Event Defintion Language:
An Aid to Monitoring and Debugging Complex Software Systems

Peter C. Bates* and Jack C. Wileden**

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

COINS Technical Report 81-17

*Supported in part by the National Science Foundation under Grant MCS-8006327 and by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract NR049-041.

**Supported in part by the National Aeronautics and Space Administration under grant NAG1-115.

Abstract

As part of a broader study of strategies for distributed computation we have begun developing tools to support the debugging of distributed systems. Our approach to distributed system debugging, called behavioral abstraction, and its realization in the Event Definition Language (EDL) are described in this paper. We have also begun an implementation of a debugging system supporting the EDL and providing rudimentary information gathering, information presentation and intervention capabilities. These aspects of our work are discussed briefly in the paper.

1.0 INTRODUCTION

As part of a broader study of strategies for distributed computation [Less80] we have begun developing tools to support the debugging of distributed systems. Debugging tools have traditionally been among the utilities provided by most operating systems. However, these tools (e.g., [Satt75], [VanT78], [Digi81]) have typically offered only limited, low-level debugging aid suitable for debugging sequential programs. Even with these aids the task of developing an understanding and keeping a consistent model of what a system is doing, which is the hardest part of locating non-trivial errors, is left to the user.

With the use of tens or hundreds of cooperating processors on problem solving or control applications some new difficulties in providing working, and preferably error-free, systems arise. Foremost among these are problems caused by <u>complexity</u> and <u>concurrency</u>. Complexity has always been a major obstacle to understanding a program's behavior. However, certain facets of distributed computation make the complexity obstacle even less tractable than it was previously. For example, the sheer size of the software component of a system of tens or hundreds of processors often result in tremendous complexity making it practically unmanageable. Furthermore, the use of a non-homogeneous set of processors will require many interpretations for similar types of behavior on the differing processors. Similarly, distributed systems amplify the problems posed by concurrency. With the increase in the number of processors, the number of truly concurrent events that are possible increases. Using a network of many processors will require exchanging complete or partial results among them. The problems introduced by synchronization, bandwidth constraints and errorful data in the communications pathways will be difficult to contend with using current debugging technology.

As a first step in meeting the challenge of debugging distributed systems, we have developed an approach for coping with the complexity and concurrency of distributed systems. This approach, called behavioral abstraction, and its realization in the Event Definition Language (EDL) are described in this paper. We have also begun an implementation of a debugging system supporting the EDL and providing rudimentary information gathering, information presentation and intervention capabilities. We discuss this aspect of our work and our expectations for future work on this project at the end of this paper.

2.0 BEHAVIORAL ABSTRACTION - WHAT AND WHY

It is our belief that debugging complex distributed systems fundamentally requires the ability to observe particular aspects of the system's detailed activity from a suitably abstract perspective. Such selective observation would permit a user to focus on suspected problem areas without being overwhelmed by all of the details of system activity, thus offering some hope that causes of failure may be located.

Our approach to selective observation is what we call behavioral abstraction. It is based upon considering a system's activity as consisting of a stream of event occurrences. Behavioral abstraction results from the ability to define a particular viewpoint, or window, on that event stream. A viewpoint is defined by filtering and clustering events from the stream. Filtering the stream deletes all but a designated subset of events from the stream. This serves to highlight those aspects of system activity that are currently of interest to the user. Clustering events treats a designated sequence of events as constituting a single higher-level event. This provides a means of obtaining an abstract view of system activity.

Using the clustering and filtering techniques a window on the event stream can be constructed that gives a view of the system relevant to the particular monitoring task being performed. The higher-level events created through clustering may themselves be incorporated into subsequent event clustering definitions. By repeatedly using clustering to build higher level events and then using these new events to create a still higher level view, a set of abstractions of the system can be obtained that will allow an observer to view the system at various levels as well as observe specific kinds of behaviors. Filtering of the event stream removes those event instances that are not considered relevant to the monitoring task being performed. Filtering is accomplished by considering the specific properties of each particular occurrence of an event. Depending upon those specific properties, a given occurrence may or may not be judged relevant to the particular viewpoint being defined. By employing an appropriate behavioral abstraction, the developer of a complex distributed software system can monitor those aspects of the system's behavior that are relevant to the specific questions presently under investigation without being distracted by other, less relevant details of the system's behavior.

