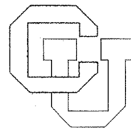
Idd: An Interactive Distributed Debugger

Paul K. Harter
Dennis Heimbigner
Roger King

CU-CS-274-84



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Idd: An Interactive Distributed Debugger

Paul Harter[1]
Dennis Heimbigner[2]
Roger King[3]

Computer Science Department
University of Colorado
Boulder, CO 80309

ABSTRACT

The Interactive Distributed Debugger (Idd) is intended to be a comprehensive system for debugging distributed programs. In addition to the standard sequential debugging techniques it provides a powerful assertion language, based on a temporal logic, for automatically monitoring the flow of a program. These assertions are checked at run-time and the program is stopped when an assertion is found to be invalid. Unlike other systems that assume a given set of assertions, Idd allows the user to expand the assertion set interactively as he gains understanding of the proper behavior of the program. In addition, Idd provides a sophisticated graphics interface to display and filter the information provided by the debugging system.

1. Introduction

The problem of effectively creating correct and efficient real-time and concurrent software is rapidly emerging as one of the key software engineering problems of the 1980's. With the sharp drop in computing hardware costs, increasing emphasis is being placed upon using dedicated computers to handle smaller pieces of large computing problems. Solutions to the larger problems are being effected by the interaction of such dedicated computing systems distributed around networks. Such architectures offer numerous advantages over older architectures under which entire large problems

[1]This author was partially supported by NSF grant MCS-8216707.

[2]This author was supported by a University of Colorado Summer Research Initiation Foundation Fellowship (1984).

[3]This author was supported by IBM through a faculty development award (1984).

were solved using a single large hardware/software system. The distributed approach offers such advantages as survivability, reliability, and the possibility of functional modularity at both the hardware and software level.

A major problem with this approach, however, is that the software required to support it is far harder to construct than sequential, single-process software. The difficulties lie in the construction activity, and to an even greater extent in the expensive and crucial activities of debugging, testing and verifying. These activities have been effectively assisted by tools and software engineering technology for single-process, sequential software. Unfortunately, software tools and technology have not been effectively applied to the support these activities for real-time and distributed software.

This work concerns the application of modern debugging tools and technology to real-time and distributed software development. The paper first discusses the strategies for distributed debugging. It then presents the overall architecture of Idd and introduces the graphic interface. Third, it describes interval logic and its role as an assertion language using a example of the two-phase commit protocol. Finally, use of the graphics interface is illustrated, again using the two-phase commit example.

2. A Strategy for Debugging Distributed Systems

In order to debug a distributed program effectively, one must use many different strategies. Distributed programs are sufficiently complex that they require all of the techniques used for sequential programs as well as new techniques specifically designed for handling their distributed nature. A complete set of techniques must include: state examination, stepping, tracing, message monitoring, and assertion testing.

State examination is the traditional tool for debugging sequential programs. The programmer places breakpoints at various places in a program and runs the program. At each breakpoint, the state is examined to see if it conforms to the programmers notion of a correct state.

Stepping is closely associated with state examination. In stepping, program execution pauses before the each statement in the dynamic execution sequence to allow programmer inspection of the current state of the computation. This is very important in determining when a program is executing correctly. Normally, the programmer knows approximately what path should be executed when given a certain initial program state, and he can detect deviations from that path.

Tracing usually refers to the automatic printing of state information during the dynamic execution of a program. Here too, the programmer knows the expected sequence of outputs and can spot anomalous behavior.

Message monitoring is not needed for sequential debugging, but it is essential for debugging distributed programs. It is true that one could substitute tracing of process states for message monitoring since the message resides in a process just before it is sent, or just after it is received. Yet, one often does not want to have to deal with the internal state of a program in order to see its externally visible behavior. Also, one may wish to inject synthetic messages into an execution sequence to observe its reactions, and this is difficult using only states.

Finally, there is assertion monitoring. Many debuggers provide no assertion monitoring at all, and rely totally on stepping and tracing. Some sequential programs provide only primitive forms of assertions, namely conditional breakpoints with simple conditions based on the state of program variables. This debugging style assumes that the program is deterministic, and so there is a uniquely specifiable state that is of interest. In the distributed program, it is often impossible to specify exactly the state of the desired breakpoint or of the possible failures. Rather, one must be able to specify the relative sequence of classes of events and to specify invariant conditions that should hold over intervals of program execution.

The traditional complaint against assertion-based debugging is the difficulty of defining sufficient assertions to characterize all of the possible errors. This difficulty arises both from the complexity of programs and from the complexities of available formalisms. But assertion monitoring is too powerful to ignore. In a distributed environment, it can relieve the programmer of the burden of examining large traces. In addition, it can help catch some time dependent errors by allowing automatic monitoring of states and message sequences, as opposed to the manual insertion of breakpoints.

In Idd, we circumvent the difficulties of defining assertions by two mechanisms: a powerful graphic interface, and incremental assertion generation. The latter technique is based on the availability of stepping, tracing, and so on in conjunction with assertion monitoring. Thus, a typical debugging session might begin by approaching the problem using one of the former techniques. Then, having observed a behavior sequence ending in error, the programmer could write an assertion describing some prefix of the sequence so that he gets control if that sequence repeats. At that time he can search for precursors of the error. Alternatively, he may observe some correct behavior related to a problem and choose to check that with an assertion so that he is notified as soon as a change undoes something that was working correctly. Thus, although complete characterization may be impossible in general, he may be able to set up assertions to guard against some of the failures that have been seen (and fixed) previously.

3. The Idd Graphics Interface

The number of messages crisscrossing a distributed network may be very large. Clearly, the interaction of messages in a distributed system would be difficult to view and to manipulate with a linear syntax. For this reason, the Idd interface is graphics-based. It provides a formated screen for viewing message traffic, and a number of graphics operators which may be used to focus on messages of interest. The goal is to allow a user to debug distributed programs by monitoring the progress of a system, and then when an error is found, exploring the types, sources, destinations, and contents of relevant messages. Aggregate information concerning message traffic may also be collected. Immediately below, we discuss the screen layout and the operations with which the user focuses his attention. A later section gives a brief scenario of the graphic interface in action. First, we describe the basic operations.

The user's view of the system traffic consists of a time line, with one axis representing processes and the other time. Two points connected by a directed line represent the passage of a message from one process to another. An initial screen is shown in Figure 1. The user may view the message traffic as the system proceeds, or view the history file of messages.

At the ends of the two axes are a series of scroll icons. When viewing history, the user may pick one of the arrows with a mouse, in order to go forward or backward in either time or process number. At the bottom of the screen is a "throttle", which may be used to speed up or slow down the viewing of the message history.

At the top of the screen are five global commands which may be picked with the mouse: If **Magnify** is selected, user then uses the mouse to pick four points on the screen. The area in the marked rectangle is then expanded. **Monitor** is picked when the user wishes to view message traffic as is occurs, and be informed when a temporal assertion is not met or some breakpoint is encountered. When the system halts, the relevant assertion or breakpoint is displayed.

If **Local** is selected, the user then selects a particular process with a pick from the Process axis. The user then enters a separate screen, devoted to only one process, called a *process screen*. In the process screen, the user may set breakpoints, view those temporal assertions that concern only that process, insert or modify assertions, examine source code, or use examine the current process state. To leave a process screen, the user picks the **Global** operation.

