# Monitoring Distributed Systems

JEFFREY JOYCE, GREG LOMOW, KONRAD SLIND, and BRIAN UNGER
University of Calgary

The monitoring of distributed systems involves the collection, interpretation, and display of information concerning the interactions among concurrently executing processes. This information and its display can support the debugging, testing, performance evaluation, and dynamic documentation of distributed systems. General problems associated with monitoring are outlined in this paper, and the architecture of a general purpose, extensible, distributed monitoring system is presented. Three approaches to the display of process interactions are described: textual traces, animated graphical traces, and a combination of aspects of the textual and graphical approaches. The roles that each of these approaches fulfill in monitoring and debugging distributed systems are identified and compared. Monitoring tools for collecting communication statistics, detecting deadlock, controlling the nondeterministic execution of distributed systems, and for using protocol specifications in monitoring are also described.

Our discussion is based on experience in the development and use of a monitoring system within a distributed programming environment called Jade. Jade was developed within the Computer Science Department of the University of Calgary and is now being used to support teaching and research at a number of university and research organizations.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; D.2.4 [**Software Engineering**]: Program Verification—*assertion checkers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; *monitors*; *tracing*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*documentation*; D.4.8 [**Operating Systems**]: Performance—*measurements*; *monitors*

General Terms: Design, Human Factors, Measurement

Additional Key Words and Phrases: Concurrent monitoring, distributed monitoring, dynamic documentation, graphical monitoring

## 1. INTRODUCTION

Monitoring supports the debugging, testing, and performance evaluation of computer programs. When a program is distributed, monitoring becomes more difficult. The monitoring of distributed systems involves dynamically extracting information about the interactions among processes, collecting this information,

This research has been supported by the Strategic Grants Program of the Natural Sciences and Engineering Research Council of Canada.

Authors' current addresses: J. Joyce, Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge, U.K.; G. Lomow, K. Slind, and B. Unger, Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada T2N 1N4.

and presenting it to users in useful formats. This paper focuses on the use of monitoring tools to support the development of distributed systems that interact solely via message passing. Our discussion is based on experience with the design, implementation, and use of a monitoring system within the Jade distributed software prototyping environment [24, 27].

### 1.1 Distributed Systems and Monitoring

A distributed system is a collection of processes working together to accomplish some task. Each process is a deterministic program unit able to execute separately from, and concurrently with, other processes. There are a number of reasons why debugging, testing, and evaluating such systems are more difficult than the same activities would be for sequential programs [1, 11, 15]:

(1) A distributed system has many foci of control. Thus, sequential monitoring and debugging techniques, such as tracing and breakpoints, based on a program counter and a process state, need to be extended to be applicable to distributed systems.

(2) Communication delays among nodes in a distributed system make it difficult to determine a system's state at any given time. For example, the initiation of an attempt to determine the system's state must be made from one node, and other nodes will always be notified of this attempt at later, unpredictable times.

(3) Distributed, asynchronous systems are inherently nondeterministic. This means that two executions of the same system may produce different, but nevertheless valid, orderings of events. Therefore it is difficult to reproduce errors, and to test possible, but improbable, situations.

(4) Monitoring a distributed system alters its behavior. The behavior of a sequential program is not affected by the amount of elapsed time between the execution of two successive instructions, for example, a symbolic debugger can interrupt a sequential process at a breakpoint without affecting the process's subsequent execution. In a distributed system, stopping or slowing down one process may alter the behavior of the entire system.

(5) Interactions between the system and the system developer, the intended user of monitoring tools, can be more complex. For example, when a terminal is connected to each processor, the system developer may need to physically move from terminal to terminal to start processes, set breakpoints, and examine traces. Thus, it is necessary to provide tools that span a distributed system and can be invoked and controlled from a single site.

One problem common to both distributed and sequential monitoring is the need to make the large amounts of data produced during a monitoring session intelligible to the user. All of these problems are addressed in this paper.