Naturally, the particular behavioral abstraction that will be appropriate when searching for a given failure will vary, and no behavioral abstraction can be expected to be appropriate for all problems. Therefore, our approach is founded upon a flexible mechanism for defining behavioral abstractions. This mechanism is embodied in a language called the Event Definition Language or EDL. Using EDL, a user can specify the particular high level viewpoint on detailed system activity that seems suitable for understanding a particular problem with a distributed system's behavior. In the next two sections we outline a few fundamental EDL concepts and then survey some related work. Section five is a more detailed explaination of the EDL notation, followed by an example to illustrate its use. Issues concerning the implementation of EDL as a tool to aid in the interactive debugging of distributed systems are briefly considered in the final section of this paper.

3.0 EDL - A MECHANISM FOR BEHAVIORAL ABSTRACTION

The Event Definition Language provides users with a means of both filtering and clustering a system's event stream to obtain a behavioral abstraction. As its name suggests, EDL supports these capabilities by allowing the user to define events. Event definitions in EDL are formulated by combining previously defined events using a set of event formation operators (clustering) and by stipulating the properties of the constituent events (filtering). We discuss these operations in more detail below. This constructive approach to viewpoint definition depends upon the existence of an initial set of events from which additional events can be constructed. We refer to this set of events as primitive events. The primitive event set for a given system is a characteristic feature of that system and determines the lowest level, most detailed, view of the system that can be obtained. Experience supports our belief that, in general, the number of distinct primitive event types for a given system will be small.

Given a collection of previously defined events, which may be primitive or the result of clustering, the features of the Event Definition Language can be used to define new events in terms of those already defined. This serves to give the user a different viewpoint on the system's activity, seeing it in terms of the newly defined events rather than their constituents.

An EDL event definition does not actually describe a specific individual event, but rather an entire event class or type of event. A specific individual occurrence of an event from some event class is referred to as an instance of that event class. Different instances of event classes are distinguished by a set of attributes that each instance of the class possesses. Depending upon its particular attributes, a given instance of an event may or may not be relevant to a given viewpoint or system behavior as defined by (in general, higher level) EDL event definitions.

4.0 RELATED WORK

The expression-based description mechanism underlying EDL is related to a variety of similar techniques, including Campbell's Path Expressions [Camp74], Shaw's Flow Expressions [Shaw78], Riddle's Event Expressions [Ridd79] and Wileden's Constrained Expressions [Wile78]. (See [Shaw80] for a survey of specification languages based on regular expressions.) All of these formalisms, as well as languages based on Petri Nets [Pete77], have been used in describing and studying concurrent systems. They differ from each other, and from EDL, primarily in the details of their notation and in their intended application.

Path Expressions were originally developed as a synchronization technique for defining coordination and cooperation among processes. Using the Path Expression notation, sequencing and concurrency constraints can be expressed for sets of procedures that have access to common objects. Flow Expressions and Event Expressions are both intended for use as modeling tools, describing the expected or desired behaviors for a system under study. An important goal of both of these notations is to support formal analyses that will demonstrate various properties of the system, defined by a given set of expressions. The use of the Petri net formalism for modeling makes it similar in application to that of the flow and event expression notations.

While all of the notations mentioned above are similar in their formal properties, the intended applications for which they have been created have influenced their descriptive capabilities, ease of use and applicablity to other domains. The EDL notation has been specifically created to allow us to guide recognition of certain 'interesting' events in a system, based upon both their ordering relations and relations among the characteristics that the events exhibit. For this reason, EDL enhances the expression based notation with capabilities for associating attributes with symbols as part of the recognition process. These features and their use are discussed and illustrated in the next two sections.

A framework for debugging based upon viewing program executions as a series of event occurrences was previously suggested by Schwartz [Schw70]. However, events in this case were the results of normal, low level program activities such as variable settings and procedure calls. Hence, events in the Schwartz approach were essentially states that the program might achieve. In particular, Schwartz did not employ the event viewpoint as a mechanism for constructing abstractions of system behavior. STABDUMP [McGr80], a debugging system based on Schwartz' ideas, accumulates a trace of program execution and allows a user to conduct a post-mortem search for patterns of execution history and variable bindings in an attempt to understand and locate errors in the program. Once an interesting point in the history is located, variables may be changed and the program started up from this point, with the new history.