The last two operations are **Display** and **Filter**, which are used to focus the display on messages of interest. These are discussed in detail later, when the sample scenario is given.

4. The Architecture of Idd

It is clear that, aside from the capabilities of a normal sequential debugger, a programmer of distributed systems must have control of inter-process communication. To provide this, the monitoring system must monitor the activities of all processes comprising the system. In order to minimize interference, monitor functions will be separated into two subsystems: the primitive monitoring functions and the analysis and display functions. We envision an architecture which consists of a set of satellite monitors to provide simple, local control of process behavior, combined with a separate supervisor monitor for global control. Each satellite communicates with the controller using small messages to reduce the effect on the program being monitored.

4.1. Supervisor Responsibilities

The supervisor program is responsible for handling control messages to and from the satellites, monitoring and collecting messages generated by the distributed program, providing the graphic interface to that data, and monitoring assertions.

Assertion monitoring may operate either on the incoming messages (approximating real time) or on some portion of the collected message history. In the first case, assertion violation causes the supervisor to stop the monitored program and allow the programmer to inspect the various processes of the program. In the latter case, the supervisor indicates the place in the message history where the assertion is violated and allows the programmer to examine that portion of the message history.

Normally, the supervisor monitors those assertions that are global to the whole program and that cannot be monitored by any one of the satellites. Not all assertions are global in nature. Assertions dealing with more than one process (eg. concerning messages) are global and are monitored by the supervisor. Other assertions involve the behavior of only a single process and may be monitored locally by a satellite. Local monitoring also minimizes the time between exception and interruption, as no message traffic is required.

4.2. Satellite Responsibilities

The satellites are responsible for the direct control of monitored processes. Control actions are of three kinds. First, a process may be stopped, continued, or restarted. Second, the state of a process may be examined or altered. Third, a satellite may be responsible for monitoring some assertions that depend only upon a single process.

The actions may be invoked in one of two ways:

(1) The programmer, through the supervisor, requests that some action be performed on some process. For example, he may request that some assertion be monitored. In this case, the supervisor may note that it is not a global assertion, and send it to a satellite for direct monitoring.

(2) The satellite's process may cause an error (including assertion violation), and this will automatically invoke the satellite which in turn will notify the supervisor.

4.3. Problems of Real-Time Control

It is important that the debugger be able to stop a set of processes as fast as possible in order to preserve the relative states of the processes for programmer inspection. This is especially important in detecting time-dependent errors. Unfortunately, no matter how quickly one can stop a collection of processes, there will be some race conditions of such short duration that they cannot be caught in the middle.

Often, the programmer has a good understanding of where in a program time dependencies are critical. Idd caters to this knowledge by providing a stepping mode for a collection of processes. Thus, a programmer can single step a collection of processes to watch their behavior in a critical piece of code. By varying the order of process execution, he can often test out the behaviour under various assumptions about message arrival and execution speed.

4.4. Systems Requirements for Distributed Debugging

An effective debugging environment for distributed programs must ultimately be embedded deeply into the operating system. Only in this way can one achieve sufficient speed and transparency. The speed is necessary to preserve the relative process states and to catch time-dependent errors that result from closely timed race conditions.

Transparency is important in separating the debugging system from the program being debugged. Ideally, the debugger is language independent and does not depend upon code residing in the user program for correct operation. Again, such an ideal state may not be entirely possible. The principal problem concerns the level of debugging. If a programming language provides relatively high level message passing primitives (Ada rendezvous, for example), then one would like the debugger to know about them. In this way, these primitives can provide a level of abstraction for the programmer, and the debugger can support operation at this level of abstraction. If the debugger only knows about the message facilities provided by the operating system, then it will force all debugging to occur at that level, and that may be irritating and confusing to the programmer.

In the initial stages of the IDD project, we will not be in a position to modify the operating system, thus we must take the tack of providing a synthetic environment within which users will operate. Our initial environment will be a network Sun workstations running Berkeley Unix 4.2. We will be using the stream socket protocol for messages, and the ptrace facility for controlling processes. We would expect to work with one of the standard programming languages (C, Modula2, and possibly Ada).

Such an environment leaves much to be desired for distributed debugging. The standard ptrace facility is designed to debug one-process sequential programs. It allows the debugger to start the user program as a child process, control its execution, and examine its state. If that child process forks it own children, the debugger cannot reference them using ptrace. Since this is likely to be a common occurrence in a distributed program, some means must be developed for attaching debugging monitors to these descendant processes.

The Unix stream socket is a specialized form of interprocess communication (IPC). Presently, it is the only completely usable IPC mechanism provided. It is, however, completely unsuited for message monitoring. Since it is stream oriented, the receiver does not get a series of disjoint messages. Rather, all the messages sent are concatenated at the receiver, and it is difficult for the debugger to automatically recognize the message boundaries.

Our (temporary) solution is to provide a specialized debugging library that would be loaded with the user's program to replace some standard Unix facilities. This library would provide replacements for the process forking and the normal stream socket primitives. The new forking routine would operate as follows:

(1) The child process is forked and a debug monitoring program starts execution.

(2) This monitor then forks the actual user process.

Thus a monitor process is interposed between the parent and the child. As an aid to transparency, the fork routine returns the process id of the sub-sub child and not that of the monitor. This will work in many situations, but any program that requires its children to be immediate will not work correctly.

The replacements for the stream socket will actually do two things. First, it will provide a form of stream socket in which the chunks of data passing through a stream can be monitored. As indicated, this is not quite what is wanted, but is useful for programs that use streams as pipe-like connections. The second feature will be a new interface that provides messages on top of the stream socket. By controlling both ends and adding appropriate control information, the library can simulate messages. Thus, programs that are written to use this interface will have the benefit of extra debugging help.

5. Monitoring Interval Logic Assertions

As mentioned earlier, a major difficulty of debugging distributed programs is presented by the very fact that they are distributed. Previous assertional methods depend on the notion of a *global state*, which is the entity tested by the assertions being written. In distributed systems with no shared memory, the state of the program is distributed throughout the system such that obtaining a "snapshot" for testing purposes is impossible. In the class of systems we are considering, i.e. those with a shared broadcast medium such as Ethernet, there is one object whose state is testable and may be considered global. This is the broadcast medium. Further, in many distributed programs, such as those implementing communication or coordination protocols, the individual process states have very little meaning and the state of the computation is best described by the sequence of messages that have been transmitted. Thus, we will focus primarily on monitoring assertions describing message traffic.

Other types of assertions can be monitored as well, assertions describing the states of individual processes can be monitored either by delegation to the local satellite or by having the process export its state explicitly via extra messages. This requires extra code in processes, but is, unfortunately, the only way to monitor the local states of two processes concurrently with some global assertion. Again, the latter will result in an approximation at best, unless the processes are also constrained to wait following state exportation.

The next two subsections describe the interval logic underlying our assertion monitoring and used in examples in this paper, and give some indication of the techniques that will be used to monitor assertions.

5.1. Interval Logic

Interval logic [Schwartz 83] is an extension of linear time temporal logic [Lamport 80] developed to eliminate a major difficulty involved with the use of ordinary temporal logic. Ordinary temporal logic has been studied with much interest in recent years as a language for stating and reasoning about properties of concurrent or distributed programs. The motivation for this is that it is capable of expressing behaviors over time, unlike the various partial-correctness logics, which are restricted to properties expressible in terms of input/output state pairs. This restriction becomes particularly relevant when considering concurrent or distributed programs because of the fact that correctness properties of many such programs do not depend on mapping some initial state to some final state, but rather on exhibiting some stimulus/response behavior. Thus, the ability of temporal logics to express properties of behaviors makes them interesting for specifying and reasoning about properties of concurrent or distributed programs.

