A Language to Support Debugging in Distributed Systems

Peter C. Bates*
Jack C. Wileden**
Victor R. Lesser*

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

COINS Technical Report 81-07

*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.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, the Defense Advanced Research Projects Agency, the National Aeronautics and Space Administration, or the U.S. Government.

Abstract

Debugging is a necessary prerequisite to reliable behavior in any software system. Complex distributed software systems pose some novel challenges for debugging. In this paper we describe the approaches we are taking to providing interactive debugging in a distributed environment based on a language for defining behavioral abstractions, called the Event Definition Language (EDL). Through the use of EDL, a user can create an appropriate and comprehensive view of the concurrent activity in the system to be debugged.

1.0 INTRODUCTION

Debugging is a necessary prerequisite to reliable behavior in any software system. Complex distributed software systems pose some novel challenges for debugging. These challenges, which are not found in more traditional software, arise primarily from the possibility of concurrent activity in a distributed computational system. Indeed, managing the complexity introduced by concurrent activity is the central challenge to developers of distributed software and hence to the tools intended to aid them. Concurrency gives rise to subtle interactions among various components of the software system. Providing the distributed software system developer with capabilities for coping with concurrency is a primary objective in the development of tools, including debugging tools, supporting distributed software development [Lesser and Wileden 1980].

Distributed computation possesses properties in addition to concurrency that complicate the problem of developing distributed software. Computational structures required to maximally exploit the capabilities of distributed computing are sometimes significantly different in character from existing structures used for conventional problem solving. These differences arise because of the need for fault-tolerance to improve reliability, the need for large, complex and dynamic process structures to exploit parallelism, and the need for problem solving structures that minimize inter-node synchronization and communication. These requirements lead to computational structures that have many of the characteristics of complex organizational structures

[Lesser 1979].

These new computational structures call for tools, including debugging tools, that support a global, high-level view of a distributed system, and also the ability to focus in on specific local aspects. Moreover, because the range of computational structures appropriate for distributed computing has yet to be extensively explored, development tools for distributed software systems must be highly flexible and non-prescriptive. That is, they should not enforce nor even strongly suggest a particular viewpoint on system implementation. Tools that bias developers toward thinking in terms of some particular computational organization for distributed systems might inhibit the exploration of alternatives necessary to the successful exploitation of distributed computation. In particular, we believe that tools for debugging distributed systems should be as flexible and non-prescripive as possible.

1.1 Debugging

Whenever a software system does not behave as expected a debugging situation has arisen. A software system may behave poorly for a variety of reasons:

1) inadequate problem definition or an incomplete understanding of the problem may result in a poorly designed and poorly constructed system exhibiting unacceptable performance;
2) erroneous system components (statements, algorithms, subprograms) may fail or may cause a failure in components they affect;

3) unanticipated or erroneous input data may lead a non-robust system into states from which it cannot recover.

The term 'debugging' generally connotes activities directed toward discovering and repairing any of the various types of flaws causing misbehavior in a software system.

Discovering the cause of a software system's misbehavior is typically the most difficult part of debugging. Simply observing the system's output is seldom sufficient to permit identification of flaws. Hence, the user is required to augment the standard views presented by the software in question. This is done in an attempt to obtain a more complete understanding of the system's activity, in hopes of discerning the cause of the system's misbehavior. To achieve a deeper understanding, the user often proceeds by focusing on some subset of the program states and observing when (and where) a deviation from an expected state occurs. The results may be inconclusive and a new set of observations may have to be made, possibly because the user's model of the system's computation is inaccurate and in need of revision. Alternatively, the results of a set of observations may provide the user with enough information to attempt to repair the software. The monitoring capabilities available to help the user create augmented views of the software system are crucial to this process.