5.0 AN OVERVIEW OF THE EVENT DEFINITION LANGUAGE

An EDL event definition describes how an instance of an event might occur and what the attributes of the instance will be if it does occur. Each event definition is composed from a heading and three types of defining clause: the 'is' clause, which defines an event expression over previously defined events; the 'cond' clause, which places constraints on the events mentioned in the 'is' clause; and the 'with' clause, which defines a set of attributes that each instance of the event class will have. (The syntax of EDL is detailed in Appendix A.)

The event heading of an event definition associates a name with the event class being defined. The name is the means by which the event class is known and referred to in the system. An optional parameter list provides a means of creating generic event descriptions that may apply in different contexts with suitable argument bindings. Actual parameters are substituted textually (macro parameter substitution) when the event name appears in an event expression. For example, the following event heading:

```
event login( port_number )
```

introduces a definition for an event named 'login'. There is a single parameter, 'port_number', which may be used in the definition body wherever it is valid to use an identifier.

The 'is' clause introduces a regular expression over event classes that occur in the system. We refer to the 'is' clause regular expression as an <u>event expression</u>. An event of the class given by the event name occurs when a string of events occurring in the system matches that described by the event expression. The event expression is composed from event class names, either primitive or previously defined, and operators indicating alternative ways to form strings that are acceptable for this event definition. The operators consist of the normal set of formation operators for regular expressions (our notation differs only slightly from the standard regular expression notation) with the addition of a shuffle operator to indicate concurrency [Ridd79, Shaw78, Gins66]. The event expression is the means for describing aggregates of events and provides the clustering capability necessary for abstraction of system behavior.

The catenation operator "'" specifies that an event follows another. As an example consider the following partial event definition:

```
event login( portnumber ) is
  portaccess ' processcreation
  :
end
```

A login event is determined by the occurrence of a 'portaccess' event followed by a 'processcreation' event. (Note that the event instances selected from the system event string as constituents of the defined event will be subjected to any applicable constraints imposed by the 'cond' clause.)

The shuffle operator '#' indicates that its operand events occur with no preferred ordering between them, opening up the opportunity for the interleaving of the constituent events. All of the events connected by the shuffle operator must occur, but the order of occurrence of the events is not relevant.

In a similar fashion, alternation, denoted "¦", indicates that occurrence of any one of its operand events is an acceptable string for this operator.

Two unary repetition operators are used to indicate a possibly unbounded sequence of their operand event. Both the star '*' and plus '+' operators are left associative. The plus operator indicates that one or more occurrences of the operand are needed. Star is the closure of plus with zero or more occurrences being a valid string. For example, the (partial) event definition:

```
event link_error( port ) is
  line_allocatation '
    (connect_attempt)* ' line_hangup
  :
end
```

defines a class of events named 'link_error' related to a failure to establish a communications link through a port in a communications device. Here the clustered event consists of the allocation of a line and a series of (possibly zero) attempts to establish the connection followed by a releasing of the line previously allocated. The argument 'port' would be used to indicate a specific port attached to the communication device. This definition could now be used to detect an event resulting in a form of isolation of a node from the subnet:

```
  event node_loss is
    (link_error( boston ) ¦
            link_error( portland ))
    :
  end
```

An event class may be used in an event expression more than one time. An event index provides a local (to the event definition) qualifier that will distinguish different mentions of the same event class in the event expression. For example, in the partial definition:

```
  event paired_error( nodes ) is
    node_error.1 # node_error.2
      :
  end
```

the indexing convention will distinguish the two instances of 'node_error' which are necessary for an instance of the 'paired_error' event. The importance of this capability is illustrated in the example given in the next section.

The 'with' clause of an event definition introduces the names for the attributes of the event being defined and indicates how to determine values for those attributes when an instance occurs. The operands of the expressions are taken from the attributes bound to instances of event expression constituents and any attributes local to the event being defined.

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.