### 1.2 Approaches to Monitoring

Facilities provided by Lisp environments bear directly on the problem of coping with large amounts of monitoring data [25]. There are three features, found in most Lisp environments, that can be particularly useful in monitoring distributed

systems: The ability to debug a process by tracing its execution, interactively setting breakpoints, and, while at a breakpoint, being able to examine the state of the process. Also, Lisp workstations, with their bit-mapped screens, have enabled the use of graphics to further support the user during the debugging process. These aspects of Lisp environments assist the user in interpreting large amounts of data.

EXDAMS [2], the EXtendable Debugging And Monitoring System, also addresses the problem of coping with large amounts of data in sequential systems by allowing for the display of source code and variable values of a program in a user-defined fashion. A modified compiler produces object code that generates history files. These are used by a debugging program to reenact the execution and to display information of interest to the user. This information may include the flow of control from one source line to another, the updating of variables, flowback analysis from a variable assignment to the points where other variables involved in the assignment were last given values, and anything else the user cares to incorporate into the debugger. EXDAMS illustrates the value of having a monitoring system that is extensible: New features that interpret monitoring data in different ways can be easily added.

In recent years, the proliferation of terminals with bit-mapped screens has spurred the use of graphics in helping to show what a program is doing. This work has developed in two directions: Using pictures to represent the progress of an algorithm, for example, changing the color of a picture representing a node when it is visited by a tree traversal algorithm, and using pictures to represent the state of a program. These two approaches fall under the topics of dynamic documentation and debugging. An example in the first category is [17]. Two examples in the second category are Dewlap [9] and Incense [19]. These interactive graphical techniques, as well as those available in most Lisp environments, all assist the user in interpreting monitoring data. The approaches to the display of monitoring information presented in Section 3 are strongly related to this work, particularly Dewlap (an interactive graphical Prolog debugger) and EXDAMS.

Garcia-Molina et al. describe a methodology for debugging distributed systems that relates to problems (1), (2), and (5) listed in Section 1.1 [11]. Essential aspects include: Debugging the system bottom-up so that individual modules are debugged separately and then integrated, extensive use of trace files of important events to provide process histories that can be examined to track down bugs, and two-phase debugging. Two-phase debugging entails having the user first examine trace files to find the source of the problem, and second, constructing an artificial environment in which to re-create and rectify the error. Garcia-Molina et al. also propose tools for generating and examining trace files. Trace information is kept locally on each computer and accessed using distributed database techniques. Breakpoint and stepping facilities are used to control the execution of a distributed system.

Snodgrass views monitoring as an information processing activity and asserts that the relational model, typified by relational databases, is an appropriate formalism for structuring the information generated by a distributed system [23]. He defines entities to be data structures, processes, and hardware components

and relationships to be processes running on processors and messages residing in queues. Also defined are time-varying relations between entities and relationships. Queries on this collection of relations are translated into actions, such as initiating the collection of information, to be performed by the monitor. This work focuses on the database query model so that the user can specify what information is to be collected. This approach also controls the amount of monitoring data collected, and thus mitigates problem (4) in Section 1.1.

Harrison deals with the monitoring of a target network in a way that enables simulation in a richer environment, that is, one with tools such as compilers, debuggers, and editors [12]. His aim is to minimize the impact of monitoring on the execution of the target system (i.e., problem (4)). Two tools for this are program state analysis and dynamic traces of behavior. A dynamic trace contains information regarding the actions of a program during its execution, such as interactions with other processes, internal decisions, and modifications to data structures. Harrison's suggested technique is to run the system to an error state on the target network and using program state analysis to find the subset of processes where the bug lurks. Then, the system is rerun while the affected processes are monitored to produce traces used for subsequent simulation in the development environment.

The work presented in this paper differs most from the work of Snodgrass because the Jade monitoring system collects all communication activity generated by application processes, whereas Snodgrass's method only collects information requested by the user. Our approach incurs a larger performance penalty during monitoring, but problem (4) is addressed directly by enabling the user to control nondeterministic alternatives. Our approach to problems (1) and (5) is similar to that taken in Multibug [8]; however, there are important differences in the way these mechanisms have been implemented. Harrison's work is directly realizable in the Jade monitoring system, as is that of Garcia-Molina and his co-workers.

