Monitor to cease to get started; the solution is to combine one or more startup files.

`usage: emon [-l libname] [-s file] -offending arg:` *s*
: the string given by *s* is not recognized as a valid Event Monitor argument.

`Couldn't connect to X`
: If the Event Monitor has the X windows interface compiled in, it must be able to access the server designated by the `DISPLAY` environment variable. Either the the environment variable is not set, or the server will not accept the connection.

`Need to specify a library with -l.`
: The Event Monitor requires a library to start things off with. Although libraries may be merged in later, starting requires a library.

`Could not establish event stream RFC listener socket -` *system-error-message*
: For some reason or other, *system-error-message* will hopefully tell us, the Event Queueing component could not establish the socket it needs for the event stream.

`Wait for bind....`

If the Event Queueing component cannot bind the event stream socket it will issue the "`Wait` ...message, then on each retry output a new .. After about 10 dots it will give it up and will issue the next message

`Could not bind event stream RFC socket -` *system-error-message*

The Event Queueing module could not bind the Event Stream connection socket to the event stream port. There is another process using the port number. Either an old incarnation of the Event Monitor, or some other component that was attached to the old incarnation of the Event Monitor, is still running and holding the port that the socket attempts to bind.

`Error accepting RFC -` *system-error-message*

A client has tried to join the event stream, the Event Queueing component has attempted to accept the client, but to no avail. Maybe *system-error-message* will be helpful.

`Clear Event Queue?  There are outstanding Pending Events`

A `(clear-eventq)` has been sent to the Event Monitor, but the clever monitor will not get rid of the events while there are outstanding pending event requests. No action is taken on the event stream. If you really want to clear the queue, delete all of the outstanding pending events. This should be made more rational in a future release.

`CreatePending:  Late tables being created for` *event*

This reflects an attempt to correct a possible internal error. A recognition request was made for *event*, but the tables that describe the recognizer for the event were not attached to the description. The event monitor attempts to get the tables at this point.

`Constituent` *name* `not defined for event` *event-name*`.`

A recognition request was made for *event-name* and implicitly (by model decomposition/reconstitution) for its constituent event *name*. Too bad, *name* has no definition at this point. Sub parts of the high-level model are being parceled out for recognition, this one will cause a problem. Define the constituent event.

`Defective recognizer for` *event-name*

This is caused when the event monitor requests that the recognizer for the event *event-name* be cast into interpreter tables and for some reason it does not happen. There is probably something wrong with the event library entry for *event-name*, or some error occurred when the library interface tried to read it. Best bet is to recompile the model and try again.

`EventInstance:  (`*event-class attribute attribute ...*`)`

Was noticed by the Event Monitor.

`Wrong number of arguments event` *event-name*`;` *n* `supplied,` *m* `needed.`

A recognition request for event *event-name* did not contain the number of event parameters specified in the model definition. This causes the machinery that is building the recognition request to grind to a halt and the request fails. This should be made reasonable some other time.

`Request for undeclared event` *event-name*`, ignored.`

Either the Event Recognizer has not been informed via a `(reload-event` *event-name*`)` *elll* message that the event is available

- or -

It needs a `(declare-event` *event-name*`)` to be issued so that the *elll* evaluator can know about the event. Next release fixes this.

`Constituent` *event-name* `not in event library`

A constituent *event-name* of a high-level event model is not in the library. This is issued when the Event Monitor is examining the structure of a model that it has been asked to recognize. The event *name* will be treated like a primitive event and given a temporary identity in the Event Monitor. So that if someone comes up with an instance, or a definition, things will proceed smoothly. The event *event-name* will still get its pending event descriptors created.

References

[1] P.C. Bates. The *ebba* Modelling Tool, a.k.a Event Definition Language. Technical Report 87–35, University of Massachusetts, April 1987.

[2] P.C. Bates. Tracking the Elusive Mandelbrot Set Error Using Event Based Behavior Abstraction. Technical Report 89-06, University of Massachusetts, February 1989.