When an event occurs, each attribute name defined in the 'with' clause is bound to a value determined by its defining expression. Expressions are composed of the usual relational, arithmetic and logical operators using operands supplied by local (defined in the enclosing event definition) or qualified attributes. The event definition:

```
event mixer_empty is
  pressure_low_warning ' output_valve_shut

with
  quickness := output_valve_shut.time
                 - pressure_low_warning.time;
  mix_station_type :=
             output_valve_shut.id mod 100;
  mix_station := output_valve_shut.id div 100
end
```

defines three attributes for each instance of 'mixer_empty': 'quickness', 'mix_station_type' and 'mix_station'. Each of these attributes is defined in terms of attributes bound to the event instances of the event expression constituents. Specifically, 'mixer_empty.quickness' is defined in terms of the 'time' attributes of the events 'output_valve_shut' and 'pressure_low_warning' while the 'mix_station_type' and 'mix_station' attributes of the 'mixer_empty' event are defined in terms of the 'id' attribute of the 'output_valve_shut' event.

Qualified attributes must be mentioned in the 'with' clause of the qualifying event name. A form of scoping rule for qualified names is effected in the following manner: an event may only examine the attributes of events it explicitly names in its event expression.

Consider an event Z defined as follows:

```
event Z is X'(Y # Q)'F
   :
end
```

with event Q defined as:

```
event Q is N'M¦J'G
   :
end
```

then Z can see the attributes of X, Y, Q and F but not the attributes of N, M, J and G. This latter group can be made visible, if necessary, by adding more attributes to the list defined for Q and simply passing them on.

The 'cond' clause defines a set of relational expressions over the attributes of the event expression constituent events. These relationals place constraints on the attributes of events that appear in the event expression. This creates the previously mentioned filtering effect by allowing events having only certain characteristics to be considered for inclusion in the event expression of the definition. For example, the 'login' event mentioned previously is only valid if the process creation is related to the port that has been accessed, and the port was inactive at the time. These constraints might be expressed as follows:

```
 event login( port ) is
   lineaccess ' process_creation

 cond
   line_access.state = "unallocated";
   line_access.multiplexor_port = port;
   process_creation.input_device =
              line_access.multiplexor_port
   :
 end
```

A 'cond' clause does not need to be present in an event description. In the absence of a 'cond' clause any set of events of the right class in the order prescribed by the event operators will constitute an instance of the defined event. The 'cond' clause serves to narrow the focus of the event being described by allowing only events having certain attributes to be constituents of the defined event. Several examples of 'cond' clause usage appear in the following example.

6.0 AN EXAMPLE

In this example, three event definitions are constructed as a means of developing a high-level abstraction for what may be a serious failure among a group of four cooperating nodes. These definitions could provide an appropriate viewpoint for a user attempting to debug a distributed system with a certain kind of faulty behavior.

The first definition, 'paired_error' is simply an event that occurs if an error occurs in two adjacent nodes within a certain time period. It is assumed that the topology of the nodes is a ring structure. The serious failure would be the loss of the communications link between two adjacent nodes. It is further assumed that the only type of error that is detectable (the primitive event 'node_error') is related to the maintenance of the communications link. When the event occurs, it has the attribute 'id' serving to identify the node pair between which the error has occurred.

```
event paired_error( nodes, epsilon ) is
  node_error.1 # node_error.2

cond
  node_error.1.id = nodes;
  node_error.2.id = (nodes + 1) mod 4;

    abs(node_error.1.time
            - node_error.2.time) > epsilon

with
  id = nodes

end
```

The event expression stipulates that two errors must occur, but that their order is irrelevant. In fact, the two errors might occur simultaneously. The 'cond' clause relations specify that the instances acceptable for the event expression must be from adjacent nodes. Further, the maximum time delay is a parameter of the event definition and hence various invocations of the event may have different time delay properties. The event indexing is necessary here for an unambiguous statement of the conditions insuring that the instances used to satisfy the event expression are not from the same node or from pairs of nodes not connected with a link.

Using this simple definition a single event class is created that will indicate an error in any of the four pairs of nodes. This definition exists mostly as a shorthand notation to indicate any 'paired_error' and its source. (Note, however, that different maximum time delays are used for the various node pairs.) This will greatly simplify the event expression in a more interesting definition to follow. The inclusion of the 'id' attribute in this definition recalls the scoping rules mentioned earlier. Without it, the scoping rules would prevent any definition using the 'multi_error' in its event expression from examining the 'id' attribute of the 'paired_error' responsible for the 'multi_error'.

```
event multi_error is
  paired_error(0, 3) ¦ paired_error(1, 7)
      ¦ paired_error(2, 2) ¦ paired_error(3, 18)
with
  id := paired_error.id
end
```