The Jade monitoring system, described in Sections 2, 3, and 4, attempts to address all of the problems outlined in 1.1. Our approach is unique in that it provides an interactive, animated display of an executing distributed program, and it enables user control of nondeterminism. It is also extensible in that the detection and collection of monitoring information is separate from the analysis and display of this information to users. This separation permits new monitoring tools to be easily created and integrated into the monitoring system.

### 1.3 Scope and Organization of this Paper

We explore monitoring in the context of the Jade programming environment. In Section 2, the parts of Jade that are relevant to monitoring are presented, followed by a description of the architecture for a distributed monitoring system. A number of tools that display monitoring information to the user in different ways have been developed. These include three ways of viewing process interactions: a simple textual trace of interaction events, a graphical representation of the system's state that changes as state changes occur, and the display of the evolution of a process versus that of events. These tools and the role that each fulfills are identified and compared in Section 3.

Event management and analysis is addressed in Section 4. The collection of statistical data regarding events and the detection of deadlock are briefly described. Next, the interactive control of nondeterminism and the collection of information that enables the re-creation of system states are presented. Finally, the use of protocol specifications for run-time protocol checking and for defining abstract events is examined. Our conclusions appear in Section 5.

## 2. A DISTRIBUTED MONITORING SYSTEM

An extensible, distributed monitoring system has been designed, implemented, and evaluated within the Jade programming environment. The monitoring system supports the observation and control of message passing within a distributed application system that consists of a set of concurrently executing processes. First, an outline of both Jade and its interprocess communication (IPC) facility is presented, and then the monitoring system is described.

### 2.1 The Programming Environment—Jade

The Jade environment supports the development of distributed software [24, 27]. The key components of Jade are a multilingual IPC facility, a window system, a hierarchical graphics package, an interactive graphics editor, and a distributed monitoring system. The monitoring system, in conjunction with the other components, enables a system of processes spanning multiple machines to be observed and controlled from a single workstation.

The Jade IPC facility [20] is called Jipc (pronounced "gypsy"). Jipc is implemented as a Unix[1] device driver and currently runs on Vax 11/780 and Sun Microsystem Unix 4.2/4.3 Berkeley Standard Distribution (BSD) hosts. A standalone version of Jipc that incorporates a multitasking kernel runs on Corvus Concepts, Cadlinc Suns, and several other workstations. The window system is implemented as a distributed Jipc system, for several different types of diskless workstations, as well as on Sun Microsystem's window system within Unix 4.2/4.3 BSD.

The Jade window system permits the user to create and manipulate windows using a mouse and pop-up menus. A window is capable of serving as a virtual terminal to a Unix host and as an interface to Jipc processes. This enables the user from a single workstation to interact with processes running on different machines. Application programs can add their own pop-up menus and are able to obtain mouse or keyboard input from the window system. This allows system and user-written tools to share a consistent user interface.

The Jade graphics system provides routines for creating and manipulating hierarchical, two-dimensional pictures. Hierarchical pictures consist of both primitives (e.g., points, lines, boxes, circles, and text) and other pictures. This hierarchical picture structure permits a picture and all of its component subpictures to be transformed (i.e., translated, scaled, or rotated) by applying a single transformation to the node in the hierarchy that contains the picture. The graphics system maintains an internal model of the hierarchy so that incremental

[1] UNIX is a trademark of AT&T Bell Laboratories.

changes to the model result in incremental changes to the display. This enables efficient screen updating for real-time applications such as the run-time display of monitoring and simulation information [30]. The graphics editor facilitates the creation of pictures that can then be used to represent specific states of an executing distributed program. These pictures can be animated, for example, via messages sent by the monitoring system or by the application processes.

### 2.2 The Interprocess Communication Protocol—Jipc

A Jade distributed system, or *Jipc system*, consists of a set of processes that communicate using the Jipc message-passing protocol and that execute on one or more Unix hosts and workstations [20]. Jipc defines the protocol for all communication among processes as well as providing primitives for dynamic process creation, destruction, and searching. Interfaces to Jipc currently exist for the Ada, C, Lisp, Prolog, and Simula programming languages.