Both ordinary temporal logic and interval logic have models based on state sequences and include operators and predicates for describing properties of individual states as well as sequences of states. The predicates on states are essentially the familiar ones from predicate calculus, and the operators yielding assertions describing sequences are $\Box$, and $\Diamond$, read "always" and "eventually" respectively. In ordinary temporal logic, these operators apply to the entire sequence, which for a non-terminating program would be infinite. Thus their temporal scope is unbounded. Consider the following examples. The formula $\Box P$ asserts that predicate P is true of every state in the sequence. The formula $\Diamond P$ is true of a sequence wherein P holds for some state. Finally, the formula $\Box\Diamond P$ asserts that P is true in infinitely many states of the sequence.

Very often, however, one wishes to state that a predicate remains true continuously over some fixed period but not forever (a bounded "always") or that an event occurs before some point in the execution and not just before the end of time (a restricted "eventually"). Unfortunately this is not possible in ordinary temporal logic with $\Box$ and $\Diamond$. While it is possible in a linear time logic which gives an interpretation to $\Box$ considered as a diadic rather than a monadic operator [Gabbay 80], the expression of all but the simplest such properties leads to cumbersome and inscrutable formulations. Interval logic alleviates this difficulty by providing a convenient and readable mechanism for defining bounded segments of execution sequences over which assertions are to hold.

The notation used to state that an assertion holds for a specific period is $[I]R$ which asserts that if the sequence contains an interval described by I, then over that interval, R is true. Here, R can be any assertion. The *interval term* I is generally constructed by use of the operators $\Rightarrow$ and $\Leftarrow$ to denote the interval between two endpoints described by their operands. These operands are assertions called *event terms*, and define particular points in time (events) corresponding to the points at which the assertions *go from false to true*. The operators $\Rightarrow$ and $\Leftarrow$ determine the order and direction of search for the endpoints of a particular interval. The endpoint at the "tail" of the arrow is located first, searching forward from the beginning of the entire sequence being considered. Starting from this endpoint, the sequence is searched for the other endpoint in the direction implied by the arrow. Thus, the assertion $[P \Rightarrow Q]\Box R$ may be read "from the next time P becomes true and until the first time that Q subsequently becomes true, R will remain (constantly) true." On the other hand, the assertion $[P \Leftarrow Q]\Box R$ may be read "over the interval extending backwards from the next time Q becomes true to the closest previous time that P becomes true, R remains (constantly) true. In the first case the search is forward from P to Q, and in the second backward from Q to P. The intervals selected may be very different depending on the direction of search.

Since, if the interval specified does not exist, anything can be said to hold for it, the interpretation of interval logic specifies that formulae containing such specifications be considered vacuously true. From a logical standpoint this makes perfect sense, but practically speaking it may be necessary to be able to say that an interval will occur. To allow the expression of such a requirement, interval logic provides notation for stating that intervals occur either by requiring that an entire interval exist, or by requiring individually that endpoints exist. If I is any interval term, then $*I$ simply states that the interval I must occur. Thus the expression $*I \wedge [I]\Diamond R$ says that R must occur during the next I interval and further that there must be a next I interval. On the other hand, if I is the interval term $P \Rightarrow Q$, then $[*P \Rightarrow *Q]\Diamond R$ makes the same statement by requiring separately, within the interval specification, that the endpoints *must be* found. This latter form is particularly useful when the interval need not exist unless the first endpoint exists. Thus, $[M \Rightarrow *N]\Box \neg O$ says that if M occurs then N must occur and further that between them O is continuously false. It is hoped that the above (somewhat loose) introduction gives enough feel for the logic to make the example assertions readable.

5.2. On-line Testing of Interval Logic Assertions

In this subsection we discuss monitoring assertions of interval logic in the context of Idd. As mentioned earlier, most assertion monitoring, particularly assertions describing message traffic, will be done by the supervisor, who reads all messages transmitted over the network. Although Idd will have a mechanism for saving and viewing message histories, it might well prove infeasible to store all messages transmitted during the long execution of a large distributed program involving frequent interprocess communication. Further, we require that the programmer be notified during execution of the program as soon as an assertion is violated. Thus, although the semantics of interval logic are given in terms of completed execution sequences rather than prefixes [Schwartz 83], in Idd we need to test for compliance with assertions "on-line" and in real time.

Because of the great expressivity of the language, on-line testing of interval logic presents a difficult problem that is not yet completely understood. The general strategy is straightforward; given an assertion of the form $[I]P$ and an input message sequence, when the message event corresponding to the left endpoint of the interval I occurs, begin the necessary testing for P. When the event corresponding to the right endpoint of I occurs, the entire interval has been seen and the state of P on the interval should be known. The difficulty arises from several places. For example, the semantics of $[A \Leftarrow B]P$ are that P must be true of the interval beginning with the last A event before the next B event and extending through that next B event. Locating the left endpoint of that interval to determine at which point the testing of P is to

commence is non-trivial. Assume that recognizing an A event is easy. Then, when it occurs we have to start checking for P, which depends in general on the entire interval. If A occurs again before B occurs, then checking for P must begin again since the endpoint is the last A before the next B.

Now consider the assumption made above that A was easy to recognize. The logic allows an arbitrary interval assertion as an endpoint in an interval term. The assertion $[\neg(\Diamond X \wedge \Diamond Y) \Rightarrow Z]P$ involves an interval that begins at a point where X and Y do not occur in the future. Given the input sequence of messages, the interval could begin at the start of the sequence if not both X and Y will occur later; thus P must be checked starting there. On the other hand, if during processing an X event occurs, then it could be that from that point only Y occurs later. In this case the interval starts after the X event. If Y will not occur later, then the start of the sequence is still the left endpoint. Thus, after the X event, there must be two ongoing tests of P over two overlapping intervals. It can only be known later which is the correct left endpoint. In both of these cases, it is the prediction of the future that causes the difficulty in on-line evaluation.

Our first approach to the problem was to parse the assertion into an expression tree, where the state sequence would affect only leaf nodes (atomic predicates on states) directly and *true* and *false* would bubble up the tree in a more or less straightforward way. Nodes corresponding to the temporal operators $\Box$ and $\Diamond$ must store some information to implement the semantics of the operators correctly. For example, if the assertion modified by $\Box$ ever goes false, the $\Box$ node goes false and stays false. To handle interval formulae such as $[I]P$, we can think of the interval modifier ($[I]$) as a filter that allows the modified assertion to react only to those states that make up the described interval. The root of P's tree would then be the result for the entire assertion. Unfortunately, despite the cleanness and simplicity of this approach, we were forced to reject it because the one-way communication of this model does not handle the semantics of the interval modifier properly. Further, the problems mentioned above with regard to prediction remain difficult to handle, as information is constrained to flow in only one direction.

