EBBA Modelling Tool
a.k.a
Event Definition Language

Peter C. Bates

COINS Technical Report 89–09
February 10, 1989
(Revised from: 87–35, April 1987)

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. And by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract NR049-041.

EBBA Modelling Tool
a.k.a
Event Definition Language

Peter C. Bates

February 10, 1989

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

ABSTRACT

This report describes how to use the Model Builder and the Event Definition Language (*EDL*) as implemented in the *EBBA* toolset. The *EBBA* toolset is a collection of programs that allow its users to monitor (and experiment with) a program distributed over a network of processors. The Event Definition Language (*EDL*) is a mechanism that enables its users to express models of system behavior as abstractions of the detailed computation comprising that behavior. *EDL* users express their behavioral abstractions as high-level event clusters in terms of the primitive events that characterize the system, and high-level events that were previously defined through that viewpoint.

This report describes a model building tool based on the *EDL* that is integrated with the other components of the *EBBA* toolset. The model builder is largely a translator for *EDL*-described behavior models. However, the language that it accepts is a bit richer than *EDL* in that it accepts directives that control compilation and event library management.

Contents

CONTENTS 3

1. Introduction

The *EBBA* toolset is a collection of programs that allow its users to monitor (and experiment with) a program distributed over a network of processors. The philosophy and implications of this high-level debugging paradigm are developed in a thesis[1] and a detailed description of the toolset internals found in another report[2]. This report describes how to use the Model Builder and the Event Definition Language (*EDL*) as implemented in the *EBBA* toolset[3].

The Event Definition Language is a mechanism that enables its users to express models of system behavior as abstractions of the detailed computation comprising that behavior. Events are used to represent some significant system transition or interaction of system components. *EDL* users express their behavioral abstractions as high-level event descriptions in terms of the primitive events that characterize the system through a *viewpoint*, and high-level events that were previously defined through that same viewpoint.

Event exchange is the medium used for all system understanding in *EBBA*. 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 category while the attribute values serve to characterize and differentiate specific occurrences of the event. 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. Every event class can be represented with a template having the general form:

(event-class a_1 $a_2 \ldots$ *time location).*

The attributes $a_1, a_2, \ldots$ *time, location* correspond to targets of value binding expressions described in **with** clauses of the *EDL* modelling language description for the event class. 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.

Primitive events represent the finest observable granularity of a system's activity, generally that provided by the functional level the system defines for its users. As a system executes, it creates *instances* of its primitive event classes to evidence its behavior. Given the current view of events, the class of an event instance is largely determined by where in the executing system the instance is detected. All instances of a single event class are (probably) created by the same

---

[1] P.C. Bates, "Debugging Programs in a Distributed System Environment". COINS Technical Report 86-05, January 1986

[2] P.C. Bates, "The EBBA Toolset Implementation Notes"

[3] P.C. Bates, "Distributed Debugging tools for Distributed Systems", COINS Tech Report 87-28, March 1987

instrumentation probe in the monitored system. The values bound to the attribute fields of the instance are determined from the system state available as 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.

The *Event Stream* 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, quantum rundown during event instance generation adds uncertainty, and the statistical access to the physical communication medium confuses internode relations further.

The model builder is basically a translator for *EDL* described behavior models. The language that it accepts is a bit richer than *EDL* in that it accepts directives that control compilation and permit macro definitions. Both primitive and high-level events must be described in *EDL*. Primitive events require little descriptive power. In an executing system primitive events can be created in ad hoc ways, or more precisely using model builder outputs together with *libEBBA* routines[4]. High-level events on the other hand, represent user models of behavior that need to describe complex system activities. High-level event instances are, in a sense, a creation of the *EBBA* model matching tools and so require more powerful means to describe the intended behavioral model.

Primitive events described in *EDL* simply indicate their unique system-wide event names and supply a list of names for the attributes possessed by the event. This simple description gives enough information to determine the shape of a primitive event and output a descriptor to be used in creating instances of the event.

Descriptions of higher level events are more elaborate since they involve use of *EBBA* clustering and filtering techniques. Clustering to describe a high-level model is effected by specifying a set of previously defined events in an event expression and imposing constraints on the event expression members' attributes in a set of relational expressions. The event and relational expressions serve in dual roles of describing a behavioral model and providing a guide for recognizing the occurrence of an event representing the model. The attributes of a high-level event are expressed in terms of the attributes defined by the cluster member events and other information obtainable from the *EBBA* model matching environment.

The next section describes the lexical elements from which *EDL* definitions are composed and the form of the input language to the model builder. The following section explains in detail

[4]P.C. Bates, "Event Monitoring and Abstraction Tools", COINS Tech Report 88-, Nov 1988

how high-level events are written in *EDL*. Section four describes how the primitive events which characterize a system viewpoint are described in *EDL* and relates these event definitions to the event templates moved around the system. The final section provides an example of the use of *EDL* to describe a couple of behavior models.

2. Lexical Elements of *EDL* Descriptions

EDL event descriptions are composed from a limited set of lexical elements including keywords, identifiers, values, expression operators, and punctuation marks. Keywords introduce clauses and special values and will be written here in bold face letters, e.g. **event**, **primitive**. Use of individual keywords will be described in appropriate sections.

2.1 Identifiers and Keywords

Identifiers are constructed from contiguous sequences of letters, digits, and underscores with the restriction that the first be a letter. Identifiers are used to represent event attributes, data type names, event class names and parameters, constants, and external functions. Identifiers will be written in italic throughout this and following sections. Some valid identifiers would be

Send *CreateHyp* *Create_Special_Node*
Event3 *TIME_OF_DAY* *legitimate_10_folds*

Identifier strings are not case sensitive. The valid identifer *MiXeD* is the same as ***mIxEd***.

