An Approach to High-Level Debugging of Distributed Systems

Peter Bates *
Jack C. Wileden **

COINS Technical Report 82-35
December 1982

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

---

*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.

An Approach to High-Level Debugging of Distributed Systems

Peter Bates and Jack C. Wileden

Department of Computer and Information Science
University of Massachusetts

Abstract

As part of a study of methods and strategies for problem solving in a distributed environment, we have been investigating techniques suitable for use in debugging programs written for implementation on distributed processing networks.

Traditional debugging methods emphasize techniques that apply at the level of computation units and tend to give the user only a low-level viewpoint on a system. Users of such debugging methods are forced to synthesize a more coherent, higher-level, view of the system activity without any help or support from the debugging tools. This higher-level view is generally necessary to gaining an understanding of errors and to effecting their correction. We believe that the lack of support for higher-level views of system behavior make traditional techniques inadequate for debugging distributed programs.

In this paper we consider the Behavioral Abstraction approach to high-level debugging of distributed systems. Behavioral Abstraction and the Event Definition Language that is used in presenting this approach as a debugging tool are discussed. In particular, issues related to recognizing the occurrence of abstracted behaviors, a fundamental capability required in tools providing debugging aid through the Behavioral Abstraction approach, are considered.

1.0 Introduction

As part of a study of methods and strategies for problem solving in a distributed environment [Less80], we have been investigating techniques suitable for use in debugging programs written for implementation on distributed processing networks.

Traditional debugging methods emphasize techniques that apply at the level of computation units and generally allow users to examine, and possibly alter, the state of a computation. Interactive debugging monitors are probably the most powerful implementations of the traditional method and usually permit a user to examine an entire snapshot of system state at any step of the computation. It is the job of the debugger (usually a person directing the error search) to determine what units are relevant to some problem, examine the units in whatever fashion is available, and then fit the results of these examinations into a model of how the computation works. Two elements essential to the successful completion of the debugging task are evident here: the ability to monitor, in some meaningful way, the relevant system activity so as to understand why system behavior differs from the debugger's model, and the ability to perform experiments based (implicitly or explicitly) on the information gathered. Through the interaction of these two elements a debugger attempts to gain an understanding of the causes of an error or at least to note where the implementation and the expected behavior differ.

In attempting to attain this understanding (which is the difficult part of debugging) a debugger usually abstracts parts of the program's activity to match parts of the debugger's model of how the system is supposed to function. Through experiments that perturb parts of the system in a controlled fashion, the debugger is attempting to verify that the implementation, the abstractions and the model fit together in a meaningful way. Traditional debugging tools support only a unit-at-a-time, state-based methodology and hence cannot provide the higher-level viewpoint necessary for making a meaningful comparison between actual system activity and a conceptual model of expected system behavior. Such a higher-level view is especially valuable for dealing with the complexity inherent in a distributed system.

In this paper we consider the Behavioral Abstraction (BA) approach to high-level debugging of distributed systems. We discuss behavioral abstraction and the Event Definition Language that aids this approach as a debugging tool in section 2. Section 3 addresses one of the fundamental issues arising in actually providing debugging aid through the BA approach, that of recognizing the occurrence of abstracted behaviors. We conclude the paper with an assessment of our present status and outstanding problems.

2.0 Behavioral Abstraction And The Event Definition Language

Behavioral Abstraction [Bate82] provides a foundation for a different way to view a system -- in terms of its activity rather than its state. The basis for behavioral abstraction is viewing a system's activity, or behavior, as a stream of event occurrences. By expressing higher level events in terms of this event stream a debugger is able to create abstractions of this detailed and possibly voluminous event stream.

Abstraction in programming languages is well recognized as a method of highlighting similarities and relations among collections of objects with the effect that more powerful and understandable structures result. Behavioral abstraction has a similar effect. Basic system activity is represented by a set of primitive events that describe fundamental and significant system behaviors (e.g., process creation, page fault, message transmission or reception). Using the primitive events as a starting point, designated sequences of system activities can then be used to represent higher level abstractions of system behavior. This process can be continued by using previous abstractions in creating new abstractions.

The use of behavioral abstraction for debugging is fundamentally different from other debugging techniques. A debugging monitor based on behavioral abstraction will allow a user to describe models of system behavior in terms of system activities and will then compare these abstractions with the actual behavior of the system. The choice of primitive events

determines the lowest level of system activity that it is possible to observe and may also suggest a particular viewpoint on the system. Behavioral abstraction, however, permits the debugger to define alternative, higher-level viewpoints on the system and then to observe the system's behavior from those alternative perspectives.