Jipc processes communicate by exchanging messages through the use of a blocking protocol based on Thoth [6]. The *send* primitive transmits a message from a sending process to a destination process and blocks the sender until it receives a reply message. A process receives a message by calling either *receive* (receive a message from a specified process) or *receive_any* (receive a message from any process); if no messages are waiting, the receiver is blocked until one arrives. After receiving and processing a message, a process can either transmit a message to the sender using the *reply* primitive, or it can pass this responsibility onto another process using the *forward* primitive. When the reply message arrives at the original sender, the sender is unblocked and allowed to continue executing. Both reply and forward are nonblocking.

A Jipc message consists of a sequence of typed data items. The types supported are integer, floating-point, character, string, process id, byte-block, and atom. When necessary, the Jipc kernel converts values of these data types between the representations used by the different machines and programming languages supported within the Jade environment.

Several aspects of Jipc support the building of monitoring tools. The fact that Jipc processes are loosely coupled and only communicate via message passing permits interprocess events to be easily defined and detected. Typed data items in Jipc messages enable the contents of a message to be displayed intelligibly. Since processes can be written in different programming languages, components of a Jipc system can be implemented in a language suited to the application, for example, simulation components written in Simula, interactive components written in Lisp, and real-time embedded components written in Ada [16].

A two-process example Jipc system is shown in Figures 1 and 2. Process Sender, implemented in Lisp, sends a message to process Receiver, implemented in Ada. Process Receiver receives the message and issues a reply to process Sender. A textual trace for the two-process system is shown in Figure 3.

### 2.3 Monitoring System Architecture

Our experience with the Jade monitoring tools suggests that users nearly always need a variety of tools during the development of an application system, and that these tools must range from detailed, low-level tools to highly abstract,

```
(j_enter_system "Sender")                    become a Jipc process named "Sender"
(setq pid (j_search_machine "vaxb" "Receiver"))  find the receiving process
(j_puti 16)                                  put 2 integers in message
(j_puti 17)
(j_send pid)                                 send message to "Receiver"
(setq result (j_geti))                       get the result
(j_leave_system)
```

Fig. 1. Lisp code for sender process.

```
with a_jipc;
use a_jipc;
procedure Receiver is
  i1, i2 : integer;
  pid    : j_process_id;
begin
  j_enter_system("Receiver");      become a Jipc process named "Receiver"
    pid := j_receive_any;          receive a message from any process
    i1 := j_geti;                  get 2 integers from message received
    i2 := j_geti;
    j_puti(i1 + i2);               send sum as reply message
    j_reply(pid);
    j_leave_system;
end Receiver;
```

Fig. 2. Ada code for receiver process.

application-specific tools. To accommodate this diversity and to encourage the implementation of new monitoring tools, the Jade monitoring system was designed to be extensible. This extensibility has been achieved by separating the task of detecting and collecting information from the task of analyzing and displaying this information. Thus, the writer of a new monitoring tool is not concerned with how the monitoring information is collected but only with interpreting and presenting that information to users.

The architecture of the monitoring system is illustrated in Figure 4. Six application processes running on two machines, vaxa and cv01, are shown. A *Channel* process resides on each machine being monitored, and it collects monitoring information from the application processes executing on that machine. A channel distributes this information to one or more *Consoles* (the consoles may be running on different machines), and each console receives information from one or more channels. A console examines and interprets the monitoring information it receives and then presents it to the user. The flow of monitoring information from application processes to consoles is transparent to application processes and does not affect the way in which they communicate with each other via Jipc.

