83-29

High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach

Peter C. Bates[1]
Jack C. Wileden[2]

University of Massachusetts
Amherst, Massachusetts

ABSTRACT

Most extant debugging aids force their users to think about errors in programs from a low-level, unit-at-a-time perspective. Such a perspective is inadequate for debugging large complex systems, particularly distributed systems. In this paper, we present a high-level approach to debugging that offers an alternative to the traditional techniques. We describe a language, EDL, developed to support this high-level approach to debugging and outline a set of tools that has been constructed to effect this approach. The paper includes an example illustrating the approach and discusses a number of problems encountered while developing these debugging tools.

---

[1] 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.

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

1.0 INTRODUCTION

Debugging of computer software is often a frustrating and difficult experience for software designers and implementors. This is due in part to the tools and techniques which are available for debugging. In order to keep pace with the increasing complexity of computer systems and the tasks they are required to perform, the emphasis in software engineering has been on improving the quality of software and on improving the ease with which software systems are specified. Debugging technology, which must also address the complexity of a system, albeit from a different perspective, has not advanced similarly. Current debugging technology tends to offer users a low-level, unit-at-a-time perspective on a system being debugged (e.g.[1],[2],[3],[4]). This perspective is necessarily detail laden and offers little aid in dealing with complex operational characteristics of a system.

In our work, debugging is viewed as a process of creating models of actual behavior from the activity of a system and comparing these models to the models of expected behavior held by implementors and users of the system. Through these comparisons, debugging tool users attempt to identify sources of errors in the system. Our goal is to facilitate this process by providing tools and techniques for describing abstractions of system behavior and recognizing the occurrence of these abstractions in the system's activity.

In contrast to the traditional, low-level approach to debugging, our "high-level" approach promotes model building and evaluation as its paradigm for debugging. The high-level approach emphasizes machine-based tools to aid user intuition in understanding system behavior. The high-level approach may be further characterized by its attempt to liberate debugging from the detailed programming-object level view of programs, its top-down approach to creating models of system behavior, its provision for creation of multiple viewpoints on a system and the ability to view the system at different levels of

abstraction.

Currently, there is a growing movement towards distributing the functionality of computer systems across local area networks consisting of small to medium size computing elements connected by a high-speed communications subnet [5],[6]. Distributed software systems are harder to debug than centralized systems due to the increased complexity and truly concurrent activity that is possible in these systems. Extant debugging technology is barely adequate for existing programming technology and should not be expected to improve with its extension into the distributed software domain. Our abstraction based approach to debugging originated with the desire to overcome many of the difficulties inherent in debugging distributed software systems and is fundamentally different from past approaches to debugging.

The next section of this paper will briefly discuss the approach to abstraction that we have taken and give a detailed description of the Event Definition Language (EDL) which is used to describe these abstractions. The following section will provide an extended example of the use of EDL in describing a high-level view of system behavior. Section four is a look at several important issues that arise in the construction of tools to recognize behaviors.

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 the considerable volume of detail present in system activity.

Our approach to selective observation is termed *behavioral abstraction*. It is based upon considering a system's activity as consisting of a stream of event occurrences representing significant behaviors of the system. 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 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 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 the causes of 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 the Event Definition Language [7]. 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.

2.1 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 the *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.

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 allows a user to gain a different viewpoint on the system's activity, seeing it in terms of the newly defined events rather than their constituents. Using these viewpoints for debugging assumes the existence of an adequate set of tools supporting the behavioral abstraction approach [8]. A minimal set consists of a

recognizer which uses EDL-created definitions as a guide to interpreting the system event stream, a *librarian* to maintain viewpoints of the system for a user and a *compiler* used to translate EDL definitions into a form suitable to guide the recognizer as well as to check their correctness. We describe such a tool set in section four.

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 higher level EDL event definitions.

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 the Appendix.)

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 parameterizing events and permits users to tailor recognition requests to dynamic conditions. For example, the following event heading:

event login(port)

introduces a definition for an event named "login". There is a single parameter, "port", which may be used as a variable within the event being defined.