In our approach to distributed system debugging, two techniques, filtering and clustering, are used to give a debugger the ability to define abstractions over the event stream as an aid in attaining an understanding of the system's behavior. Clustering coalesces a designated sequence of events into a single higher-level event and provides the principal abstraction mechanism of our approach. Such higher-level events can then be considered as part of the event stream and can be used as constituents in still higher-level clustering operations. Filtering the event stream has the effect of removing selected event instances from consideration as constituents of higher-level events. This aids the debugger in focusing on those events that are relevant to a particular viewpoint on system activity. Filtering and clustering greatly increase a debugger's ability to project a model onto a system's activity. This contrasts sharply with traditional methods in which the user must first determine which among a (generally large) set of state variables are relevant to the pertinent system model, then observe their behavior, and subsequently attempt to integrate these observations into abstractions of system activity suitable for comparison to the assumed system model.

2.1 The Event Definition Language (EDL)

The Event Definition Language [Bate82] was created to aid a user in describing event-based abstractions. Clustering is accomplished in EDL by using event expressions to indicate a set of events and the sequencing relations among these events that will constitute a higher level event. Two kinds of event filtering are provided by EDL. A coarse filtering is accomplished by the clustering event expression since only event types that are considered relevant to the abstraction are mentioned in the expression. A finer filtering can be specified based on the attributes of the events that are the constituents of a higher-level event. This is accomplished in EDL by specifying relations among these attributes in a series of constraining clauses, expressed in terms of relational and arithmetic operators and various named attributes of the constituent events mentioned in the event expression. To allow for flexible use of this type of filtering, EDL definitions may also designate the set of attributes to be associated with the event being defined, specifying how values are to be bound to these attributes when an instance of the defined event occurs.

3.0 The Recognition Problem

A significant problem in constructing a debugging system using an event based behavioral abstraction viewpoint lies with the ability to detect the occurrence of behaviors that match the abstractions. The EDL provides a means for defining system abstractions in terms of events and event attributes. Given the

abstraction capabilities expressible in the EDL and the goals of the BA based debugger, recognition of event strings that satisfy the abstractions proves to be more than a simple string match of the system event stream against a supplied string.

3.1 Desired Capabilities Of The Recognizer

Demands on the capabilities of the recognizer stem from two sources, the needs of the recognition algorithm and the needs of other components of the debugging tools using the recognizer. The latter are considered to be mainly requests for recognition of abstracted events and questions concerning the status of such requests. Fulfillment of inquires are easily derivable from structures involved in recognizer implementation and are not considered terribly difficult to deal with.

Any recognition algorithm to be considered has only a few seemingly simple tasks to perform for each request for an event recognition. One of these is the ability to ignore 'noise' in the event stream. The event stream will contain events that are possibly needed as constituents of abstractions as well as events that have no bearing on the problems at hand. Additionally, the event stream will most likely be present with 'other' message traffic that is not considered event-type material and also constitutes noise.

A more directly obvious capability needed by the recognizer is that of extracting from the event stream a string 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, possibly relevant to other abstractions, possibly not needed at all are present -- this is the previously mentioned event-type noise. The recognizer must also be capable of 'looking' for a number of abstractions simultaneously. Related to both of these is the possiblity that as the stream (minus non-event type noise) enters, many sets of events that begin but do not complete an event expression can be created. A decision must be made how to handle these sets and what their presence means. They can all be retained or all but a 'most likely' set can be retained leaving open the possiblity of missing an important occurrence. Further, as a set of events enters that permits an event expression to complete, this ending may be compatible with many of the prefixes which are currently being considered. Some of these considerations are aided by the filtering capability of the recognizer.

Filtering based on the attributes of events is a principal aid to abstraction of behavior that must be supported by the recognizer. This filtering is expressed in the 'cond' clause expressions of an event definition. This filtering can eliminate many events in the stream from consideration as instances to be used in satisfaction of the event expression. Some constraining clauses will be simple and result in an immediate determination of an event's worthiness for inclusion, but other expressions will complicate evaluation by being dependent on events that have not arrived yet or allow many events to be considered for the

same place in an event expression.

Finally, it is desired that the recognition algorithm itself be capable of being distributed in a distributed system or at least that the algorithm not frustrate attempts to distribute the debugging task around the system. The motivation for investigation of behavioral abstraction based debugging has resulted from recognizing the potential advantages of writing programs distributed across a distributed computing system [Ensl78, Less81]. It is envisioned that the debugging tools developed for this type of work can also use these advantages to perform their functions. Further, other properties of distributed systems may make it necessary to distribute the debugging tools in the system.