A related and more general approach should be more satisfactory. We parse assertions into a tree as before, this time treating $[*]*$ as an operator with two subtrees, one for the interval term and another for the modified assertion. Since assertions consisting only of simple predicates and standard boolean connectives have no "temporal content", we then prune off subtrees that correspond to to such properties. These may be viewed as simple predicates on states for the purpose of the interval logic. We are left with a tree of "state predicates" joined via the temporal connectives ($\Box$, $\Diamond$, $[\,]$). Corresponding to each node or operator, we create one pseudo process that implements the semantics of the operator labeling that node. This is described below.

These processes cooperate in the analysis of the input sequence, communicating in two directions. For example, given the assertion $[A \Rightarrow B]P$, several types of communication can occur. A asks the $\Rightarrow$ node when to begin examining states. This is because $[\,]$ may appear within a larger assertion that limits the scope of the search for the endpoints A and B. Once A has found a possible beginning for the interval, it notifies $\Rightarrow$, who in turn notifies $[\,]$ and B, who then begins looking for the right endpoint. $[\,]$ notifies P, who begins testing states. Once B has found the right endpoint, it notifies $\Rightarrow$, who then tells $[\,]$ that the interval has been found, which causes $[\,]$ to request a result from P. This scenario reflects the case where A and B are easy to recognize. If this is not the case, then "finding" must be tentative and only a final "commit" will terminate the searching and testing.

Space restrictions have forced us to paint the above with very broad strokes indeed. We hope that the example has given a hint at the method. These techniques will be described in detail in a forthcoming report [Harter 85].

6. An Example

The example to be used is taken from the two-phase commit protocol [Gray 78] The two-phase commit is an example of a protocol for commit of distributed database transactions. A principal feature of a transaction is that it is atomic. This means that either a transaction completes successfully (commits) and its modifications are available to other transactions, or it aborts and it has no effect on the database at all. The two phase commit protocol is intended to guarantee either that all pieces of a distributed transaction commit, or that all abort.

Briefly, the two phase commit operates as follows (see also figure 2):

(1) The coordinator requests all participants to vote either for commit or for abort.

(2) The coordinator then chooses to commit if all participants vote to commit, otherwise the decision is to abort.

(3) The coordinator sends the decision to all participants.

(4) The coordinator waits for acknowledgement from each participant, and re-sends the decision if no response is heard after some time period.

(5) As each participant gets the decision, it carries out the appropriate action (commit or abort).

6.1. Assertions for the Two-Phase Commit

The following describes several representative constraints on message behavior for the two-phase commit protocol. Each constraint is specified first in english and then as an interval logic assertion such as could be monitored by the supervisor. In the interval logic assertions, we use a tuple notation to describe a message. Each message is

assumed to consist of a type field, a source process id, a destination process id, and a value. The distinguished identifier "M" is used to signify the last message transmitted. Thus, the phrase "M = <P,-,-,- >" specifies the occurrence of a message whose type field indicates that it is a poll request: "-" indicates that the actual field value is irrelevant. Note: This syntax is used to illustrate the examples, and is not representative of the actual syntax to be used in Idd.

(1) The coordinator, (and the participants also) should execute in the sequence Vote and then Commit. Any other sequence is an error.

This constraint might be expressed as follows:

$$*(M=<P,-,-,-> \Rightarrow M=<D,-,-,->)$$
$$\wedge [M=<P,-,-,-> \Rightarrow begin(M=<D,-,-,->)] \Box M=<V,-,-,->$$
$$\wedge [M=<D,-,-,-> \Rightarrow] \Box M=<OK,-,-,->$$

Recall that M is set to the value of each message monitored by the debugger. Here, type D is a decision and OK is the response to that decision.

The use of intervals is crucial to expressing the sequencing of program sections. In the above constraint, the first line indicates that it must occur that a Poll message is sent out followed by the sending of a Decision message. The next two lines restrict the kinds of messages that can occur during two stages of the protocol specified by two intervals. From the time that the Poll message until just before the Decision message, all messages must be Votes, and from the time the Decision message is sent onward only OK acknowledgements may be sent.

(2) Each participant should vote once and only once when polled by the coordinator.

This might be expressed:

$$1 \le i \le MPID \supset [M=<P,-,-,-> \Rightarrow M=<D,-,-,->] \Diamond M=<V,i,-,->$$
$$\wedge [M=<V,i,-,-> \Rightarrow] \neg \Diamond M=<V,i,-,->$$

We assume here that the participants are numbered 1 through MPID. The phrase <V,i,-,- > refers to a message of type vote sent by process i. Thus, the first line specifies that in the voting interval (from first ballot to the first decision) every process sends at least one Vote. The second line specifies that from the time a process Votes onward, that process will not Vote again. This constraint is typical of send-acknowledge kinds of communications in which multiple acknowledgements would cause havoc if not detected during debugging.

(3) If any participant votes to abort, then the coordinator will decide to abort. Otherwise, the decision will be to commit.

This constraint is useful as a check on the decisions made by the coordinator. It may be expressed as follows (here, the tag COORD is the coordinator process):

$$1 \le i,j,k \le MPID \supset [M=<V,i,k,abort> \Rightarrow] \Diamond M=<D,COORD,j,abort>$$
$$\wedge \Box M \ne <V,i,k,abort> \supset \Diamond M=<D,COORD,j,commit>$$

Note that in this case, the we have included in the message structure the destination process (j, k) and the contents ("abort" or "commit"). Here we specify that the first

"abort" message begins a period during which the coordinator must decide to abort, and second that if there's never a Vote to abort, then eventually the coordinator must commit.

(4) The participants should not respond to the decision until it is actually sent.

This last constraint is typical of send-acknowledge situations where we want to check that indeed the acknowledgement occurs and that it does not occur before the send by the coordinator.

$$[\Leftarrow begin(M=<D,COORD,j,commit>)]\Box \neg M=<OK,j,COORD,->$$
$$\wedge [(M=<D,COORD,j,commit> \Rightarrow]\Diamond M=<OK,j,COORD,->$$

The first line says that before the decision (i.e., during the interval before the first Decision message), process j will not prematurely send an OK message, and the second line says that once the Decision message has been sent each process will eventually acknowledge it.

6.2. The Graphics Interface in Action

The two-phase commit example is used below to illustrate the interaction of a user and the graphics interface. In this example, focus operations are used to manipulated the formatted screen in order to visually isolate messages and processes of interest.

6.2.1. The Focus Operations

There are two categories of focus operations. When viewing the message history, the user may rearrange the graphical image by using **Display** operations, or screen the message traffic with **Filter** operations. Picking either category with the mouse causes a pop-up menu to appear. The user then narrows his focus by selecting menu items. The user may also be required to select operands by picking tokens on the screen or items from a specific submenu.

Under the Display menu, there are two submenus: **Time** and **Process.** The **Time** submenu gives the user the choices of **Compress** and **Expand** to adjust the focus along the Time axis.

In the **Time** submenu, the user may also select **Restart,** which will cause the distributed program to start from its beginning. By requesting a restart, the user is indicating that any saved message history is to be discarded, and a new history is to be collected. Of course, restarts do not automatically invalidate previously established breakpoints or assertions.

The **Time** menu also has a **Breakpoint** selection, which is used to place a breakpoint with a mouse pick. These breakpoints are intended to allow the user to specify a point

in time and have the system restart and halt when that time point is reached. Actually, the "time" is indicated by a message event in each of the processes. Using this form of breakpoint, the user can rapidly skip an initial execution sequence and stop somewhere before a known occurrence of an error. At that point, stepping and state examination may be invoked to detect error preconditions.