The occurrence of the 'multi_error' may not be serious or even interesting in the absence of other conditions. The following definition captures what may be a serious failure in the link between two nodes.

```
event big_error( threshold ) is
  multi_error.1 '
        restart_attempt+ ' multi_error.2
cond
  multi_error.1.id = multi_error.2.id;
  abs(multi_error.1.time
                  - multi_error.2.time) < threshold
  with
    location := multi_error.1.id;
    severity := errorestimate( threshold )
  end
```

Briefly, 'big_error' is defined as an error in a link followed by a number of attempts to reestablish the link (at least one is necessary) and a subsequent error on the same link. It is assumed that the 'restart_attempt' event is either primitive or previously defined. The 'cond' clause expressions insure that the identity of the erroneous link is the same in both constituent 'multi_error' events and that no more than a designated amount of time elapses between the two errors. Bindings to the attributes 'location' and 'severity' when the 'big_error' occurs would allow an observer to determine both the location and the approximate severity of the failure.

7.0 SUMMARY AND CONCLUSIONS

We have given an overview of some of the difficulties involved in debugging distributed software systems and have described the approach we are taking to overcome some of those difficulties. We have outlined the concept of behavioral abstraction and described the Event Definition Language, which is intended to provide a debugging tool supporting behavioral abstraction, as illustrated by the preceding example.

Naturally the language alone is of limited help in debugging. To be truly useful, EDL must be implemented as part of an interactive debugging facility for distributed systems. Such an implementation must support the detection of occurrences of primitive events and provide capabilities for monitoring the stream of events occurring throughout a distributed system. It must also accept EDL event definitions and be capable of discerning when events defined by a user in EDL have occurred. Further, it must be able to display system activity, in terms of the current perspective specified by the user via EDL, in a convenient and comprehensible format (perhaps using color graphics). Finally, it must provide the user with the ability to intervene in the distributed system's operation at any time. Given these capabilities, a user could observe the activity of a distributed system from any desired perspective, watching for the occurrence of particular events or event sequences, such as those described in the example of the previous section, then intervene to gather further information or to interactively alter the system's activity. This would greatly facilitate the debugging of distributed systems.

We are currently working toward the implementation of a debugging facility providing just these capabilities as part of our broader study of strategies for distributed computation. In addition to the definition of EDL, a high-level design for the monitoring and intervention facility has been completed. Detailed design and implementation are currently underway. We anticipate that a prototype version of an EDL-based distributed system debugging utility will be operational early in 1982. Future papers will describe the design and implementation of the facility and report on our experience in using behavioral abstraction as an aid for debugging distributed systems.

We believe that EDL provides a valuable means for describing interesting and useful viewpoints on a distributed system. Examples, such as the one presented in this paper, reinforce our confidence that most abstract perspectives on a distributed system's behavior that would be of interest in debugging can be succinctly and comprehensibly described with EDL. Nevertheless, we anticipate that additional descriptive capabilities will prove useful, and we therefore view the EDL presented in this paper as a first version. In particular, we have yet to settle on an acceptable notation for describing 'negative' events, such as "no event x occurs between events y and z". In some specific cases, negative events can be descibed quite easily using our present EDL, but other times the descriptions are complicated and confusing at best. This difficulty, and others that will doubtless be uncovered once our prototype debugging utility becomes available, are among the topics that we expect to address in future work on this project.

Acknowledgements. We wish to thank Victor Lesser for providing the original motivation for our study of distributed system debugging and for numerous contributions to our work on behavioral abstraction. We are also happy to acknowledge Roger Thompson who worked on the preliminary syntax definition for EDL.

Bibliography

[Camp74] R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," In G. Goos and K. Hartmanis (ed.), Lecture Notes in Computer Science, Vol. 16 Springer-Verlag, Berlin, 1974.

[Digi81] VAX-11 Symbolic Debugger Reference Manual, Digital Equipment Corporation, Maynard, Massachusetts, 1981.

[Gins66] S. Ginsburg, The Mathematical Theory of Context-Free Languages, McGraw-Hill, New York, 1966.

[Less80] V.R. Lesser, P. Bates, R. Brooks, D. Corkill, L. Lefkowitz, R. Mukunda, J. Pavlin, S. Reed and J. C. Wileden, "A High Level Simulation Testbed for Cooperative Distrubuted Problem Solving," Technical Report TR-81-16, Department of Computer and Information Sciences, University of Massachusetts, 1981.