Traditional debugging utilities provide a tool or set of tools enabling a user to observe the details of a software system's internal activity. Such tools provide capabilities for obtaining memory dumps, setting breakpoints, examining variables and call histories, tracing references and modifications to

selected variables, and so forth. These tools all offer a very detailed, low level perspective on the system's activity, leaving the task of developing a higher level interpretation of that activity to the tool's user, i.e. the person attempting to debug the system. Similarly, the user with a high level conception (or model) of how the system works must determine which low level program objects are relevant to a given view of the system's activity in order to use these standard tools. Occasionally tools are provided to study a particular software product from a more abstract level. For instance, a tool might accumulate statistics on a system's performance, guiding a user to alter some system parameters to effect a 'tuning' of the system. In general, however, tools of this sort are specialized to operate only with a specific software system and their output is limited to a few fixed format displays. Debugging tools providing the user with an abstract, high level view of selected details of system activity have been sorely lacking.

The advent of distributed software systems has served to accentuate our general lack of sophisticated debugging tools, since distributed systems amplify the traditional difficulties of debugging while at the same time introducing some new complications. Monitoring the internal activity in a distributed system is more difficult than in a centralized system, both because the amount of relevant information has been multiplied and because only a fraction of that information is readily available. In particular, when processing is distributed among several sites, there is no longer necessarily a single place in which all interesting activities will occur -- a fundamental

assumption in centralized systems, even those that are organized as multiple process systems. Accessing information from distant processing sites may be difficult due to bandwidth limitations on communication channels. Meanwhile, questions involving concurrency, synchronization and the sequencing and relative timing of events occurring at various sites are added to the standard list of issues that must be considered in debugging a centralized system. All of this suggests that new and more powerful tools are needed to assist in the debugging of distributed systems. Of particular importance, as we have emphasized previously, are tools to support high level, abstract views of the detailed activity within a distributed software system.

The remainder of the paper discusses our approach to distributed debugging, which is based upon the concept of behavioral abstraction. Behavioral abstraction is a means for providing the user with an appropriate and comprehensible view of the complex distributed system that is being debugged. A language for defining behavioral abstractions, called the Event Definition Language (EDL), is developed and examples of its use are discussed. The final section of the paper briefly discusses how EDL fits into an interactive distributed debugging facility and some of the major issues that need to be resolved in order to implement EDL.

2.0 BEHAVIORAL ABSTRACTION AND THE EDL

2.1 Behavioral Abstraction - What And Why

A complex distributed system can be very difficult to comprehend; it is our belief that debugging such a system 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 sources of misbehavior could be located.

Our approach to selective observation is what we call behavioral abstraction. It is based upon viewing 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. The iterative combination of filtering and clustering offers a powerful mechanism for defining viewpoints. By repeatedly filtering out the events of interest, clustering those into higher level events, then filtering out only the most interesting of those higher level events, and so on, the user can define a

suitably focused and arbitrarily high level abstraction of the system's activity. By employing an appropriate behavioral abstraction, the developer of a complex distributed software system can monitor those aspects of the system's detailed internal behavior that are relevant to the system flaw being sought without being distracted by other, irrelevant details of the system's behavior.

Naturally, the particular behavioral abstraction that will be appropriate when searching for a given flaw will vary, and no behavioral abstraction can be expected to be appropriate for all flaws. 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 flaw in a distributed system's behavior. In the remainder of this section we outline the features of EDL, then illustrate its use with an example. Issues concerning the implementation of EDL as a principal tool for interactive debugging are briefly considered in the next section.

2.2 EDL - How It Provides 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 will discuss these operations in more detail shortly. First, however, we observe that this approach depends upon the existence of an initial set of events from which additional events can be constructed. We refer to these initial 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. We believe that typically the primitive event set is small.

Given a collection of previously defined events, primitive or otherwise, 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.

2.3 EDL - Brief View Of The Language

Event definitions are composed from four primary parts, not all of which are required for every event definition (Table 1). The event heading (line 2) contains the name to be used for the class of events defined by the definition. An optional parameter list provides a means of creating generic event descriptions that might be used in different contexts with suitable argument bindings. Actual parameters are substituted textually (macro parameter substitution) to provide a fully specified event

definition when the event name appears in an event expression.