Keywords are syntactic markers that provide a framework for event descriptions and serve to delimit user supplied parts of definitions. Like identifier strings keywords are not case sensitive, as well as being reserved to the model builder. No keyword in any combination of character case may be used as an identifier. The following are the keywords of the *EDL* model building language.

cond **end** **event**
is **primitive** **with**

Type names are not keywords but identifiers that occur in limited contexts following event parameter and attribute name declarations. Currently valid type names are

`integer` - denoting positive or negative (or zero) whole numbers,

`double` - floating point numbers whose container size is the determined by the host system,

`bitstring` - 32 bit unsigned integers,

`time` - values that denote time of day, used for event time stamps. Clock resolution and computational formats are determined by the host systems, but the external representation is as described below,

`string` - contiguous lists of unsigned character-sized values that may contain both printing and non-printing values terminated by a `nul` character,

`location` - treated exactly like strings, but is used to location stamp event tuples. Interpretation of location types is reserved for later to accomodate different hardware and software architectures. For example, the location string might contain the processor number of a multiprocessor or a thread identifier of a lightweight process.

2.2 Constant Values

Constant expression values may be either numbers or character strings that are used to represent absolute quantities. Constant values are used in arithmetic and relational expressions and as indexes for event expression member events.

2.2.1 Numbers

EDL allows numbers to be specified as signed decimal and floating point numbers or radix-specific bit strings. The range of values, the container size, alignment, and representation are, as always, system dependent. Some examples

```
37.661      -392       0xFF01
0b101101    12.2e-2    0o77376
```

Numbers consisting of a string of decimal digits (0..9) are typed as integers. Likewise, radix-specific bitstrings are classed as integers. Radix specified numbers begin with a zero followed by a radix specification character; then valid characters from the radix. Three common radix forms are available:

binary - specified as `0b`*ddd*; where each *d* may be `0` or `1`,

octal - specified as `0o`*ddd*; each *d* is `0` .. `7`,

hexidecimal - specified as `0x`*ddd*; each *d* is `0` .. `9` or (`A` ..`f`).

Other numbers (not radix-specific) may also begin with a zero character but are interpreted as decimal numbers.

Floating point numbers can be written as decimal digit strings containing a decimal point (to separate the fractional part in the usual manner) or in scientific notation of a decimal string

followed by a base ten exponent part. The decimal point or exponent part (or both) is required to distinguish floating point from integer numbers.

The general forms are:

[{±}]*dd.dd*
[{±}]*dd*e[{±}]*ddd*
[{±}]*dd.dd*e[{±}]*dd*

Embedded spaces cause problems (e.g. between the e and the exponent value digit string) and the digits are, of course, all decimal digits.

2.2.2 Time values

Time values are an important part of events generated in complex systems. Event generators time stamp events with a form

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

EDL descriptions containing 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`.

2.2.3 Strings

A character string is written as a sequence of characters enclosed in quotation marks, '"'. Non-printing characters can be inserted into string constants by writing the integer values preceded by a "\" escape character. Some common non-printing characters have shorthand forms:

\n inserts a new-line character;
\t inserts a horizontal tabbing character;
\e inserts the ESC character;
\b inserts the backspace character (ctrl-h);
\r inserts a carriage return;
\f inserts a form feed character;
\\ inserts the "itself;
\" inserts a double quote character (the normal string terminator).

As with numbers, all specifics of the representation are system dependent. Some string examples

```
"vax9"
"An escape sequence \27[H"
```

Single character values common in most programming languages are supported only as integers. Single character values are expressed by sandwiching the single character between two ' marks, e.g., 'Q' or '\033'. Since integer and string types are incompatible types tool users should choose either a string or integer representation for single character values.

2.3 Extended Input Language–Preprocessor Directives

Translation of primitive events and model descriptions is a two part process. The first part uses a macro preprocessor (usually /lib/cpp) to remove comments, conditionally insert text, include auxillary files, and perform substitution of named macro text. The reworked source text is then examined by the *EDL* language translator to create the descriptive structures and recognizers for the described models.

2.3.1 Comments

Comment text may be embedded in the input to the model builder by enclosing any set of characters between the opening delimiter "/*" and the closing delimiter "*/". For example

```
/*
 * e_BarrierExit Wed Oct 19 09:51:12 1988
 */
event e_BarrierExit is ...
```

The text in the first three lines is a comment, removed by the preprocessor and so ignored by the compiler. A comment string can appear anywhere in the text, but its removal is not always as though it just "vanishes". For example the following:

```
event e_Barrier/* is there a space??  */Exit is

primi/*
there are linefeeds*/tive "e_BarrierExit"
```

results in the following

```
event e_BarrierExit is

primi
tive "e_BarrierExit"
```

which is not syntactically correct. Also, comments do not nest.

2.3.2 Macro Definitions

Macro definitions have the form

`#define` *name[(*arg_1,arg_2,...*)] token-string*

Following a macro definition any use of *name* will be replaced by *token-string*. If arguments $arg_1, arg_2, \ldots$ are supplied, they should appear in the *token-string* if they are to be useful. For example a macro `min` defined as:

```
#define min(x,y) (x < y) ?  x :  y
```

when used such as

```
with
  count := min (e_BarrierEntry.count, e_BarrierExit.count)
```

results in the following input

```
with
  count := e_BarrierEntry.count < e_BarrierExit.count
         ?  e_BarrierEntry.count :  e_BarrierExit.count
```

Macro names embedded in string or character constants are not expanded into their token string.

A macro name may have its definition cleared and the name no longer recognized as a macro by using the form

`#undef` *name*

Macro name usage may not be nested, two `#define`'s for the same name without an intervening `#undef` for that name will be greeted with a diagnostic message from the preprocessor.

2.3.3 File Inclusion

The form of the file include directive is

`#include` "*filename.edl*"

The preprocessor will read text from the included file until it encounters the end of file. Following end of file on the included file the preprocessor resumes reading from the file that contains the `#include` directive. Included files may be nested.

2.3.4 Conditional Compilation

Statements may be conditionally included in or excluded from the compilation process by using the one of the `#if`*something* forms.