Normally, processes are assigned to positions on the Process axis in order of their creation. With the **Process** submenu under **Display** operations, the user may use the mouse to move processes around on the Process axis and to group them together into subsets based on frequency of communication. This facility is useful for reducing the crossing of message lines in the display.

Under the **Filter** menu, there are three submenus: **Time, Process,** and **Message.** If the user selects **Time,** he then picks two points on the Time axis; only messages sent or received during this period are shown.

Under the **Process** submenu, the user may specify types of senders or receivers he is interested in; he may also indicate process numbers of specific senders or receivers. The user may also pick Sum, meaning that the number of processes meeting the restrictions he has selected are to be totaled. In this case, the response to the filter operation is a number, not a message display.

Also under the **Process** submenu is the choice of selecting a *chain,* which is a subset of the available processes. This subset is dynamically determined by the actual communications that occur. The user specifies one or more processes that are to start the chain. As execution progresses, any process that is sent a message by one of the chain processes is itself added to the chain. This facility is often useful for monitoring a causally linked set of messages. For example, one may wish to separate a request acknowledgement sequence out of a mass of other message traffic. This is easy using a chain from the initial requestor.

Some of the chains formed with this mechanism may later be used as the basis for temporal logic assertions to be checked automatically. In essence, a chain is a simple way of specifying a temporal order on a sequence of message events over some set of processes. If these chains are augmented with more information about the message types that start them, they could then serve as reasonable temporal assertions.

In the **Message** submenu under **Filter,** the user may request to see certain message types, contents, and the sum of the number of messages which meet the other conditions specified. Predicates are used to indicate the contents of messages of concern. Predicates may contain comparator, logical, arithmetic, and aggregate operators. The aggregate operators include Max, Min, and Count. If arithmetic operators or the Count operator is used in a predicate, the result is a numeric or character query response, and not a message display.

6.2.2. A Sample Session

Figure 3 contains selected screen images from an example scenario. It should be noted that, for brevity, time sequences of pop-up menus have been condensed; often, more than one menu appears, but in actuality, Idd shows these menus sequentially.

In our scenario, Mr. Smith is trying to debug his two-phase commit program. He begins in **Monitor** mode and casually watches messages fly by until the system is suspended. The assertion stating that the coordinator should decide to commit if and only if every participant agrees to commit has not been met. Examination reveals that the coordinator has commited, so that means that some participant has voted to abort. It should be noted that the user may supply English versions of his assertions, in order to simplify the display. See figure 3a.

At this point Mr. Smith has two hypothesis about the failure. Either (1) the coordinator did in fact receive a commit vote from every Participant process, or (2) the coordinator committed even though it received an abort vote from some process. Mr. Smith decides to investigate the first possibility. In figure 3b, Mr. Smith filters out the number of processes of type Participant, and notes that there are ten of them (figure 3c). In figure 3d, he sets the filter to count the messages of type "vote". As shown in figure 3e, he discovers the problem; there are eleven votes. Some process has voted twice. Figures 3f and 3g show Mr. Smith locating the individual who voted twice. Apparently, process number 16 voted to commit. Then, after the coordinator had received ten commit votes, 16's second vote - an abort - came through. Process 16 needs to be debugged.

7. Discussion and Related Work

There are a number of published proposals for distributed debuggers. DAD (Do-All-Debugger) [Victor 76] is one example of a distributed debugger for the National Software Works project. As with the BLIT debugger [Cargill 83] and others [Baiardi 83, Bates 83, Weber 83], the emphasis is on the state of the various processes and the events defined by changes in that state. Some work has been done on characterizing the sequences of messages in concurrent programs [Riddle 78, Garcia-Molina 81, Snodgrass 84]. The first of those was based upon regular expressions, and the resulting assertions were very large and clumsy. This is principally because of the possible interleavings of messages. The Garcia-Molina paper uses a database formalism to manipulate the messages produced in the system. We believe that interval logic can provide more expressive assertions for message passing patterns than either regular expressions or database query languages. Snodgrass's work is perhaps closest to ours and supports the importance of temporal assertions. He provides a database query language (TQUEL) that includes some temporal concepts. His work appears oriented

more towards straight monitoring and not debugging, and it is not clear how the relative power of his temporal query language matches with interval logic.

8. Summary

This paper outlines a novel dynamic debugging system for a distributed environment. It uses a modified temporal logic, interval logic, as the language for expressing the constraints to be monitored by the debugging system. It provides a sophisticated graphics interface for displaying and filtering the debugging information.

[Baiardi 83] F. Baiardi, N. De Francesco, E. Matteoli, S. Stefanini, and G. Vaglini.
Development of a Debugger for a Concurrent Language.
Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging, (Pacific Grove CA, March 1983)
Published as *SIGPLAN Notices 18* (8):98-106 (August 1983).

[Bates 83] P. Bates, and J. C. Wileden.
An Approach to High-level Debugging of Distributed Systems.
Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging, (Pacific Grove CA, March 1983)
Published as *SIGPLAN Notices 18* (8):107-111 (August 1983).

[Cargill 83] T. A. Cargill.
The Blit Debugger (Preliminary Draft).
Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging, (Pacific Grove CA, March 1983)
Published as *SIGPLAN Notices 18* (8):190-200 (August 1983).

[Gabbay 80] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi.
On the Temporal Analysis of Fairness.
Conference Record of the 7th Annual ACM Symposium on Principles Of Programming Languages (Las Vegas), pages 163-173, ACM SIGACT-SIGPLAN, January 1980.

[Garcia-Molina 84] H. Garcia-Molina, F. Germano, Jr., and W. H. Kohler.
Debugging A Distributed Computing System.
IEEE Transactions on Software Engineering, SE-10(2): 210-219 (March 1984).

[Gray 78] J. N. Gray.
Notes on Data Base Operating Systems.
In R. Bayer, R. M. Graham, and G. Seegmuller, (eds.), *Operating Systems: An Advanced Course. Lecture Notes in Computer Science Volume 60*, Springer Verlag, 1978, pages 393-481.

[Harter 85] P. K. Harter, Jr.
On-line Testing of Interval Logic Formulae.
In preparation.

[Lamport 80a] L. Lamport.
"Sometime" is Sometimes "NOT Never": On the Temporal Logic of Programs.
Conference Record of the 7th Annual ACM Symposium on the Principles Of Programming Languages (Las Vegas), pages 174-185, ACM SIGACT-SIGPLAN, January 1980.

[Riddle 78b] W.E. Riddle, J.C. Wileden, J.H. Sayler, A.R. Segal and A.M. Staveley.
Behavior Modelling During Software Design.
IEEE Transactions on Software Engineering SE-4 pp. 283-292 (July 1978)

[Schwartz 83a] R. L. Schwartz, P. M. Melliar-Smith, F. H. Vogt.
An Interval Logic for Higher-Level Temporal Reasoning.
Proceedings of the Second Annual Symposium on Principles of Distributed Computing, (Montreal, Quebec), pages 173-186, August 1983.

[Snodgrass 84] R. Snodgrass.
Monitoring in a Software Development Environment: A Relational Approach.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, published as: *SIGPLAN Notices 19* (5):124-131, (May 1984).