3.2 Some Aspects Of Any Recognition Algorithm

The design of a recognizer that is to use EDL definitions as a guide to the recognition of abstractions is first of all dependent on the view of the set of strings defined by an EDL definition. One view is obvious -- a definition simply specifies a set of strings composed from event names. An alternative and more powerful view is that any high-level event can be examined in terms of its constituent events and their sequencing operations. These constituents can be spoken of in a similar fashion and will resemble a tree structure with events and operators filling in as nodes with only primitive events found at the ends of the branches. Using this view as a guideline, our interpretation of the expansion of a high-level event to its

primitives defines a set of disjoint tree structures rooted at the constituents of the high-level event string. The constituents of these may then be similarly expanded. Note that this only gives a strucural view of a definition, it remains for instantiated events to be bound to the event nodes and application of the event operators to determine an instantiation of a higher-level event. In addition to reconciling these alternative views of an event, the recognition problem attempts to define an algorithm that blends well with the desired capabilities of the debugging tools using the recognizer.

The following sections will introduce various aspects of the recognition problem that influence the design of a recognition algorithm. The manner in which an algorithm addresses these aspects will determine how well the goals of the recognizer and debugging tools are met.

3.2.1 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 source. Sequencing is easily expressed in EDL using the event catenation operator between two events in an event expression. The most intuitive method of determinining 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 [Lamp78]. While there are algorithms for synchronizing and

determining skew between clocks, their usage may not always be feasible. The main point still remains, can time be discussed in a reliable manner as might be required for synchronization or dependency problems.

3.2.2 Level Of Visible Event

The occurrence of a high-level event will have an event instantiation and its attributes bound to each primitive at the branch ends. The higher level events that they define will be instantiated as attributes are defined and alternates are resolved.

The concern of the level of visible event aspect is whether or not it is possible for the recognizer to 'see' and/or use high level events in recognition. Higher level events that are recognized and instantiated locally (where the recognition algorithm resides) can be considered part of the stream and are an aid to abstraction. It might be possible that the recognizer is unable to use high-level events that have been inserted into the stream. This situation could arise primarily from the desire to control 'sharing' of constituent events among distinct parts of the expanded event definition. Difficulties related to seeing high-level events occur when the recognizer is distributed and the desire to send high-level events to cooperating recognizers occurs. The reasons to want to send high-level events around are varied: it is an abstraction mechanism that is supported by the EDL and is considered quite useful; network constraints such as node topology or security issues may require transmission of only

high-level events; or it may be important to reduce the amount of event traffic and exchanging high-level events would accomplish this. If high-level events can be passed around, a consideration that is contrary to some of these reasons but consistent with other aspects of the recognition problem is whether the structure that has been instantiated for the high-level event is available to be examined. Again, sharing of event instantiations is the reason for the possible need to cart the instantiated structure around with each instantiated event.

3.2.3 Level Of Recognition

Level of recognition is concerned with what level of events of the tree structure defined by an event definition is used to match the event stream. One question here is whether or not only primitive events are to be used in the recognition of a higher level event. The load sharing and abstraction abilities possible in a distributed system might be impacted if only primitives were to be used. Operation of the recognizer itself would become more complicated. Since any higher-level event can have its definition expanded to the primitives at its leaves, a simple experssion at the top level could become quite complicated by the time it has been expressed in terms of its primitives. On the positive side, sharing of events among various instantiations is less of a problem.

The 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 the event watching. The recognizer is not constrained to only use a single level but it may be the only desirable method given that the EDL user has no control over instantiation of the underlying structure of an event.

Given a fully expanded definition of a high-level event, an alternate way to consider the event is to draw a digraph through the derivation tree that represents all possible paths through the various levels of constituents that could be a valid string. This is a powerful view of a high-level event and will work for any mix of high and low level events as constituents.

3.2.4 Sharing Of Events

When a user specifies an event string 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.

The sharing aspect involves a decision on whether or not a single instantiated event may be used to satisfy two or more mentions of the event in an expanded definition of a high-level event. When sharing is allowed, the notion of disjoint event occurrences becomes blurred. The key point to be made regarding event sharing is that the set of strings of event occurrences

acceptable for an instantiation of an event will be different depending on how sharing is treated. However, this difference is only related to the level at which event strings are viewed.

3.2.5 Limits On Constraints And Event Operators

The constraining expressions defined in the 'cond' clause of an event definition can be quite expressive and unrestricted in the kinds of relations possible among events. In the absence of constraints, recognition becomes a pattern match of the event expression against the event stream. These cond clause expressions provide an important filtering capability that seeks to narrow the focus of behavioral searches. Two basic kinds of cond clause expression are possible. The first generally will help to make the search space for a particular event class smaller by allowing an immediate decision on whether an event should or should not be considered as a constituent of some high-level event. Included in these are expressions that only involve equality (and inequality) between attributes and simple scalars. If an event possesses (or does not possess) a simple quality it is a candidate for inclusion or exclusion.