2.3.1 *Monitorable Processes and Events.* A Jipc process can be either *monitorable* or *unmonitorable* depending on whether it is loaded with the version of Jipc that incorporates monitoring. Processes loaded with the monitorable version of

```
 1. vaxb.Receiver enters the Jipc system
 2. vaxb.Receiver waits to receive a message from any process
 3. cv01.Sender enters the Jipc system
 4. cv01.Sender searches vaxb for a process with the name 'Receiver'
 5. cv01.Sender finds vaxb.Receiver
 6. cv01.Sender sends a message to vaxb.Receiver
      16 17
 7. vaxb.Receiver receives a message from cv01.Sender
      16 17
 8. vaxb.Receiver replies with a message to cv01.Sender
      33
 9. vaxb.Receiver continues after replying to a message
10. cv01.Sender receives a reply from vaxb.Receiver
      33
11. vaxb.Receiver attempts to leave the system
12. cv01.Sender attempts to leave the system
```

Fig. 3. Textual trace for two-process example.

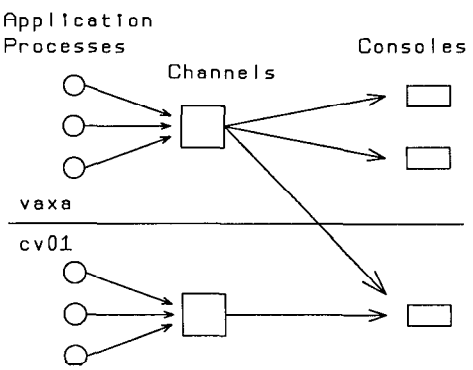


Fig. 4. Monitoring system architecture.

Jipc suffer a slight execution speed penalty. Typically, processes under development are monitorable, whereas system processes and application processes, which have already been tested or that are installed in a production environment, are unmonitorable.

A *monitorable event* is defined as any Jipc process operation that may have an effect outside of that process. A monitorable event occurs whenever a process initiates or completes any of the following operations: entering or leaving a Jipc system, creating or killing a process, searching for another process to acquire its process identifier, and message sends, forwards, receives, and replies. An event also occurs when one of these operations fails. The internal actions of a process, such as local computations or the manipulation of a message buffer, are not events.

At compilation time, an application process is loaded with either the monitorable version of the Jipc library or the unmonitorable version. The detection of events is embedded in the monitorable version of the library. The processes supporting monitoring, (channels, controllers, and consoles) are necessarily compiled with the unmonitorable version to prevent them from monitoring

themselves. Application processes may also be compiled with the unmonitorable version for security or efficiency reasons. This approach ensures that events are collected in a transparent and consistent manner and that source code changes are not necessary to generate monitorable events.

When an event is detected in a monitorable process, information concerning this event is sent to the channel process that is executing on the same machine. The sequence of events generated by an application process forms an *event stream* that is merged, by the channel, with the event streams generated by other monitorable processes to form a channel event stream.

The events generated by the Receiver process in the two-process example of Section 2.2 are entering the system, waiting to receive a message from any process, actually receiving a message from process Sender, replying to the message, continuing after replying, and leaving the system. Similarly, the Sender process produces the following events: entering the system, searching for process Receiver, sending a message to process Receiver, receiving a reply to this message, and leaving the system. These events are shown in Figure 3.

2.3.2 *Communicating Monitoring Information.* A problem in the design of a monitoring system is deciding how to collect events generated by application processes. That is, an IPC for monitoring information must be defined. One approach is to use a different IPC mechanism than the one used by the application processes. An alternate solution is to use the same IPC mechanism used by the application processes. The latter approach makes the monitoring system more portable because it relies on only one message-passing mechanism. A problem with this approach is that the monitoring system must be able to distinguish between messages used to pass information among application processes and messages used to pass monitoring information.

We have adopted the second approach; the Jade monitoring system is a Jipc system. When an event is detected in an application process, a Jipc message containing information about the event is sent to the local channel. The monitoring system can distinguish between messages being used to pass monitoring information and messages being passed among application processes because only the former are sent from application processes to channels.

2.3.3 *Controllers.* When an event is about to occur in a monitorable process, monitoring information is conveyed to the channel. Since the Jipc send is blocking, the application process is blocked, and the event cannot occur until the channel replies to the process. As soon as the channel replies, the event is allowed to occur. Normally, a channel replies to a monitoring message before receiving any other monitoring messages. However, a user can control the order of events by introducing another monitoring process called the *Controller.* When a controller exists, all channels forward their monitoring messages to the controller. The controller can then postpone replying to a monitoring message, thereby suspending the application process generating that event. It can continue to receive other monitoring messages without causing an illegal sequence of events. The user can interact with the controller, as described in Section 4.2.