[McGr80] D. R. McGregor and J. R. Malone, "STABDUMP - A Dump Interpreter Program to Assist Debugging," Software - Practice and Experience, Vol. 10, pp 309-332, John Wiley, 1980.

[Pete77] J. L. Peterson, "Petri Nets," Computing Surveys, Vol. 9, 3, September 1977.

[Ridd79] W. E. Riddle, "An Approach to Software System Behavior Description," Computer Languages, Vol. 4, pp. 29 to 47, Pergamon Press Ltd., 1979.

[Satt75] E. H. Satterthwaite, "Source Language Debugging Tools," Technical Report STAN-CS-75-494, Computer Science Department, Stanford University, May 1975.

[Schw70] J. T. Schwartz, "An Overview of Bugs," in R. Rustin (ed.), Debugging Techniques in Large Systems, Courant Computer Science Symposium 1, Prentice-Hall, 1971.

[Shaw78] A. C. Shaw, "Software descriptions with Flow Expressions'", IEEE Transactions on Software Engineering, SE-4, 3, May 1978.

[Shaw80] A. C. Shaw, "Software Specification Languages Based on Regular Expressions," in W. E. Riddle and R. E. Fairley (ed.), Software Development Tools, Springer-Verlag, Berlin, 1980.

[VanT78] D. Van Tassel, Program Style, Design, Efficiency, Debugging and Testing, Prentice-Hall, Englewood Cliffs, New Jersey, 1978

[Wile78] J. Wileden, "Techniques for Modelling Parallel Systems with Dynamic Structure," Technical Report TR-78-4, Department of Computer and Information Sciences, University of Massachusetts, 1978.

APPENDIX A

```
<event_description> ::=
        event <event_heading>
        <is_clause>
        <cond_clause>
        <with_clause>
        end

<event_heading> ::=
        <event_name><param_list> ¦
        <event_name>

<event_name> ::= <id>
<id> ::=
        <letter> ¦ <id><letter> ¦
        <id><digit> ¦ <id><underscore>

<is_clause> ::= is <event_expression> ;
<event_expression> ::= <re_expr> ¦ primitive

<re_expr> ::=
        <re_sexpr> ¦
        <re_expr> <alternation> <re_sexpr>

<re_sexpr> ::=
        <re_term> ¦
        <re_sexpr> <catenation> <re_term>

<re_term> ::=
        <re_factor> ¦
        <re_term> <shuffle> <re_factor>

<re_factor> ::=
        <event_name><event_index> ¦
        (<re_expr>) ¦
        <re_factor><repetition>

<event_index> ::= .<number> ¦ <empty>

<catenation> ::= '
<shuffle> ::=
<alternation> ::= ¦
<repetition> ::= * ¦ +
<with_clause> ::= with <attribute_list> ¦ <empty>
<attribute_list> ::=
        <attribute> ¦
        <attribute> ;  <attribute_list>

<attribute> ::= <attribute_name> := <expression>

<expression> ::= <simple_expression> ¦
        <simpleexpression> <relop> <expression>

<simple_expression> ::=
        <term> ¦
        <term> <addop> <simple_expression> ¦
        <sign> <term>

<term> ::=
        <factor> ¦
        <factor> <mulop> <term>

<factor> ::=
        ( <expression> ) ¦
        not <factor> ¦
        <qualified_name> ¦
        <value>

<value> ::= <number> ¦ <string> ¦ <boolean>
<boolean> ::= true ¦ false
<relop> ::= < ¦ <= ¦ = ¦ <> ¦ >= ¦ >
<mulop> ::= and ¦ / ¦ * ¦ mod
<addop> ::= or ¦ + ¦ -
<sign> ::= + ¦ -

<qualified_name> ::=
        <qualified_event_name><dot><attribute_name>

<dot> ::= .
<qualified_event_name> ::=
        <event_name><event_index> ¦
        <event_name>

<cond_clause> ::= cond <boolean_exprlist> ¦ <empty>
<boolean_exprlist> ::=
        <boolean_expr> ¦
        <boolean_expr>;  <boolean_exprlist>

<boolean_expr> ::= <qualified_name> <relop> <expression>
```