[Victor 76] K. B. Victor.
The Design and Implementation of DAD: A Multiprocess, Multimachine, Multilanguage Interactive Debugger.
Augmentation Research Center, Stanford Research Institute. Menlo Park, Ca, Draft (August 1976).

[Weber 83] J. C. Weber.
Interactive Debugging of Concurrent Programs (extended abstract).
Proceeding of the ACM SIGSOFT/SIGPLAN Software engineering Symposium on High-level Debugging, 20-23 March 1983, Pacific grove CA.
Published as *SIGPLAN Notices 18* (8):112-113 (August 1983).

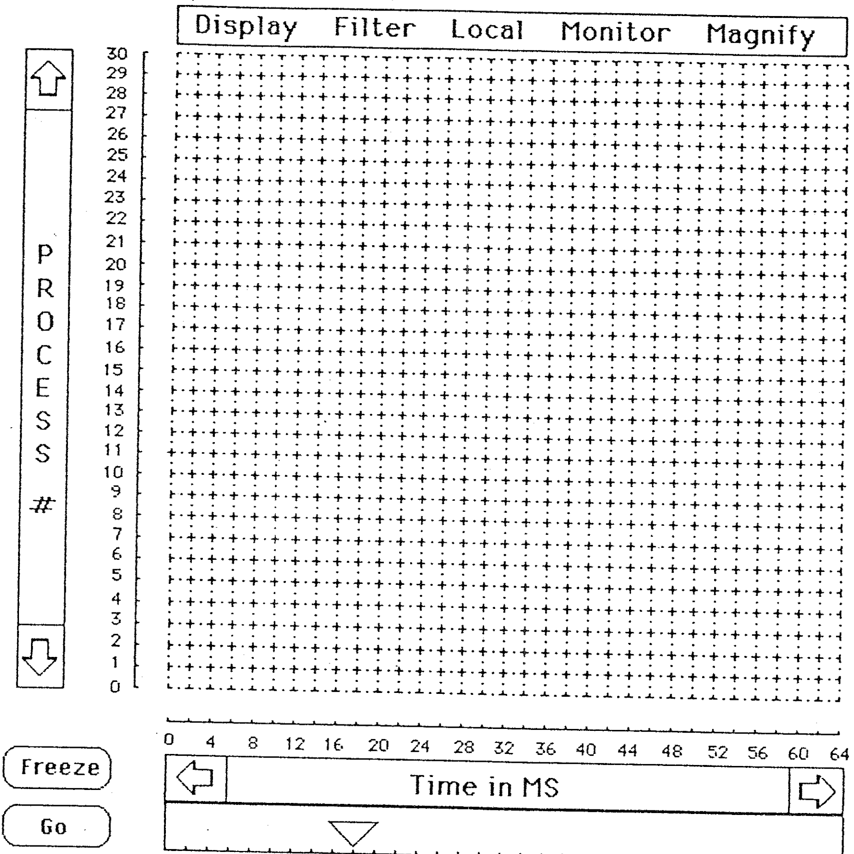


Figure 1.

```
COORDINATOR                     PARTICIPANT

begin                           begin

VOTE:
 for all participants
   send(poll)
                                VOTE:
                                 receive(poll)
                                 force data to
                                   non-volatile storage
                                 if force succeeds then
                                   send(vote,commit)
                                 else send(vote,abort)
 for all participants
   receive(vote)

DECIDE:
 if any vote is abort, then
   decision is abort
 else decision is commit

COMMIT:
 force all data to
   non-volatile storage
 for all participants
   if decision is commit then
     send(commit)
   else
     send(abort)
                           COMMIT:
                                 receive(decision)
                                 if verdict=commit then begin
                                   perform commit actions
                                   release locks
                                   send(ok)
                                   end
                                 else begin
                                   rollback
                                   send(ok)
                                   end
   for all participants
     receive(ok)

end coordinator                 end participant
```

Figure 2. Two Phase Commit Protocol (Coordinator and Participant)

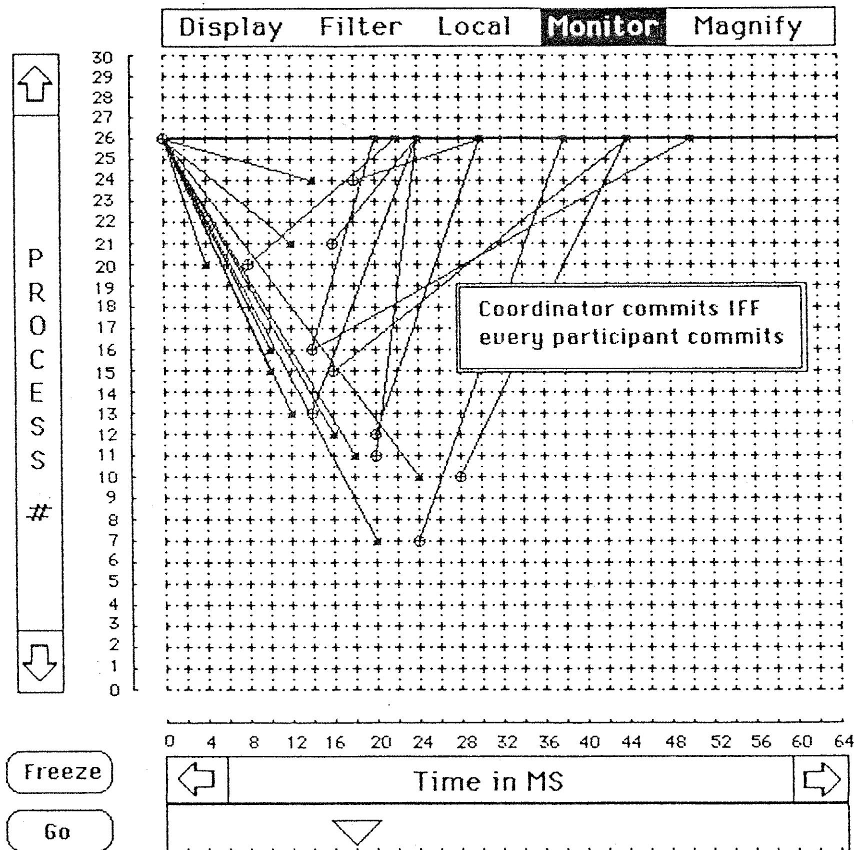


Figure 3a.