Figure 5 shows a configuration of the basic monitoring system that includes a controller. A system can contain only one controller; its purpose is to serve as a

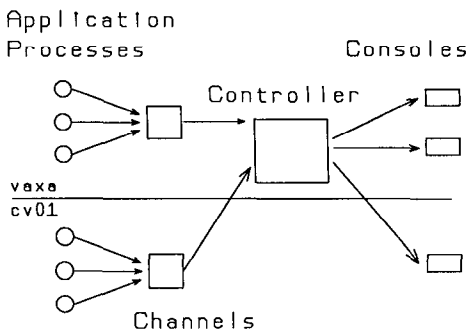


Fig. 5. The monitoring system with controller.

central site through which all events reported to the channels must pass before they are distributed to the consoles. A controller can be started or terminated at any time without affecting the events received by running consoles.

*Consoles* are processes that plug into one or more channel event streams. Consoles collect, interpret, and display event information and serve as the interface between users and the monitoring system. Each type of console interprets the channel event streams it receives and communicates the activities of the application system to the user in a different way. The simultaneous use of different consoles provides the user with different views of an executing distributed program.

2.3.4 *Consoles.* When a console is started, the user supplies a list of machine names that are to be included in the monitoring session. The underlying monitoring system is then responsible for either creating channels on the appropriate machines or linking this new console to already existing channels. Consoles can be started or terminated at any time and on any machine without affecting the events received by other Consoles. When a monitorable process enters a Jipc system, or is created, it is automatically included in any monitoring session active on its host machine.

Easy prototyping and testing of new consoles was a central goal in the design of the Jade monitoring system. This was another motivation for separating the detection and distribution of events from the implementation of the individual consoles. Monitoring information is collected automatically, and all consoles receive monitoring information in a predefined format from a single controller or from several channels. This removes data collection concerns from the development of consoles. Consoles for displaying individual Jipc events (Sections 3.1, 3.2, and 3.5), accumulating execution statistics and detecting deadlock (Section 4.1), re-creating previous executions (Section 4.2), and run-time, communication protocol checking (Section 4.3) have been built.

## 3. THE DISPLAY OF MONITORING INFORMATION

A monitoring *trace* can be defined as the depiction of communication events occurring in a distributed system [11]. A major part of any monitoring system is how a trace is presented to the user. Graphical display terminals expand the alternatives available for the display of interprocess communication events. In this section, approaches to the depiction of such traces, based on textual and graphical user interfaces, are presented and compared.

A simple example of a distributed banking system is used to illustrate the different approaches to the display of monitoring information. Individual bank accounts are held at geographically separated banks, forming a distributed database. The banking system consists of branch computers and personal banking machines. Transactions are initiated at personal banking machines that communicate with the local branch computer. If the account involved in the transaction is maintained at the local branch, then the transaction is handled by the branch computer; otherwise, the transaction is forwarded to the branch where the account is maintained.

There are two types of transactions that the user can engage in: deposits and withdrawals. For each transaction the user must provide an account number, an account type, and the amount of money being deposited or withdrawn. The banking system implements these transactions with a protocol that involves the following types of messages: *connect*, *security check*, *deposit*, *withdrawal*, *disconnect*, *error*, and *success*. Branch computers are distinguished from personal banking machines by capitalization: *Calgary* is the name of a branch computer, whereas *calgary* is the name of a personal banking machine that communicates exclusively with Calgary.

### 3.1 Textual Traces—A Text Console

A *Text Console* has been developed that reports each event in the event stream with one or two lines of textual output. The name of the process that initiated the event, the event type, and the name of the process that is the subject of the event, if any, are written on the first line. If the event is one in which processes communicate, the contents of the message are printed as the second line of output. A textual trace for the example banking system is shown in Figure 6.