`#if` *constant-valued-expression* -or-
`#ifdef` *macro-name* -or-
`#ifndef` *macro-name*

introduce a conditional inclusion block. The statement

`#endif`

terminates a conditional block.

In the first form, the constant expression consists of value bearing macro names and expression operators (all of those mentioned in section 3.3 are permitted). If the expression evaluates to a non-zero value, the statements following the `#if` line up to an `#endif` or `#else` are included into the text. A zero value results in exclusion of the lines from the resultant text. In the second and third forms *macro-name* is simply tested to see if it has been `#define`'ed or added as a `-D`*name* compile option (`#ifdef`) or not (`#ifndef`). The else construct

`#else`

may be used between an `#if` and its `#endif` to include (or exclude) text based on the inverted results of the if test.

3. Describing High-Level Event Models

A single high-level event definition describes a model of some aspect of a system's overall behavior. Each event definition is composed from a heading which supplies the name for the event class being defined and up to three kinds of defining clause which specify the structure and intent of the event. The **is** clause defines the event expression that specifies what event classes constitute the cluster and indicates acceptable orderings for their instances. The **cond** clause details a set of relational expressions defined over the attributes of the event expression event classes. These relational expressions constrain the attributes of events that appear in the event expression to specific values or value ranges. The event heading together with a set of attribute binding expressions defined in the **with** clause of a definition describe the event tuple for the event class represented by the event definition. The attribute binding expressions are also expressed in terms of the event expression members' attributes.

3.1 Event Heading

The event heading of an event definition associates a name by which the event class is known and referred to by users of the modelling tools employing *EDL*. The event name that appears in the heading is not necessarily the identifying name used internally by the *EBBA* model matching components. However, it is important that there is available a unique mapping of system defined event names onto the external names with which an event-based view is constructed by a user. This is necessary so that users have some basis to believe that what they see represents their view of a system. Within a given viewpoint, all event names are required to be unique.

An event heading may have an optional parameter list which provides a means of parameterizing events. The parameter list is specified as a list of names acting as place holders for values that are supplied when a request is made to match the event description to actual system activity. This permits a limited degree of tailoring an event recognition request to dynamic system conditions. The following event heading

event *FrontEndCompletion(node:location)* **is**

introduces an event class named *FrontEndCompletion* which is invoked with a single parameter, *node*. The *node* parameter can appear later as an operand in relational or arithmetic expressions in the **cond** or **with** clauses of the event description. A value is bound to this parameter when a request is made for recognition of an instance of the *FrontEndCompletion* event. Types will be described later in section 3.3.1. The example heading might be useful where a user is interested in instances of the *FrontEndCompletion* event only if they occur on a specific node of the distributed system. For example, in the debugging monitor, a request for recognition of an instance of the *FrontEndCompletion* would have the form[5]:

```
Match-> (recognize "fec" (FrontEndCompletion "vax9"))
```

The event recognizer would be requested to watch the event stream for an instance of the high-level *FrontEndCompletion* event with the string value "vax9" bound to the *node* parameter.

3.2 Event Expression

The event expression is a regular expression-like string of event symbols and event operators. The event symbols represent constituent behaviors of the high-level event model which is described

[5]The "Match->" is a prompt from the Event Monitor toolset component indicating that it is looking for something to do.

by the enclosing event description. The operators indicate alternative orderings that sets of event instances may have in order to match the event expression members. In general, the event expression will only describe an outline for a behavior model. The event operators define acceptable orderings, thus implicitly imposing time relations on event expression constituent events. However, events carry a richer set of attributes that may be used to relate collections of events. These relations are handled by the **cond** clause, described later.

Figure 1 expands the earlier *FrontEndCompletion* example to include an event expression. The *FrontEndCompletion* event is defined in terms of three other events: *InstantiateKsi*, *CreateHyp*, and *InvokeKs*.

event *FrontEndCompletion(node:location)* **is**
InstantiateKsi[1] o *CreateHyp** o *InstantiateKsi[2]* o *InvokeKs*
end

Figure 1: Event heading with event expression

The event expression provides a means to perform coarse filtering of event stream instances. When an attempt is made to recognize an instance of the event, only event stream instances in the same class as event expression symbols need be examined for possible inclusion as constituent events.

The subscript notation appended to each of the *InstantiateKsi* event symbols serves to distinguish the two occurrences when they are used as operand qualifiers in **cond** and **with** clauses. Without the subscripts all expression references to attributes of an *InstantiateKsi* event would access the values contained in the instance bound to the first *InstantiateKsi* of the event expression.

Event expressions need to describe a wide range of behavioral patterns. In distributed systems, behaviors will consist of sets of events which are only partially ordered. Sometimes, the set consists of events which need to be ordered in strict sequences. Other times the events occur concurrently, and will be observed in arbitrary order. Conventional regular expression formalisms are capable of describing sequences in various ways but do not provide means for expressing concurrency. *EDL* event operators include operators that specify sequencing, choice and repetition as well as an operator describing concurrently occurring sets of events.

3.2.1 Sequencing

The binary sequence operator, indicated with " o " (@ on some keyboards), specifies that its operand events are to occur in the left to right order in which they are written. For example the

partial expression

$$\ldots \mathit{InstantiateKsi} \circ \mathit{InvokeKs} \circ \mathit{SendHyp} \ldots$$

indicates that the described behavior consists of an instance of the *InstantiateKsi* event followed at a later time by an *InvokeKs* event, which is subsequently followed by a *SendHyp* event. Implicitly, the three events are related by a relation of their time attributes where

$$\mathit{InstantiateKsi.time} < \mathit{InvokeKs.time} < \mathit{SendHyp.time} .$$

3.2.2 Repetition

Indefinitely long sequences of events are specified by two postfix unary operators, plus and star, indicated by $^+$ and *, respectively. Plus indicates that the described event sequence consists of one or more repetitions of its event operand. Star indicates a sequence of zero or more instances of its operand event. Thus, the partial event expression

$$\ldots \mathit{InstantiateKsi} \circ \mathit{CreateHyp}^* \circ \mathit{InstantiateKsi} \ldots$$

describes the sequence

```
(InstantiateKsi . . .)
(InstantiateKsi . . .)
```

or

```
(InstantiateKsi . . .)
(CreateHyp . . . )
(InstantiateKsi . . .)
```

or

```
(InstantiateKsi . . .)
(CreateHyp . . . )
(CreateHyp . . . )
(CreateHyp . . . )
(CreateHyp . . . )
(InstantiateKsi . . .)
```

and so on. Whereas

$$\ldots \mathit{InstantiateKsi} \circ \mathit{CreateHyp}^+ \circ \mathit{InstantiateKsi} \ldots$$

will describe all but the first of the above event sequences.