The 'is' clause (line 4) describes a regular expression that names constituent events for the event and defines the sequencing dependencies among the constituents. The event names specified in the event expression must be either primitive events or previously defined events. Sequencing relations are indicated through the use of event operators. The possible operators in event expressions are: Catenation (specified by '), meaning that the right operand event must follow the left operand event in time of occurrence; Alternation (¦), meaning either the right operand event occurs or the left operand event occurs, but both do not occur; Shuffle (#), allowing an interleaving of the operand events and thus useful in describing concurrent activity; Plus (+), a unary operator which is left associative and specifies that at least one occurrence of its operand event must occur; Star (*), the closure of plus, indicating a possibly empty sequence. In addition, parentheses may be used to group sub-sequences allowing the creation of very complex expressions.

The optional 'cond' clause (line 5) constrains the attributes of the event expression constituent events and enables filtering of events based upon the attributes they possess. For an event occurrence to be considered a constituent of the event being defined by the event expression, all of the 'cond' clause relational expressions involving attributes of that constituent event must be true. In other words, the 'cond' clause expressions form a set of assertions over the constituent events and their attributes and these assertions must hold (be true) for an instance of the enclosing definition to be signalled.

The 'with' clause (line 7) denotes the names for the visible attributes of an instance of the event being defined and indicates how to determine values for those attributes when an instance occurs. 'with' clause statements are similar to assignment statements in that they indicate how to bind a value to a name when supplied with the appropriate operands. The operands of the expressions are taken from the actual attributes exported by the event expression ('is' clause) constituents and any attributes local to the event being defined.

It should be noted that an event definition does not denote a specific event occurrence. It represents an entire class of events that meet the conditions imposed by the definition. Further, it is assumed that there exists some recognition mechanism able to detect the occurrence of the primitive events and also able to detect the clustered events defined by event defintions. We return to these assumptions briefly in the last section of this paper.

3.0 EXAMPLES OF EVENT DESCRIPTIONS

Two examples of the use of the Event Definition Language are presented. The first example (figure 1) illustrates some aspects of the language and primarily shows how clustering can be used to create a higher level event from a series of primitive events. This event is later used in a more complex event that forms an abstraction of a possible errorful situation. Briefly, the event describes a node receiving a data message, then at some later time sending out a message based in part on the received data.

The second example (figure 2) demonstrates a possible use of the EDL as a debugging tool. A system debugger might suspect that a group of nodes are not performing correctly because data communication among them takes too long. The NRPS event established that two nodes are connected because they share a data history. To describe this connection, NRPS makes use of the intranode event from the first example. A 'cond' clause constraint over the time the event requires provides the user with a means of filtering only the communication events requiring more that a certain time period.

3.1 Example 1 - Intranode

The event begins with the receipt of a message at a node (processing center) causing the receiving node to initiate one or more processing tasks based upon the received message. The executing tasks may use some local information as well as the data contained in the original message to perform an arbitrary number of processing steps. The results of one or more of these processing chains are subsequently sent to other nodes in the system.

The required event expression constituents in this example are all primitive events; receive indicates a message reception; create is the creation of a new data item by a task; send is a transmission of a message to another node. Observe that the event expression says nothing about the attributes of the events, only naming them and indicating their ordering relationships.

The filtering of the primitive events required to establish the needed chain of related data items is accomplished by the 'cond' clause expressions. The notation used to indicate the attributes of events is the dot operator used in various programming languages to indicate qualified names. In our usage the form is:

<event_name> . <attribute_name>

where event name is a primitive or previously defined event and <attribute_name> is the name of an attribute given a value by an instance of the event.

The 'data' attribute of the send, create and receive is the data item manipulated by instances of these events. The 'input_data_set' attribute of the create event is a list of data items that have been used in the processing required for the create event instance. Lines 4 thru 7 establish inductively that the data item sent out from the node is indeed related to the original received message. Finally, the relation over the 'node' attribute of the constituent send and receive confirm that all of the processing that we are interested in takes place at the same node (line 8).