A textual trace provides little more than what would be achieved by printing debugging information at strategic points in each process. However, the user is not required to insert monitoring statements because events are detected and reported automatically by the monitoring system. Thus, the possibility of introducing errors while inserting monitoring statements into each process is eliminated, and consistent monitoring information is provided to consoles.

Facilities for *event filtering*, *breakpoints*, and *execution histories* are included in the text console for assisting the user in dealing with the large quantities of information produced by the monitoring system. Each of these facilities depends on pattern matching in the event stream. There are two types of patterns: Process patterns and event patterns. A *process pattern* is an expression that identifies a process or group of processes. An *event pattern* identifies an event or group of events and may include a process pattern. For the distributed banking system some example event patterns are

| EVENT PATTERN | | | MATCHES |
|---|---|---|---|
| calgary: | send | Calgary | calgary sending to Calgary |
| *: | reply | Vancouver | any process replying to Vancouver |
| *: | comm | | all interprocess communication events |

The leftmost column entries are examples of process patterns, for example, calgary. Event filtering uses event-stream pattern matching to determine which events to display: When the console receives an event that matches one of the

1. vaxc.Calgary enters the system
2. vaxc.Calgary waits to receive a message from any process
3. vaxa.Vancouver enters the system
4. vaxa.Vancouver waits to receive a message from any process
5. vaxb.Edmonton enters the system
6. vaxb.Edmonton waits to receive a message from any process
7. vaxd.Saskatoon enters the system
8. vaxd.Saskatoon waits to receive a message from any process
9. vaxd.saskatoon enters the system
10. vaxd.saskatoon sends a message to vaxd.Saskatoon "connect"
11. vaxd.Saskatoon receives a message from vaxd.saskatoon "connect"
12. vaxd.Saskatoon replies with a NULL message to vaxd.saskatoon
13. vaxd.saskatoon receives a NULL reply from vaxd.Saskatoon
14. vaxd.Saskatoon continues after replying to a message
15. vaxd.Saskatoon waits to receive a message from any process
16. vaxb.edmonton enters the system
17. vaxb.edmonton sends a message to vaxb.Edmonton "connect"
18. vaxb.Edmonton receives a message from vaxb.edmonton "connect"
19. vaxc.calgary enters the system
20. vaxc.calgary sends a message to vaxc.Calgary "connect"
21. vaxd.saskatoon sends a message to vaxd.Saskatoon "security code" 200
22. vaxd.Saskatoon receives a message from vaxd.saskatoon "security code" 200
23. vaxd.Saskatoon sends a message to vaxa.Vancouver "security code" 200
24. vaxa.Vancouver receives a message from vaxd.Saskatoon "security code" 200
25. vaxa.Vancouver replies with a message to vaxd.Saskatoon "success" 5
26. vaxd.Saskatoon receives a reply from vaxa.Vancouver "success" 5
27. vaxa.Vancouver continues after replying to a message
28. vaxd.Saskatoon replies with a message to vaxd.saskatoon "success" 5
29. vaxa.Vancouver waits to receive a message from any process
30. vaxd.saskatoon receives a reply from vaxd.Saskatoon "success" 5
31. vaxd.Saskatoon continues after replying to a message
32. vaxd.Saskatoon waits to receive a message from any process
33. vaxd.saskatoon sends a message to vaxd.Saskatoon "withdrawal" 200 "savings" 100
34. vaxd.Saskatoon receives a message from vaxd.saskatoon "withdrawal" 200 "savings" 100
35. vaxd.Saskatoon sends a message to vaxa.Vancouver "withdrawal" 200 "savings" 100

Fig. 6. Textual trace of distributed banking system.

filters that the user has specified, that event is displayed. This enables the user to interactively specify the set of events to be displayed by the text console.

Breakpoints are also event patterns specified by the user. When an event matching a breakpoint occurs, monitoring is suspended and control is given to the user. The value of breakpoints is that they free the user from having to constantly watch the console to detect important events. Breakpoints can also be used to detect impossible events or events that signal error conditions. It is easy to determine the program state of any process when stopped at a breakpoint because a sequential debugger can be invoked from the text console.