3.2.3 Choice

Specifying a choice between different events is accomplished by the alternation operator, " | ". Alternation indicates that an instance of at least one of its operand events is necessary to match the event expression. For example, the partial event expression

$$\ldots \mathit{CreateDDGoal} \circ (\mathit{ReceiveHyp} \mid \mathit{CreateHyp}) \circ \mathit{InvokeKs} \ldots$$

matches the event subsequence

```
(CreateDDGoal . . .)
(CreateHyp . . .)
(InvokeKs . . .)
```

or the subsequence

```
(CreateDDGoal . . .)
(ReceiveHyp . . .)
(InvokeKs . . .)
```

Random choice among a group of operand events is possible by specifying a form:

$$\mathit{CreateHyp} \mid \mathit{SendHyp} \mid \mathit{CreateGoal} \mid \ldots \mid \mathit{SendGoal}$$

The above expression indicates that the specified behavior consists of any *one* of the *CreateHyp*...*SendGoal* events. The alternation operator also has the property that the fully parenthesized expression

$$(\ldots((\mathit{CreateHyp} \mid \mathit{SendHyp}) \mid \mathit{CreateGoal}) \mid \ldots \mid \mathit{SendGoal})$$

is the same as the above, unparenthesized expression.

Note that the definition of alternation does not state "... but not both ..." – exclusion is not its purpose. The nature of the event stream is such that all alternates could occur in the proper context. However, to successfully match the event expression to the stream only requires one of the alternates. Which event is chosen will depend on the tools employed to match the models to event streams.

3.2.4 Concurrency

Concurrency is described using the shuffle operator, written " $\triangle$ " (^ on some keyboards). The shuffle operator designates a set of events, *all* of whose members must occur, but there is no preferred ordering imposed on the set members. For example,

$$\ldots \mathit{InvokeKs} \circ (\mathit{CreateHyp} \triangle \mathit{SendHyp}) \circ \mathit{InvokeKs} \ldots$$

describes the sequence

```
(InvokeKs . . .)
(SendHyp . . .)
(CreateHyp . . .)
(InvokeKs . . .)
```

or the sequence

```
(InvokeKs . . .)
(CreateHyp . . .)
(SendHyp . . .)
(InvokeKs . . .)
```

Similar to the alternation operator, a series of event names joined with shuffle operators in an event expression string such as

$$\ldots \mathit{CreateHyp} \triangle \mathit{ReceiveHyp} \triangle \mathit{CreateDDGoal} \ldots$$

is the same as the parenthesized version

$$\ldots ((\mathit{CreateHyp} \triangle \mathit{ReceiveHyp}) \triangle \mathit{CreateDDGoal}) \ldots .$$

This use of the shuffle operator designates a cluster of events in which *all* of the events participating in the shuffle must occur.

The use of the shuffle operator to denote concurrent events does not imply that (in the above example)

$$\mathit{CreateHyp.time} = \mathit{ReceiveHyp.time} = \mathit{CreateDDGoal.time}$$

It is also not true that if one operand of a shuffle occurs "first", the rest will occur "later". That is, if the following are true about the constituents bound to the event instance represented by the above shuffle

$$\begin{aligned} \mathit{CreateHyp.time} &= t_1 \\ \mathit{ReceveHyp.time} &= t_2 \\ \mathit{CreateDDGoal.time} &= t_3 \end{aligned}$$

there is no guarantee that an event sequence matching the shuffle event string implies any relations between the attribute times t_1, t_2, or t_3.

```
event FrontEndCompletion( atnode:location ) is
    InstantiateKsi[1] o CreateHyp*o InstantiateKsi[2] o InvokeKs
cond
    InvokeKs.location ==atnode;
    InstantiateKsi[1].location ==InstantiateKsi[2].location
end
```

Figure 2: Event definition with constraining clauses

3.3 Cond Clauses

The **cond** clause contains a set of relational expressions which are used to constrain the attributes of event expression members to certain values or value ranges. The set of **cond** clause relationals that are part of an event description specify that only instances of event expression members possessing certain characteristics are to be considered eligible to match the described high-level behavior.

The relations among events expressed by **cond** clause constraints provide finer grained filtering than that provided by the event expression since they are defined in terms of the attributes of event expression members. For recognition of a particular event, the high-level event recognition machinery only selects, from the event stream, event instances that match event expression members. Following this coarse filtering, any **cond** clause relations defined in the high-level event involving attributes of the instance matching an event expression member must be evaluated to determine if the selected instance possesses appropriate attributes.

These constraints on event attributes thus serve both to narrow the scope of a model and as filters which eliminate unnecessary instances of events from inclusion into a behavior model. A **cond** clause is an optional part of an event description. In the absence of a **cond** clause, any set of events from the appropriate classes which match a pattern prescribed by the event expression will constitute an instance of the defined event. Figure 2 is the earlier *FrontEndCompletion* example extended to include **cond** clause constraints.

3.4 Operators and Operands for Relational and Arithmetic Expressions

3.4.1 Arithmetic and Relational Operators

The set of operators available for **cond** clause relational expressions is a rich set of arithmetic, logical, and relational operators. Arithmetic operators include addition (+), subtraction (-), mul-

tiplication (*), division (/), and modulus (%). Bitwise logical operations are shift left (<<), shift right (>>), and (&&), and or (||). The conditional expression selection operators ? and : used as

e ? e1 : e2

evaluates the expression *e*, then evaluates the expression *e1* if the value for *e* was non-zero, or evalutates the expression *e2* if the value for *e* was zero. The arithmetic and bitwise logicals are all binary numeric operations and return numeric values.