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 *event expression* (cf. [9],[10],[11]). An event of the class given by the event name occurs when a sequence of events occurring in the system matches one of those in the set 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 sequences that are acceptable for this event definition. The operators consist of the normal set of formation operators for regular expressions with the addition of a shuffle operator to indicate concurrency [9],[10],[11],[12]. The event expression is the means provided by EDL for describing aggregates of events and hence it 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( port ) is
        port_access ´ process_creation
        :
end
```

A "login" event is determined by the occurrence of a "port_access" event followed by a "process_creation" event.

The shuffle operator "^" indicates that its operand events may occur with no preferred ordering between them. All of the events connected by the shuffle operator must occur, but the order of occurrence of the shuffle operand events is not important (although ordering of each operand's constituents is still subject to the ordering constraints imposed by the operand's defining event expression). Inclusion of the shuffle operator provides the

ability to express concurrency among participating events of the shuffle. Expression of concurrency results from the lack of any implicit time ordering imposed on the operands participating in the shuffle.

In a similar fashion, alternation, denoted "|", indicates that occurrence of *any one* of its operand events is an acceptable sequence for this operator. Both the alternation and shuffle operators are commutative and as a result may denote groups of operands that are to participate in their operations. For example the sequence:

x ´ (a | b | c | d) ´ z

denotes that an "x" event followed by any one of "a", "b", "c" or "d" followed by a "z" will constitute a valid sequence.

Two unary repetition operators are used to indicate a possibly unbounded sequence consisting of repeated occurrences 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 in an instance of the event. Star is the closure of plus with zero or more occurrences being a valid string. For example, the (partial) event definition:

```
event breakin_attempt( port ) is
        port_access ´ (login_failure)+ ´ port_release
    :
end
```

defines a class of events named "breakin_attempt" related to an attempt by possibly unauthorized users to gain access to a dial-up communications port. Here the clustered event consists of the allocation of a line and a series of (at least one) attempts to log on to the system 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 from an attempt to enter the system through a privileged dial-up port:

```
event security_alert is
        breakin_attempt( "diagnostic_port" ) | breakin_attempt( "console" )
        :
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 all_entries_tried is
        breakin_attempt[1]("diagnostic_port") ^ breakin_attempt[2]("console")
                ^ breakin_attempt[3]("mail_delivery_port")
    :
end
```

the indexing convention will distinguish among the three instances of "breakin_attempt" which are necessary for an instance of the "all_entries_tried" 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. 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.

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 (in the style of the programming language C [13]) using operands supplied by local (defined in the enclosing event definition) or qualified (by a constituent event name) attributes. Adding to the previous partial event definition, "breakin_attempt":

```
event breakin_attempt( port ) is
        port_access ' (login_failure)+ ' port_release
with
        port_id = port_access.port_name;
        interval = port_release.time - port_access.time;
        attempts = count( login_failure )
end
```

defines three attributes for each instance of "breakin_attempt": "port_id", "interval" and "attempts". Each of these attributes is defined in terms of attributes bound to the event instances of the event expression constituents. Specifically, "breakin_attempt.interval" is defined in terms of the "time" attributes of the events "port_access" and "port_release" while the "port_id" and "attempts" attributes of the "breakin_attempt" event are defined in terms of the "port_name" attribute of the "port_access" event and a function "count" (defined elsewhere) of the "login_failure" event.

Qualified attributes must be mentioned in the *with* clause of the event definition associated with 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. To make attributes of events visible which are

more than one level of definition away, the intervening events must declare attributes which will serve to simply pass the lower level attributes up to higher levels.

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 as constituents of 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
        port_access ´ process_creation
cond
        port_access.state == "unallocated";
        port_access.multiplexor_port == port;
        process_creation.input_device == port_access.multiplexor_port
    :
end
```

A *cond* clause is an optional part of an event description. In the absence of a *cond* clause any set of events from the classes and in the order prescribed by the event operators will constitute an instance of the defined event. The *cond* clause serves to narrow the scope 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.