When an instance of RPS occurs, it carrys along with it three attributes: 'Sent_data', naming the data it has sent to another node; 'Receiver', the identification of the node in which all of the activity has taken place; 'receive_data', the data item whose reception at the node was the initial stimulus for the event.

3.2 Example 2 - Internode Communication

The second example (figure 2) uses the event definition of the first example to define an event whose occurrence spans a group of nodes. In this example, some node A sends out a message to node B; node B does some processing that is directed by the input message and sends it along (this is the first example); at some later time, a node F receives a message that contains the original message from A in its input data set. The definition for event NRPS takes three parameters, i and j, representing the identities of the original sender of the message and the final destination node, respecively, and t, the maximum acceptable time limit between the first send and the final receive. The event expression specifies that a send event instance occurs at a specific node (whose id is bound to i) followed by a string (possibly of zero length) of the RPS events defined previously. Finally, a receive event occurs with the receiving node being the one indicated by the parameter substitution for j. The 'cond' clause relations attempt to show that there is a data link between node i and node j established by the sending of messages -- some of node j's processing activity has been influenced by an event originating at node i.

4.0 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 examples.

Naturally the language alone is of little help in debugging. To be 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, 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 the DSN Testbed project [Lesser, et. al. 1981]. In addition to the definition of EDL, a preliminary design for the monitoring and intervention facility has been completed. 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.

5.0 REFERENCES

V.R. Lesser and D.D. Corkill, Functionally Accurate Cooperative Distributive Systems, Proc. International Conference on Cybernetics and Society, Denver, Colorado, 1979, pp. 346-353

V.R. Lesser and J.C. Wileden, Issues in the Design of tools for Distributed Software Systems Development, in Software Development Tools, W.E Riddle and R.E. Fairley, ed., Springer-Verlag, Berlin, 1980, pp. 22-39

V.R. Lesser, et. al., A High Level Testbed for Cooperative Distributed Problem Solving, COINS Technical Report 81-5, University of Massachusetts, Amherst, 1981

```
(1)  event RPS is
(2)    receive'(create)*'send

(3)  cond
(4)    receive.DATA in create[first].INPUT_DATA_SET;
(5)    create[i].DATA in create[i+1].INPUT_DATA_SET;
(6)    create[i].NODE = receive.NODE;
(7)    create[last].DATA = send.DATA;
(8)    send.NODE = receive.NODE

(9)  with
(10)   SENT_DATA := send.DATA
(11)   RECEIVER := receive.NODE
(12)   RECEIVE_DATA := receive.DATA

(13) end
```

Figure 1 - Intranode example

```
(1)  event NRPS( i, j, t ) is
           { Parameters:
                   i - send node
                   j - receive node
                   t - time interval after which this is too late
           }
(2)    send'(RPS)*'receive

(3)  cond
(4)    send.node = i;
(5)    RPS[first].RECEIVE_DATA = send.DATA;
(6)    RPS[k].RECEIVE_DATA = RPS[k-1].SENT_DATA;
(7)    RPS[last].SENT_DATA = receive.DATA;
(8)    receive.node = j;
(9)    receive.TIME - send.TIME > t

(10) with
(11)   SENT_DATA := send.DATA;
(12)   TIME := receive.TIME - send.TIME

(13) end
```

Figure 2 - Internode example

(1)	<event_definition> ::=	event <event_heading> <is_clause> [<cond_clause>] [<with_clause>] end_event
(2)	<event_heading> ::=	<event_name> | <event_name><parameter_list>
(3)	<parameter_list> ::=	(<id> {,<id>})
(4)	<is_clause> ::=	is <event_expression>
(5)	<cond_clause> ::=	cond <boolean_expr> {,<boolean_expr>}
(6)	<boolean_expr> ::=	<attribute_name><relop><expression>
(7)	<with_clause> ::=	with <assignment> {;<assignment>}
(8)	<assignment> ::=	<attribute_name> := <expression>
(9)	<attribute_name> ::=	<id>

Table 1 - Language Summary

$