Available relational operators are the usual less than (<), less than or equal (<=), equality (==), inequality (!=), greater than or equal (>=), and greater than (>). These are defined over all values and return false (value 0) and true (value 1) results. Logical connectives and (&) and or (|) supply conjunction and disjunction of boolean values. Unary complement operators are negate (-) for arithmetic values, bitwise logical complement (~), and not (!) for boolean expressions. Of course, all of these may be augmented with external functions defined in the *EBBA* tool environment.

Expression operands have five sources: event heading parameters, event instance attributes, so-called "lexical" functions, system defined functions and simple constant values. Explicit means is provided to associate data type information with all operands. Thus, type compatibility checking is performed for individual expressions to ensure they can actually be evaluated meaningfully.

No capability is provided for creating data types. Users are limited to several scalar types generally considered useful

numbers - can take on type *integer*, *double* (floating point), and *bitstring*

strings - have a single data type, *string*.

time - a specific time type is provided, *time*. The written value for time has the form

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

The hours and hundredths parts are not necessary for a complete time specification.

location - provision exists for values to be typed as *location* although currently location and string types are considered the same. The general convention for designating locations has values take the form

"*node-name*:*component-name*"

The following subsections detail each source of expression operand by describing the access mechanism and when an operand takes on a value. This discussion of operands and value-producing expressions applies equally well to attribute binding expressions in **with** clauses.

3.4.2 Event Heading Parameters

Event heading parameters are identifiers listed in the parameter section of an event heading. Their importance is that they can be used to parameterize the value producing expressions in an event definition. Each parameter in the list is declared with its type using the

parameter-name: *type*

notation. Event heading parameters are accessed in expressions as unadorned identifiers. Values are bound to event heading parameters when a recognition request is made for an instance of the event class. In **cond** clause relational expressions, use of event heading parameters serves to tailor the filtering effect of the **cond** clause to dynamic system conditions. Refer to figure 2 for an example of the use of an event parameter as an expression operand.

3.4.3 Event Attributes

Event attributes are the most important kind of valued operand because event attributes are the primary means for focusing particular behavior abstractions. Values represented by these operands are obtained from event instance tuples bound to event expression members. Event attribute operands are specified by using the attribute name prefixed by the name of the event with which the attribute is associated. The qualifying event symbol must appear as a member of the event expression of the enclosing event description.

If the *FrontEndCompletion* event expression members have the event templates

(InstantiateKsi ... time location)
(CreateHyp time ... time location)
(InvokeKs time ... time location)

the focus of the *FrontEndCompletion* event can be narrowed by specifying constraints in terms of the event expression member attributes. The attribute access is written using the common field access dot notation as in numerous programming languages. For example, the event definition of figure 2 constrains all of the event stream events which are bound to the event expression members to have occurred on the same node of the system.

Recall that the subscript notation is important when an event name is used more than one time in an event expression. Value expressions use the subscripts to specify which qualifying event supplies the attribute value when the expression is evaluated. The subscript notation only applies to the explicit mentions of an event in an event expression. The correspondence between event expression events and qualified event attributes is made when the event description is created. The

event *FrontEndCompletion(count:integer)* **is**
InstantiateKsi[1] o *CreateHyp** o *InstantiateKsi[2]* o *InvokeKs*
cond
numberof(CreateHyp) <*count*
end

Figure 3: Event expression "Lexical" functions

implication of this is that the attributes of a repetitive event instance, e.g. the *CreateHyp* events bound to the event expression

$$\ldots \text{o } \mathit{CreateHyp}^{+} \text{o } \ldots$$

are **not** accessed as *CreateHyp[1].node, CreateHyp[2].node,* etc. The only valid subscripted event attribute qualifier for this string would be *CreateHyp[1].node,* since only a single mention of the *CreateHyp* event is made in the event expression. Accesses to specific instances of the string of *CreateHyp* events bound to the $\mathit{CreateHyp}^{+}$ event expression member are performed through lexical functions.

3.4.4 Event Expression "lexical" Functions

Certain event expression constructs have attributes which result from the binding of event stream instances to the event expression member events. Attributes include the total number of events bound to a repetitive event expression or the absence of an event instance from the bound event string. The first is given by the *numberof()* lexical function, the later by the *undefined()* lexical function. For example, as shown in figure 3 the *FrontEndCompletion* event might only be "interesting" if the number of *CreateHyp* events is smaller than some number, *count,* supplied as a parameter. Other types of lexical functions help to unravel things which are not easily expressed in the event expression format, such as accessing the first or last instance of a repetitive sequence. For example the function

last(CreateHyp).node

selects the *node* attribute of the last *CreateHyp* instance which has been bound to the $\mathit{CreateHyp}^{+}$ part of the event expression string.

3.4.5 Environment Specific Functions

These functions are supplied to return values obtained from the environment supporting the abstraction of event instances. What functions are supplied is highly implementation dependent, but they at least include a *timestamp()* and a *locationstamp()* function. Environment specific functions are provided largely as a catch-all for unanticipated, but necessary, functionality in the evaluation environment. Functions are defined using the *EBBA* toolset Extension Language (*elll*) and are loaded into the Event Recognizer when it starts either through startup files or interactively as the recognizer is executing.

3.5 With Clauses

The list of names for attributes possessed by an event and a description of how to bind values to each attribute when an instance of the event is recognized is found in the **with** clause expression. Each **with** clause expression names and describes a single event attribute. The attribute is defined as an identifier together with the attribute's type as the target of an assignment operator. The expressions which provide values for event attributes are composed from the same set of operators and operands as **cond** clause expressions.