3.0 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[1] :

```
event paired_error( nodes, epsilon ) is
        node_error[1] ^ node_error[2]
cond
        node_error[1].id == nodes;
        node_error[2].id == (nodes + 1) % 4;
        abs(node_error[1].time - node_error[2].time) < epsilon
with
        id = nodes
end
```

The event expression indicates 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

[1] "%" is a modulus operator as in the programming language C [13].

instances of the event may have different time delay properties. The event indexing is necessary here to distinguish the two "node_error" events needed to satisfy the event expression. Further, indexing helps provide 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.

```
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
```

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". Indexing the events is not required since only one of the "paired_error" instances will be bound to an instance and therefore there is no need to differentiate between the various mentions in the event expression.

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 = error_estimate( 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 and that the function "error_estimate" is defined elsewhere. 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.

4.0 A PROTOTYPE BEHAVIORAL ABSTRACTION MONITOR

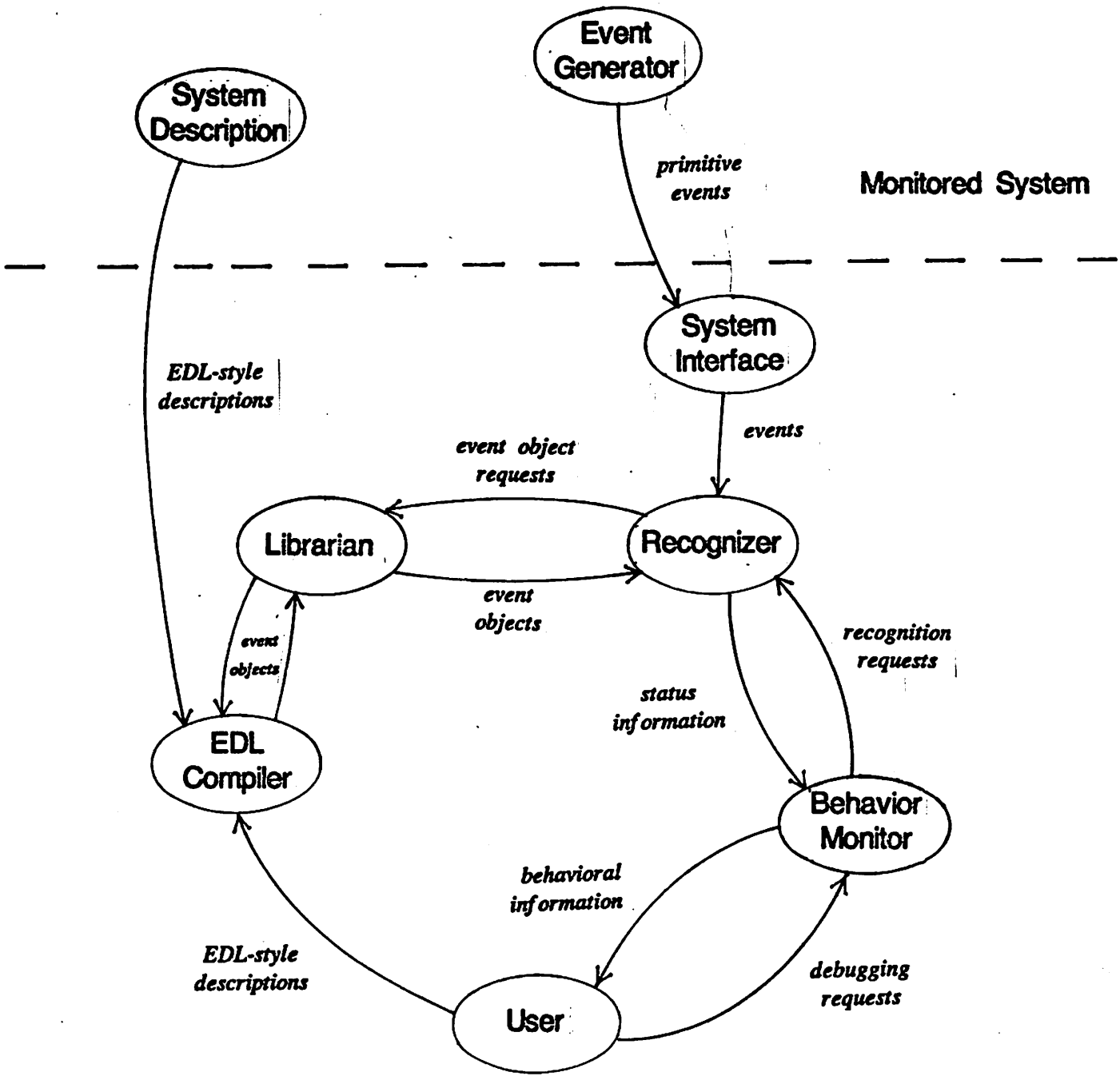
A necessary prerequisite to applying behavioral abstraction and effecting our high-level approach to debugging is providing tools for comparison of actual system behavior to a user's abstracted view of the system. EDL provides a means for defining system abstractions in terms of events and event attributes. To be useful, this mechanism must be integrated with suitable aids to maintain and evaluate these abstractions. A set of debugging tools based on behavioral abstraction permits a tool user to describe significant system behavioral models and to receive feedback on how well these models match the actual behavior of the system. The simplest version of a behavioral abstraction based