3a

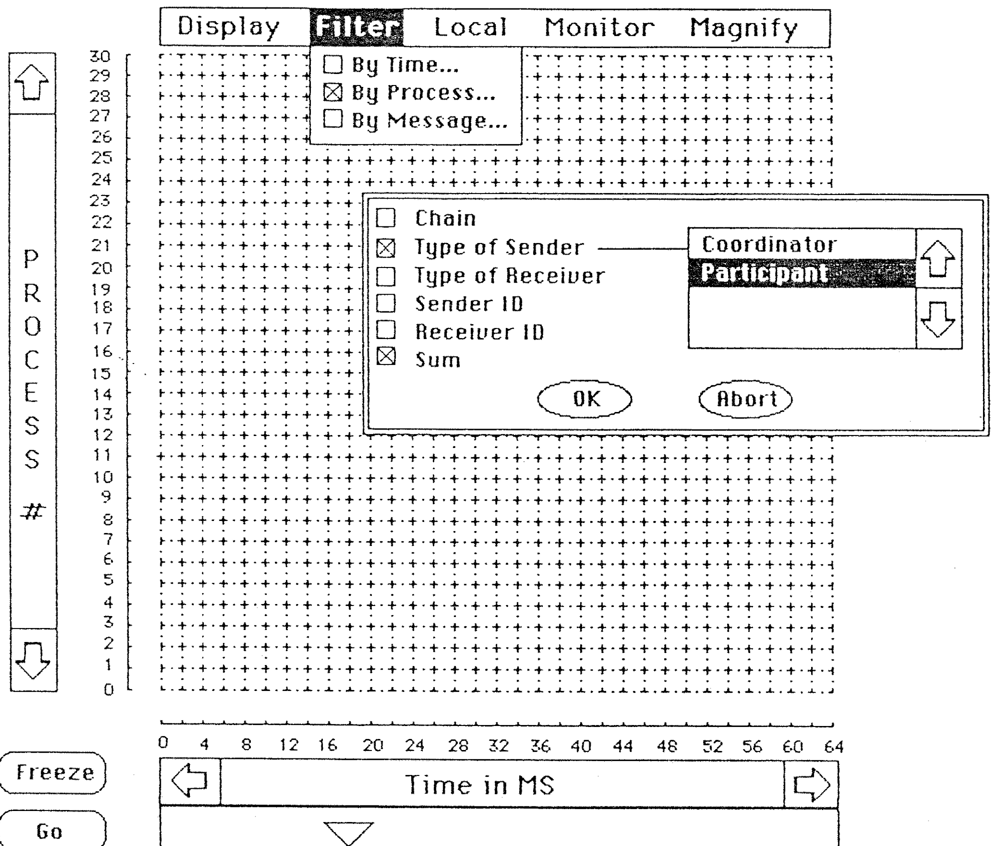


Figure 3b.

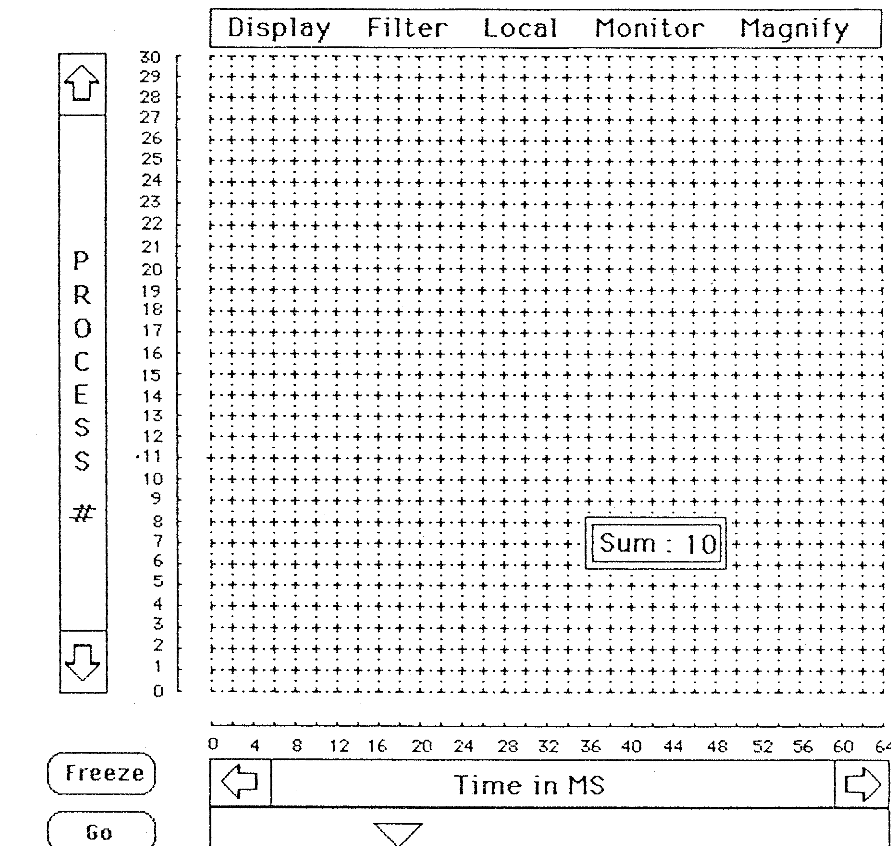


Figure 3c.

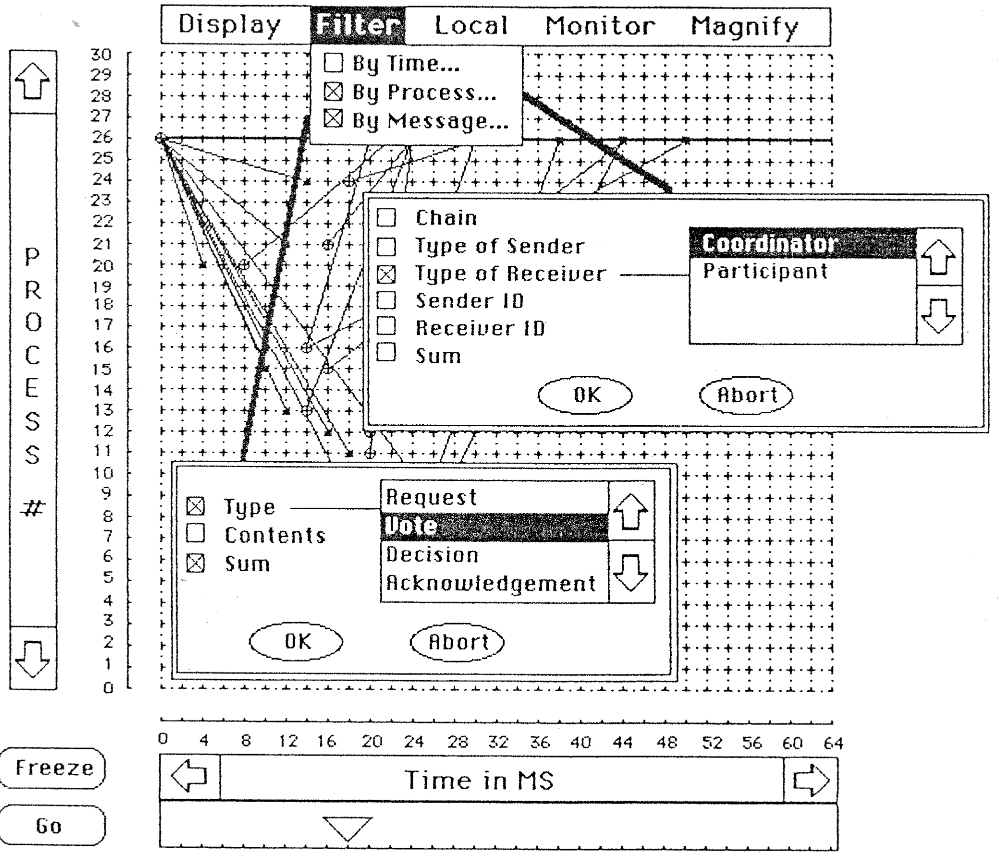


Figure 3d.