All events have predefined attributes time and location. By convention, these attributes fill in the last two attribute slots of an event tuple. This can be overridden by explicit inclusion of the time attribute as

time: *time* := *timestamp()*

and the location attribute as

location: *location*:= *locationstamp()*

in of the desired position in the **with** clause. Overriding the automatic inclusion of the time and location attributes, or changing their position in the attribute list could make use of the *libEBBA* event posting routines less reliable.

When a set of events matching one of the possible event expression sequences has been accumulated and all **cond** clause constraints evaluate satisfactorily, the event is instantiated in its own right. Using the context supplied by its parameters and attributes of the constituent events, the attribute binding expressions are evaluated and the value for each slot in the event tuple template is filled in.

To illustrate, in figure 4 the familiar *FrontEndCompletion* event can be expanded to possess attributes (in addition to the predeclared attributes for time and location) for *elapsedtime* and

```
event FrontEndCompletion( count:integer ) is
    ...
with
    hypcount:integer := numberof(CreateHyp);
    elapsedtime:time := InvokeKs.time -InstantiateKsi[1].time
end
```

Figure 4: Example attribute binding expressions

hypcount. The *elapsedtime* attribute is bound to the difference in the time attributes of the events which are first and last in the event string bound to an instance of the *FrontEndCompletion* event. The *hypcount* attribute reflects the length of the string of *CreateHyp* instances bound to this event expression.

Like the **cond** clause, the **with** clause is an optional part of the event definition. However, every event instance will carry with it certain predefined attributes, such as time of occurrence, that might serve to distinguish various instances of the class. Other predefined attributes might be dependent on specific characteristics of the system, such as the name of the processor node on which the event occurred.

4. Describing Primitive Events

Primitive event descriptions consist of a special event heading and a reduced form of **with** clause. The event heading names an event class and declares the event class as **primitive**. Following the **primitive** keyword is a value, the system specific identifier, that the event generating source puts into the event class slot for an instance of this event class. The form and value of the system specific identifier following the **primitive** keyword is system and implementation dependent. It could be a number, a string, or some arbitrary bit pattern. The standard event reader prefers an ascii string representation.

The unique system identifier has an effect of permitting flexible binding of names to actual events. It is sometimes necessary to shift the viewpoint on a system to a "lower" level of primitive event. The capability in *EDL* to easily rebind primitive events facilitates this type of rebinding of event to event generator.

The **with** clause part names any attributes that the primitive event might possess. The time and location attributes are predefined and occupy the last two event tuple attribute slots. However, there are no code strings associated with the attributes. Event attributes result from action by the

event *CreateHyp* **is primitive** "CREATE_HYP"
with
hypname:string
end

(*CreateHyp*, *hypname*, *time*, *location*)

```
(CREATE_HYP "h:09:0034" 39:10 "VAX9")
(CREATE_HYP "h:02:0173" 59:32 "VAX2")
```

Figure 5: Primitive description, event template, and instances

event generating annotations attached to the system under study. Attribute values are derived from the execution environment available when the primitive event generating annotation is encountered. There is currently no mechanism that helps to specify how to bind environment values to attributes (although work is underway to do this).

The *EDL* primitive event definition thus defines the tuple for a characteristic primitive event and provides a handle to match it with system generated events. Figure 5 shows the relationship between an *EDL* primitive description, the template for the event tuple, and a few instances of the primitive event. All that is necessary to use this primitive event is that the system generate some event tuple which matches the event definition.

5. A Complete Example

Now that all of the parts of a high-level event description have been explained it might be useful to illustrate many of the views of an event definition. A complete *FrontEndCompletion* event might be described as in figure 6. This description represents the event template

(`FrontEndCompletion` *hypcount elapsedtime time location*) .

The implicit attributes *time* and *location* are always present and are part of every event tuple. Some example instances of this event could be

```
(FrontEndCompletion 25 371 2:33:03.25 "vax9")
```

and

```
event FrontEndCompletion( atnode:location ) is
    InstantiateKsi[1] o CreateHyp* o InstantiateKsi[2] o InvokeKs
cond
    InvokeKs.node ==atnode;
        InstantiateKsi[1].location ==InstantiateKsi[2].location
with
    hypcount:integer := numberof(CreateHyp);
    elapsedtime:time := InvokeKs.time -InstantiateKsi[1].time
end
```

Figure 6: Complete *FrontEndCompletion* event definition

```
(FrontEndCompletion 4 54 0:18:04.13 "vax6")
```

If the observed event stream contained a sequence of events

```
(InstantiateKsi . . . 0:4:10.0 "vax7")
(InstantiateKsi . . . 0:4:9.0 "vax6")†
(InvokeKs . . . 0:4:12.0 "vax7")
(CreateHyp . . . 0:4:13.0 "vax7")
(InvokeKs . . . 0:4:14.0 "vax6")
(CreateHyp . . . 0:4:16.0 "vax6")†
(CreateHyp . . . 0:4:17.0 "vax7")
(InstantiateKsi . . . 0:4:22.0 "vax6")†
(InvokeKs . . . 0:4:25.0 "vax6")†
```

then an instance of the *FrontEndCompletion* event derived from this stream would have the events marked with † as constituents. The instantiated *FrontEndCompletion* event could be

```
(FrontEndCompletion 1 0:0:15.0 0:4:25.0 "vax6")
```

The *FrontEndCompletion* event can also be viewed in terms of its structure, which is derived from its constituent events, their constituents, etc. Since self-referent event descriptors are not allowed, the structure defines a tree with the given high-level event as the root. If the *FrontEndCompletion* event is incorporated into a higher level event, *FirstSigActivity* with the definition of figure 7 the structural representation of *FirstSigActivity* would be as shown in figure 8.