debugging tool would simply compare the user models to the actual system behavior. More elaborate and helpful tools would be able to note differences between user models and system behavior and have a rudimentary advisory capability to aid the user in the search for the error sources.

A system to provide the necessary detection capabilities and support for modeling has been constructed based on the design described in this section. Figure 1 is a diagram of the system indicating the major components, their connections and the kinds of information exchanged by the components. It is intended that this debugging tool be capable of being distributed across a network of processors to take advantage of the desirable properties of distributed computing.

4.1 A Set of Tools for Monitoring Behavior

The monitoring system is reasonably independent of the system it monitors. An event based behavioral abstraction approach imposes only a few simple demands on elements external to the monitor. The event traffic that forms the basis for behavior recognition and abstraction is similar to message traffic that is normally found in distributed systems. To integrate the monitoring tools into a system that is to be monitored, a description of an appropriate view of the monitored system written in EDL and a network interface which can translate system event traffic into a form acceptable to the monitoring tools are needed. Additionally, probes must be inserted into the system to detect and report the occurrence of primitive events. In a distributed implementation, the network interface would be responsible for creating and sending messages which report locally recognized events.


Event Generator
System Description
primitive events
Monitored System
System Interface
EDL-style descriptions
events
event object requests
Librarian
Recognizer
event objects
event objects
recognition requests
status information
EDL Compiler
Behavior Monitor
behavioral information
EDL-style descriptions
debugging requests
User

Following is a discussion of the important components of the monitoring system. The goal of the discussion is not to describe their workings in detail, but rather to provide a functional view and relate these components to the behavioral abstraction techniques which they implement.

Event Compiler and Librarian.

The event librarian maintains a particular view on a system for the monitor. Each view of a system is defined in terms of the set of primitive events upon which all high-level events in the view are expressed. The librarian is also charged with keeping newly defined abstractions consistent with previous ones. Monitor users can directly delete definitions from the library, while the adding and replacing of definitions is done through the EDL compiler. Users interact with the EDL compiler to create abstract views of the system. The compiler checks these abstractions for syntactic and local semantic correctness and will determine if usage in new abstractions of previously defined events is consistent with their definitions. The compiler outputs are in a form suitable to be directly used by the recognizer to perform its behavior recognition task and to aid the librarian in maintaining consistency among the events in its view.

Behavior Monitor.

The behavior monitor is the interface between the debugging system and the user. It is responsible for presenting system behavior to users and requesting that the recognizer examine the event stream for instances of particular events. The capabilities of this component can range from simply interacting with the user and passing direct requests to the recognizer to a quite sophisticated version that could digest system events and notice relationships among events or help the user by noting inconsistencies between what the user is using for diagnosis and what the system is actually doing. Our current version of the

behavior monitor lies between these two as its advisory capability is limited to accumulating statistics on user requests and requesting the status of recognition tasks from the event recognizer. An enhanced version is being developed that has a graphical component which relates the structural view of an abstraction to the state of its recognition. It is hoped that this presentation technique will be a significant aid for users in directing their attention to important behavioral aspects of a system.

Event Recognizer.

The event recognizer is the heart of the monitoring system. It accepts requests for detection of primitive and high-level events, makes requests on the event librarian for event definitions and watches the event stream for occurrences of these events. When a requested recognition is completed, the requestor is signalled and the recognized definition may be placed into the event stream for potential use as a constituent of a further high-level recognition.