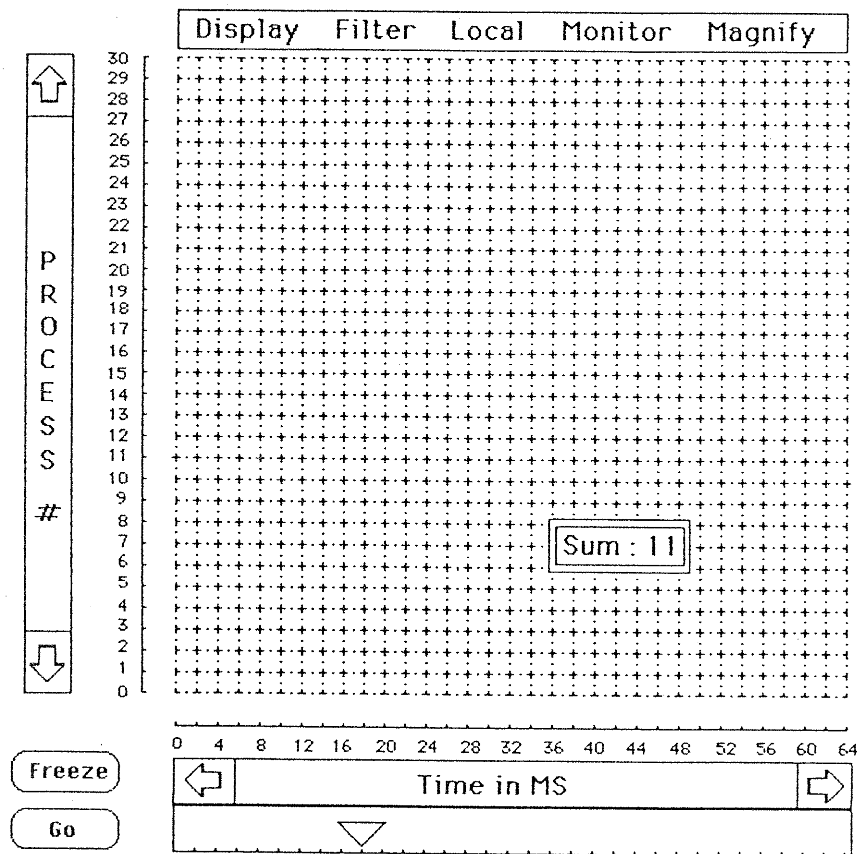


Figure 3e.

3d

3e

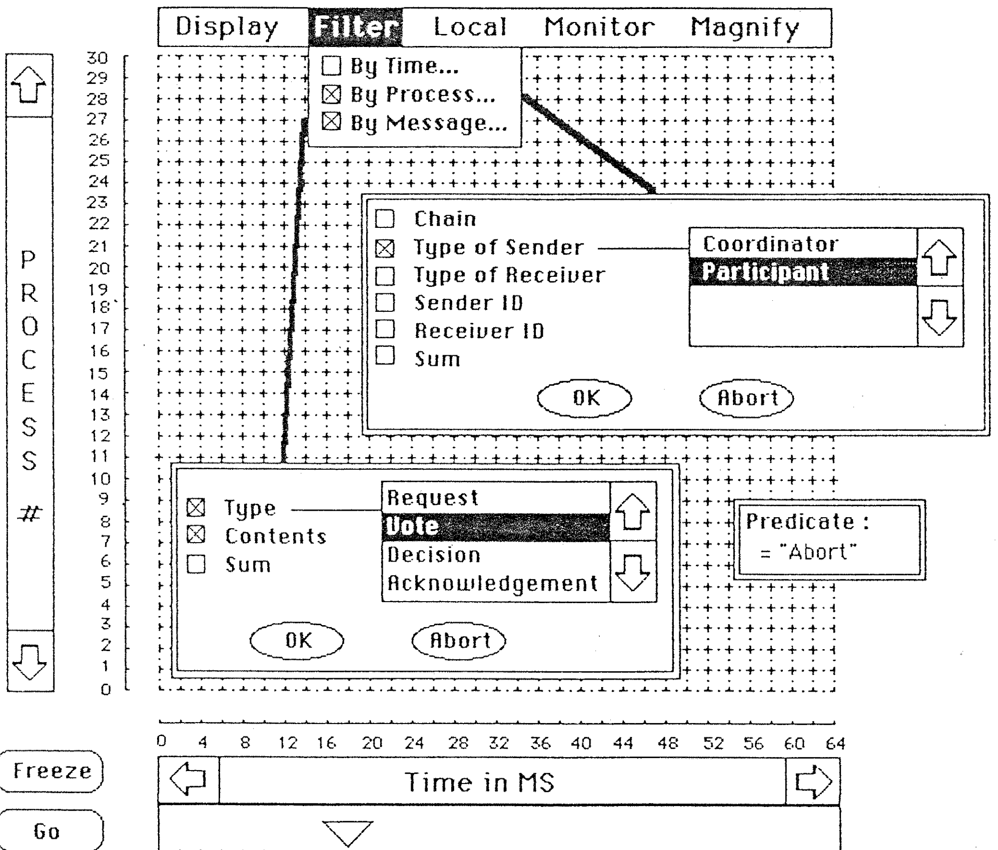


Figure 3f.

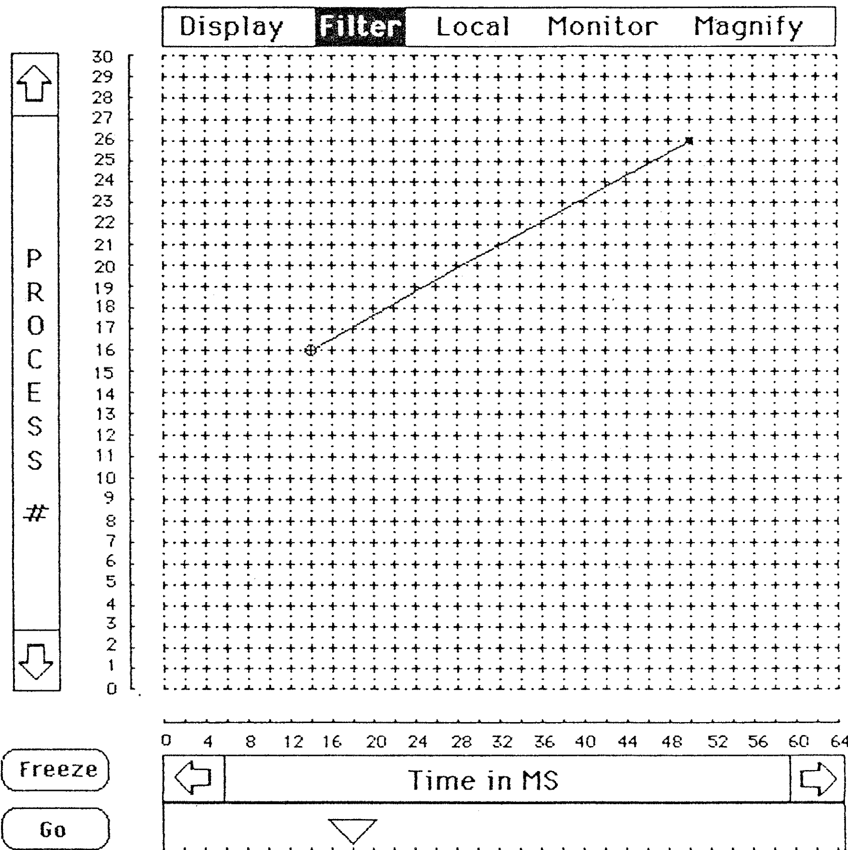


Figure 3g.