Relations among attributes of different events will tend to delay the inclusion decision and increase the number of events and possible prefix strings that must be maintained while looking for an event. Closely related to this is the question of which set of events should be used to instantiate an instance of a high-level event if there are multiple eligible sets.

One final part of the constraint aspect is whether or not certain relations or attributes make sense at all. For example, consider that time as an event attribute cannot be considered reliable, then time and any relation defined in terms of time cannot be discussed meaningfully as well as rendering the event catenation operator somewhat less useful. Clearly, some determination must be made on how much of the {event operator X constraint expression space} is defined, meaningful or even useful.

3.2.6 Real Time/Space Problems

As for any algorithm that is to be useful, what aspects of the recognition algorithm will cause problems for the amount of memory it needs to run in and the amount of time it needs to perform its task? A simple pattern matcher with no constraints on attributes, that operates on a single level of recognition with all events, both high and low level, being visible and useful will not demand too much time or space. On the other hand, unconstraining these aspects will possibly make the recognizer big and slow. We expect the recognizer to run in parallel with the system it is monitoring in order to observe actual system behavior and to aid in any experimentation capabilities provided.

3.3 A Recognizer Currently Being Considered

The type of recognition algorithm currently being used is based on a single level of recognition. An event is recognized when a string consisting of the events mentioned in its event expression, subject to the ordering imposed by its operators, and of course, meeting the filtering criteria imposed by its constraining expressions, is noticed in the event stream. The kinds of constraining expressions allowed are not restricted. It is felt that a certain amount of experience with different recognition algorithms and recognitions will be necessary before reasonable restrictions can be made on the use of constraining expressions -- if restrictions are necessary at all.

When a request for recognition of a high-level event is made, the definition of that event is located and then the definitions for all of its high-level constituents are located. Each of these high-level events also becomes a request for recognition. This scheme of instantiating the entire structure of a request provides a single level of recognition as well as the necessary link to the primitive events supporting an event instantiation.

Currently there are no controls placed over the individual recognizers with the effect that there is a 'bag' of recognizers that continue to recognize their high-level events and insert them into the event stream indefinitly. This recognizer says nothing about sharing of constituents since each recognizer operates independently and each may use any event it sees and has

a need for. Using this kind of a recognition algorithm as a base, a more powerful recognizer can be constructed by placing controls on the collection of single level recognizers such that they are only operating when there is a possiblity that the piece each notices is needed. Sharing can more easily be controlled and the amount of unproductive recognition work is lessened. Using these algorithms, high-level events from many sources are useable but the algorithm will also function where high-level events external to a recognizer are not visible and primitives must be used for all recognition.

For our current pass at recognition, the questions concerning time are assumed to be resolvable. Each recognizer contains an event history that may be reordered when events with times between previously received events are received. Being left open for the present is whether the history may be reordered when other information is received. Other information would include such things as more accurate time stamps for events or certainty over possible errors in reported events.

4.0 The Future

To date we have defined the Event Definition Language, studied its suitability for use in distributed debugging, and undertaken a prototype implementation of a distributed debugging aid based on the behavioral abstraction approach and EDL. Our experience, thusfar limited to manual application on small examples, indicates that these methods should lead to extremely valuable tools to assist with the debugging of distributed software

systems.

Our current work is directed toward improving both our understanding of and our automated tools supporting distributed debugging. The design of our prototype debugging system supporting EDL and the behavioral abstraction approach is completed, and many components of that system are already working. This version of the system employs simple, crude techniques for recognizing occurrences of EDL-defined events, however, and we are investigating more powerful and more efficient approaches to this important aspect of the debugging tool. The prototype version also relies on a centralized debugging system located in a single node of the distributed system that is being debugged. We are currently studying distributed organizations for the debugging system. Other enhancements to the debugging system, incorporating knowledge-based AI techniques and functionally accurate, cooperative distributed problem-solving methods, are also being considered. Experimentation with the prototype and with its successors should greatly expand our understanding of distributed systems and how to debug them.

References

[Bate82] 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.

[Ensl78] Philip H. Enslow, "What is a 'Distributed' Data Processing System", IEEE Computer, Vol. 11, no. 1, pp. 13-21, Jan. 1978

[Lamp78] 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

[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 Distributed Problem Solving," Technical Report TR-81-16, Department of Computer and Information Sciences, University of Massachusetts, (1981).

[Less81] 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.