The most important capabilities needed by the recognizer are to support filtering and clustering. Clustering is effected by extracting from the event stream a set of events that matches an event expression representing an abstraction. The events that will match a particular abstraction are not required to be contiguous in the stream. Other events, as well as normal system message traffic, are present in the stream and the recognizer must filter this "noise" from the stream.

Filtering based on the attributes of events is a principal aid to abstraction of behavior that is supported by the recognizer. This filtering is expressed in *cond* clause constraining expressions defined over the attributes possessed by an individual event and can eliminate many events in the stream from consideration as instances to be used in satisfaction of the event expression. When an event instance becomes a candidate for inclusion as a

constituent of a higher level event any *cond* clause relations involving attributes of the candidate are evaluated.

4.2 Problems in Recognition of Behaviors

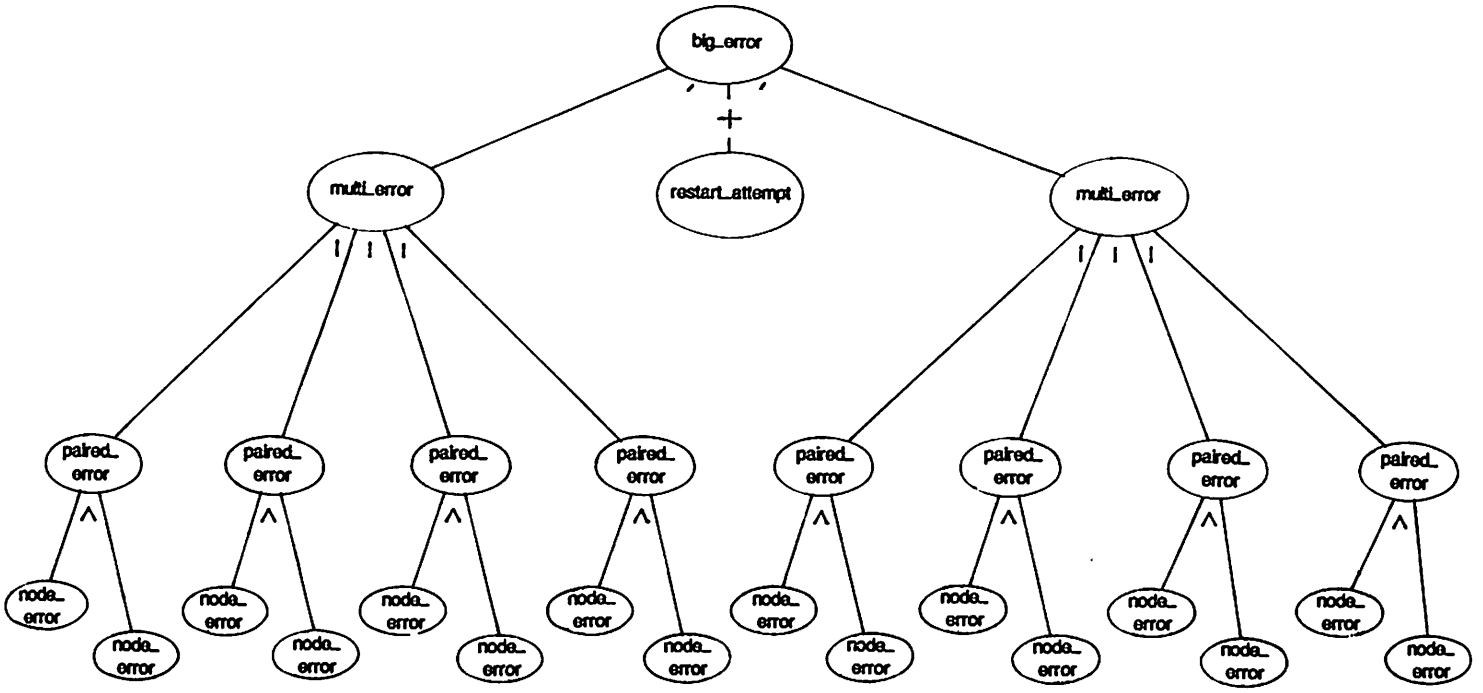
EDL provides a means for defining system abstractions in terms of events and event attributes. Abstractions defined using the Event Definition Language are used by the recognizer and behavior monitor of the monitoring system to guide recognition of behaviors and assist in correlating actual system behavior with user views of intended behavior. Many issues combine to make recognition of the occurrence of abstracted events more than a task of matching user supplied patterns against the system event stream.