This structural view is useful when trying to understand the overall structure of an event based behavior description. With proper annotation this representation might make it easier to judge how well this abstraction matches the actual system activity. Also, the structural view aids in analyzing certain difficulties with the abstraction process itself.

```
event FirstSigActivity( atnode:location ) is
    FrontEndCompletion o (ReceiveHyp | CreateHyp)
cond
    CreateHyp.node ==atnode;
    ReceiveHyp.node ==atnode;
    FrontEndCompletion.node ==atnode
end
```

Figure 7: A higher level event

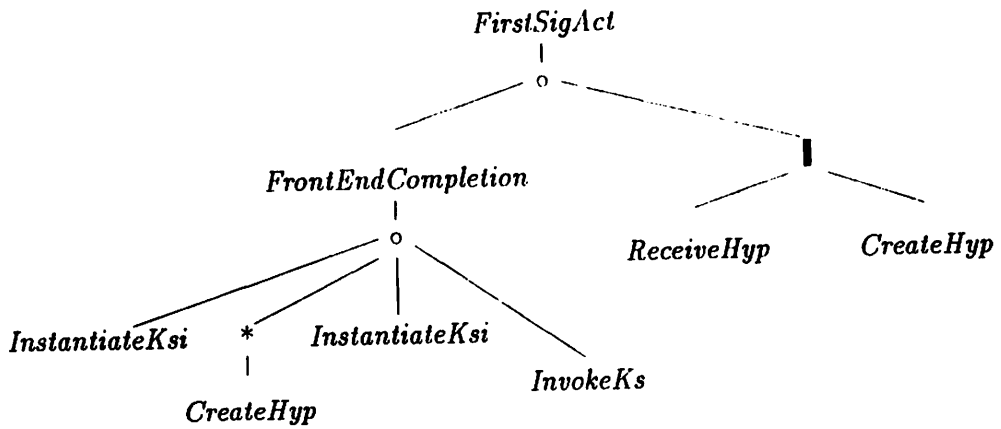

Figure 8: Structural representation of a high-level event

A. Using the Model builder

The user should put the location of the *EBBA* toolset into the search path environment variable. Barring that level of sophistication, after it is determined where the executable form of the model builder resides, it is handy to create an alias (`edl`) to be used to invoke it.

`edl [-o] [-t] [-l` *libname*`] [-m` *libname*`] [-e` *event*`] [-D`*macro-definition*`]`
`[-U`*macro-name*` [-I`*include-file-directory*`] [`*file1 file2 ...*`]`

file1, *file2*, ... are text files containing *EDL* definitions and preprocessor directives that are to be translated into recognizers or pending event descriptors. If no files *file1*, *file2*, ... are specified, then the model builder has been invoked in interactive mode and will attempt to read from standard input, which can be attached to almost anything. The indicated files are read in the order they are given on the invocation line and the events they contained are processed in the order they appear in the source files.

The `-l` directive specifies the event library (or librarian) that maintains the viewpoint that the compiled events are related to. The *libname* parameter names a previously created event definition library (or one about to be created). The model builder will access the library indicated by the `-l` option, then use the definitions contained in *file1*, *file2* ... to edit the opened library. If no library is specified (no `-l` option selected) the library named "`default`" is used.

The `-m` option specifies a library that is to be merged with the main library. Any number of merge libraries may be specified with the proviso that the events in the main and merged libraries do not conflict. Each merged library event will be checked for uniqueness and if no problems arise will be added to the main library. More than one merge library may be specified with multiple `-m` options.

If `-o` is specified, no library outputs will be created by the compiler. Syntax and consistency checking is performed but no new or changed events are written back to the event library. The base library for the compiled events must be available to allow allow examination of high-level event constituents.

The `-e` *event* option indicates that only the named *event* is to be translated. All events in the file list should be syntactically correct, since selection for translation is performed after parsing of individual event models (and the selected event might get caught up in the dumb error recovery technique).

Normally *time* and *location* fields are automatically added to any event tuple that does not explicitly include them. The `-t` option tells the model builder to NOT supply default *time* and *location* fields for events that do not have them specified. Using this option could wreak havoc with `ebbaPostEvent()` calls.

The -D*macro-definition* permits definitions of the form *macro-name=value* to be effective at the start of preprocessing. -U*macro-name* removes the definition of any predefined macro names owned by the preprocessor. -I*include-file-directory* adds *include-file-directory* to the search path for files that are to be included using the `#include` preprocessor directive. Note that there are no spaces between the option letter and the value that is to be supplied.

B. Model Builder outputs

The model builder prints some diagnostic and informative messages as it executes. Less exciting outputs are the translated event descriptions that may be used by the event recognizer to recognize model instances. Also output by the model builder are files that aid primitive event generation.

B.1 Diagnostic and informative messages

The following list of messages is output by the Compiler and written to the standard error stream. Most of them are self-explanatory, and will require that the source text of an event be edited and compiled again. Error recovery is *very* dumb, so innocuous syntax errors can be quite amusing.

`Attempt to redefine formal parameter` *param*
: The named event heading parameter was defined more than one time.

`Attempt to redefine primitive event` *event-name*
: Primitive events may not be redefined. The user should clear the library using the delete files shell command or request the librarian interface for the library delete the the redefined primitive event.

`Duplicate attribute name` *attribute-name*
: An attempt was made to redefine a primitive event attribute.

`Attempt to redefine attribute` *attribute-name*
: An attempt was made to redefine the named high-level event attribute.

`No event number for event` *event-name*
: A request was made to the librarian for an event number for the named event, but none was returned. Either more than 2^{16} high-level events have been defined or there is an internal error.

`Unknown Event` *event-name*
: An event name which has not been defined yet appears as an operand in an event expression.

`Too many/Insufficient parameters to constituent` *event*
: An event expression member is a high-level event with parameters, and the number of actual parameter bindings is incorrect.

`Undeclared local variable` *name* `in expression`
: A simple name that is not declared as an attribute or formal parameter is being used in a **cond** or **with** clause expression.

`Event` *event* `not in event expression.`
: An event name that is not used in the event expression for the event is used to qualify an attribute used as an operand in a **cond** or **with** clause expression.

`Attribute` *name* `not part of event` *event*`.`
: The attribute *name*, qualified by the *event* is not defined by the *event*.