Simple issues such as filtering "noise"—system message traffic as well as unneeded event type messages—are easily handled by the recognition system. Other, more difficult issues result from the characteristics of distributed systems, such as the lack of a central, reliable clock. In addition to these, another group of hard issues are those which relate the structural characteristics of EDL definitions to the dynamic properties of a system.

The following sections describe a number of the less easily resolved issues which influence the design of a recognition algorithm. The manner in which an algorithm addresses these issues will determine how well the goals of the recognizer and debugging tools are met. To place the discussion of these issues in perspective, figure 2 is a structural representation of the "big_error" event from the example of presented in section three.

Time.

Many high-level system events consist of an ordered sequence of more primitive events. A violation of the proper ordering of these events is most likely to be regarded as an error. Sequencing is easily expressed in EDL using the event catenation operator


big_error
multi_error
restart_attempt
multi_error
paired_ error
paired_ error
paired_ error
paired_ error
paired_ error
paired_ error
paired_ error
paired_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error
node_ error

between two events in an event expression. In addition, since each event has its time of occurrence as an attribute, constraining clauses can place time restrictions on events to be used as constituents of higher level events. The most intuitive method of determining if a sequence of events has occurred in the proper order is to examine their time of occurrence. This method may not always be valid in a distributed environment as individual processors will define different clocks. Although there are algorithms for synchronizing and determining skew between clocks [14], it still may not be possible to make time comparisons [15] that are accurate enough to assist in debugging synchronization or dependency problems.

At present we see no way to circumvent the inherent limitations on clock synchronization in distributed systems. Hence our prototype debugging monitor relies on the assumption that all clocks are synchronized accurately. We rely on the time attribute each event carries with it to order events even though they may be from different sources. Experimental evaluation of the impact of this assumption is planned.

Use of Abstraction Levels.

EDL event definitions permit event abstractions to be defined in terms of previously defined abstractions creating various *levels* of abstraction. The use of these levels by recognizers occurs in two complementary ways. First, when an event abstraction is recognized, it may be placed into the event stream and appear as any ordinary event does. Second, the recognizer may incorporate these high-level events into its recognition of still higher level abstractions. Using high-level events in this fashion is potentially an important technique for distribution of the debugging task and recognition of complex behavioral patterns.

One question that must be resolved when using this scheme is whether the entire substructure of a higher level event must accompany the event when it is sent to other, cooperating debugging nodes. If the structure must be sent, then the advantages of abstraction to reduce overhead induced by the debugging tools will be lost. If the structure does not accompany a higher level event, it may be difficult to determine the role that this event plays in satisfying higher level abstractions. This difficulty is due primarily to a possible inability to determine time relationships among constituents which are a number of levels removed from the higher level and from effects caused by event "sharing" in the substructure of a high-level event.

To put this time problem into perspective refer to figure 3. Each of the "multi_error" events that constitute a "big_error" instance is dependent on the occurrence of two "node_error" events. Under the event definitions given in section three, it is entirely possible that the second "multi_error" could have one or both of its substructure constituent "node_error" events occur before those needed for the first "multi_error". This possibility arises because without explicitly passing time attributes from lower levels of abstraction to higher levels, this information can be inadvertently lost during the abstraction process. This becomes a possible problem because the second "multi_error" implicitly has an instantiation time greater than the first. The "restart_attempt" events also have an implicit time ordering, placing them between the occurrences of the "multi_errors". This is easily violated by the previously mentioned scenario.

As presently defined EDL only allows a single level of event definition for a given event. An event definition may not look beyond its constituents for attributes or to direct its recognition. This design decision was influenced by our goal of limiting communication overhead in distributed versions of the monitoring system.

Sharing Of Events.

Sharing of events occurs when a single instantiated event is used as a constituent to satisfy more than one request for higher level event recognition. When a user specifies an event sequence as an abstraction the intention is probably for its constituents to be independent occurrences unless specified explicitly to the contrary. In the expanded form of the definition, it is possible for the same event class to be mentioned in many places within the same structural definition. Also, within a given view of a system, an event class may be used as a constituent of many distinct higher level event classes.

In which cases are the uses of a single instantiation to satisfy more than one request proper? The latter case, where a single instance is used to satisfy unrelated event expressions, is permissible and natural. Other cases will depend on what the creator of an abstraction had in mind. EDL has no mechanism for indicating what is actually intended. A means for letting users specify exactly what is desired is being considered for inclusion into EDL. In the meantime, the current recognizer implementation is permissive and allows the same event instance to satisfy many event expressions.

The sharing issue is closely related to the levels of abstraction issue in the following way: It may be impossible to detect that events have been shared if high-level events are communicated between nodes without also passing the details of their structure.

5.0 SUMMARY

At this time, a debugging monitor has been implemented and the problems involved in connecting it to the system that is to be debugged are being investigated. The system currently being used as an event source is a distributed version of the VMT testbed [16], which is used to test various strategies in cooperative distributed problem solving [6]. This version of the debugging monitor is centralized and only receives primitive events from the

system being observed.

Work is proceeding on a number of different aspects of the debugging monitor. One important area is the development of techniques for presenting debugging information in behavioral terms rather than as simple textual displays of state information. Distribution of the debugging task is another important aspect of our current work. There are at least two facets of distribution which are considered important. The first is determining what parts of the debugging monitor need to be distributed to maximally exploit properties of distributed systems and minimize the effects of debugging on system operation. Another concerns the sort of strategies required for requesting and communicating high-level events between nodes cooperating in debugging activities.

We believe that the behavioral abstraction approach, and the EDL based tools supporting it, will provide valuable debugging aid to developers of large computer software systems, particularly distributed systems. Having now constructed a prototype version of the debugging system, we plan to undertake an experimental evaluation of this belief. We anticipate that such experimentation will lead to new and refined tools supporting a high-level approach to debugging.

APPENDIX

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 ::= identifier | arglist , identifier

is_clause ::= *is* event_expression

event_expression ::= re_expr | *primitive* number

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 ::= empty | (expr_list)

event_index ::= empty | [number] | [identifier]

repetition ::= * | +

with_clause ::= empty | *with* attribute_list

attribute_list ::= attribute | attribute_list ; attribute

attribute ::= attribute_name = expression

attribute_name ::= identifier

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 | boolean

boolean ::= *true* | *false*

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

qualified_name ::= identifier event_index . attribute_name

cond_clause ::= empty | *cond* boolean_exprlist

boolean_exprlist ::= expression | boolean_exprlist ; expression

References

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

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

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

[4] 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.

[5] Philip H. Enslow, "What is a 'Distributed' Data Processing System," *IEEE Computer*, Vol. 11, no. 1, pp. 13-21, Jan. 1978

[6] Victor R. Lesser and Daniel D. Corkill, "Functionally Accurate, Cooperative Distributed Systems," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-11, no. 1, pp. 81-96, Jan. 1981.

[7] Peter C. Bates and Jack C. Wileden, "EDL: A Basis For Distributed System Debugging Tools," *Proceedings of the Fifteenth Hawaii International Conference on System Sciences*, (1982) pp.86-93.

[8] Peter C. Bates, Jack C. Wileden and Victor R. Lesser, "A Debugging Tool for Distributed Systems," *Proceedings of the Second Annual Phoenix Conference on Compters and Communications*, (1983) pp.311-315.

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

[10] A. C. Shaw, "Software Descriptions with Flow Expressions," *IEEE Transactions on Software Engineering*, SE-4, #3, May 1978.

[11] 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.

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

[13] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978

[14] Leslie Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, no. 7, pp. 558-565, July 1978

[15] K. Marzullo and S. Owicki, "Maintaining the Time in a Distributed System," in *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, (1983), pp. 295-305

[16] 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, Amherst, Massachusetts, 1981.