`Invalid numeric text to decode '`*text*`'`
: This is probably an internal error, it indicates that what was once considered to be valid numeric text cannot be translated into a binary form. The text is probably the index of a qualified event.

`Character` *char* `does not begin a token.`
: The indicated character is not a valid prefix for a symbol in the *EDL* language. This error will be followed by zero or more messages regarding recovery attempts.

`Include files nested too deeply, file` *file-name* `ignored.`
: Only 8 levels of source file can be maintained by the compiler (one original plus 7 `#include`'ed files). The indicated file tried to be the 9^{th}.

`Newline before terminator (`*char*`) for string`
: *char* was used to begin a character string, but no matching character was found before the end of the source text line.

`Cannot open include file` *file-name*`.`
: The named file could not be opened for reading, either it does not exist or one of the directories in its file path is inaccessible.

`Unknown option '`*string*`' specified.`
: The command that invoked the compiler contained an unrecognized option, or a previous option did not have a value, but was given one.

`Start state mapping is missing a routine start.`
: This is an internal error. The Shuffle Automata contains a number of sub-machines, one of which has not had its starting state added to the list of machine start states.

`Syntax error near` *token-name.*
: Dumb error recovery in action. There was a syntax error at the token indicated. This error will be followed by zero or more messages regarding recovery attempts.

`Unexpected End-of-File encountered.`
: The end of the source text was found before the current event definition was completed.

`- Token` *token-name* `deleted from input.`
: More dumb error recovery. When a syntax error is encountered, symbols are deleted from the input stream until something the compiler likes better comes along.

`Invalid ASTnode op` *number.*
: This is an internal error. The abstract syntax tree constructed to represent an event has an unrecognized value in its node type field.

`Attempt to insert duplicate symboltable entry for` *name.*
: This is an internal error. A routine attempted to add a name to the local symbol table for an event without checking to see if it was already there.

`Unknown node type in ConstEvalExpr.`
: This is an internal error. An arithmetic or relational expression has an invalid node type in its abstract syntax tree.

B.2 The Real Compiler Outputs

The compiler reads event source text and events stored in libraries and outputs libraries and a file of primitive event descriptions. The files created as a result of a compilation of a collection of primitive and high-level event descriptions include the following:

- a library directory that is located in the library root directory given by either the environment variable `ELIBROOT` or a subdirectory, ~/`EventLib`, located in the user's home directory. This file will have the name ~/`EventLib/`*lib*.
- a library information file *lib*`.IFO` that holds the library status (high-level and primitive event-number servers, counts, etc....) The name for this file will be ~/`EventLib/`*lib*`/`*lib*`.IFO`.
- a primitive event formatting file *lib*`.h` that contains descriptors for the primitive events. This file is used by the default primitive event generator in *libBA* to put together event tuples. The final name for this file is ~/`EventLib/`*lib*`/`*lib*`.h`
- as many files of the form *nnnnn*`.EVT` as are needed for the translated event descriptors. *nnnnn* is $1 \ldots 2^{16} - 1$ for primitive events; and $2^{16} \ldots 2^{\texttt{MAX EventNumber}}$ for high-level events. (`MAX_EventNumber` can be found in the `debug/include/params.h` file if the reader is really interested. These files will have the final name ~/`EventLib/`*lib*`/`*nnnnn*`.EVT` There may be more high-level event files than there are high-level events since each time a high-level event is compiled a new event number is issued for it. The librarian utility (`lu` or the librarian/model-builder interface) or the delete files command can be used to clean up those unwanted event files.

The details of the file contents are found in another report

P.C. Bates, "The EBBA Toolset Implementation Notes"

Also, detailed descriptions of the compiler internals and operation are found in that report.

C. BNF Description of EDL

event_defs	::=	event_description | event_defs event_description
event_description	::=	**event** event_heading is_clause cond_clause with_clause **end**
event_heading	::=	identifier | identifier (arglist)
arglist	::=	formal_id | arglist , formal_id
formal_id	::=	identifier:type
is_clause	::=	**is** event_expression
event_expression	::=	re_expr | **primitive** value
re_expr	::=	re_sexpr | re_expr | re_sexpr
re_sexpr	::=	re_term | re_sexpr ∘ re_term
re_term	::=	re_factor | re_term ∧ re_factor
re_factor	::=	constituent_event | re_factor repetition | (re_expr)
constituent_event	::=	identifier elist event_index
elist	::=	(expr_list) | ϵ
event_index	::=	[number] | [identifier] | ϵ
repetition	::=	* | $^+$
with_clause	::=	**with** attribute_list | ϵ
attribute_list	::=	attribute | attribute_list ; attribute
attribute	::=	attribute_name | attribute_name := expression

attribute_name ::= identifier:type

expression ::= primary
| - expression
| ! expression
| ~ expression
| expression binop expression

primary ::= value
| qualified_name
| identifier
| (expression)
| identifier (expr_list)

expr_list ::= expression
| expr_list , expression

value ::= number | string

binop ::= * | / | % | + | - | << | >> | <
| <= | >= | > | == | != | & | | | && | ||

qualified_name ::= identifier event_index . identifier

cond_clause ::= **cond** boolean_exprlist | (

boolean_exprlist ::= expression
| boolean_exprlist ; expression

string ::= " characters "

characters ::= character | characters character

character ::= ASCII-character
| \decimal

number ::= bin_num | oct_num | decimal | hex_num | char_num | real

bin_num ::= **0b** bin_string

bin_string ::= bin_digit
| bin_digit bin_string

bin_digit ::= **0 | 1**

oct_num ::= **0o** oct_string

oct_string ::= oct_digit
| oct_digit oct_string

oct_digit ::= **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7**

decimal ::= dec_digit
| dec_digit decimal

dec_digit ::= **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

hex_num ::= **0x** hex_string

hex_string ::= hex_digit
| hex_digit hex_string

hex_digit ::= **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7**
| 8 | 9 | a | b | c | d | e | f

type ::= `integer | double | time`
`| string | location`

char_num ::= 'character'