The history mechanism allows the user to reexamine a specified number of previous events, in the order of occurrence, for a set of processes defined by a process pattern. The history facility is a particularly useful adjunct to textual monitoring because it allows the user to easily determine how a process, or set of processes, reached a specific state. It also permits events that have scrolled off of the screen to be redisplayed. The history mechanism maintains a fixed-length copy of the console event stream that is periodically truncated. The most recent event generated by each process is also maintained in a separate history structure that is not truncated, enabling the most recent action of all processes to be obtained at any time.

When trying to understand a large distributed system, it is important for the user to be able to focus on only those processes and events that are of immediate relevance, without having extraneous events cluttering the display. Since both event filtering and breakpoints can be altered while monitoring, the user can readily change the focus of the monitoring session. The addition of these facilities to the basic text console makes it a very useful tool, qualitatively different from the graphical console presented next.

### 3.2 Graphical State Displays—The Mona Console

Mona provides the user with an animated graphical view of the event stream. Whenever Mona receives an event, it updates a picture that represents the current state of interprocess communication in an application system. Mona has been implemented on the Jade window and graphics systems; an early version was described in [14].

Each update to the picture results in the display of a new *frame*. A frame describes the current state of the application system; successive frames present successive states. The sequence of frames is called a *movie*. The graphics package only updates the portion of the picture that is actually changed, so the screen is not completely redrawn for each new frame. Mona requires a bit-mapped screen, along with an input device such as a mouse, for pointing to locations on the screen. Many of the details found in the textual trace are not available with Mona; for example, the contents of messages are not shown.

A sequence of frames from a simple example is shown in Figure 7. The first frame (7a) shows the application system with process Receiver running on the machine named "vaxb." In the second frame (7b), the small circle inside the larger circle indicates that process Receiver is waiting to receive a message. In the next frame (7c), process Sender has entered the system running on "cv01." In frame 7d, process Sender has sent a message to process Receiver. The dashed

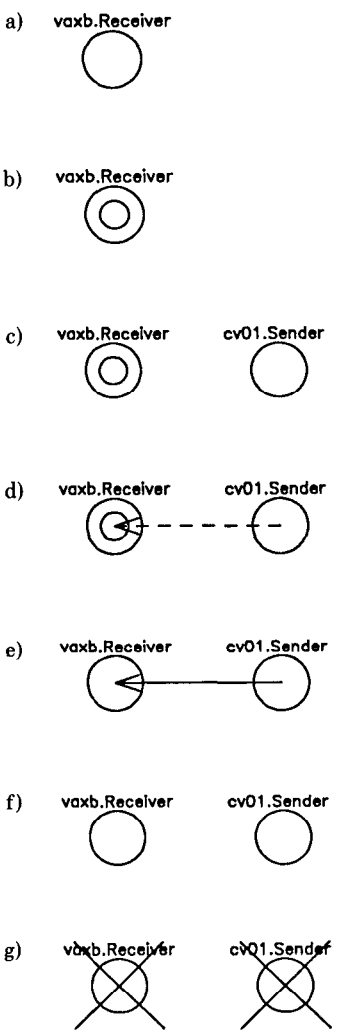


Fig. 7. Graphical frames corresponding to textual trace.

arrow indicates that the message has been sent but not yet received by process Receiver. The fifth frame (7e) shows that process Receiver has received the message from process Sender; the dashed arrow turns solid, and the small circle disappears. When process Receiver replies to process Sender (7f), the solid arrow is removed. Finally, as each process leaves the system, it has a cross drawn over it (7g). Figure 8 shows the frame from the distributed banking system that corresponds to the system state after the events shown in Figure 6 have occurred.

When a process comes into existence, Mona, by default, places its icon on the circumference of a series of concentric circles. This often results in an arrangement of icons that does not reflect the structure of the system. To alleviate this problem, Mona allows the user to reposition icons using the mouse. An arrangement of icons can be saved, and subsequent invocations of Mona can use these previously defined arrangements when deciding where to position icons.

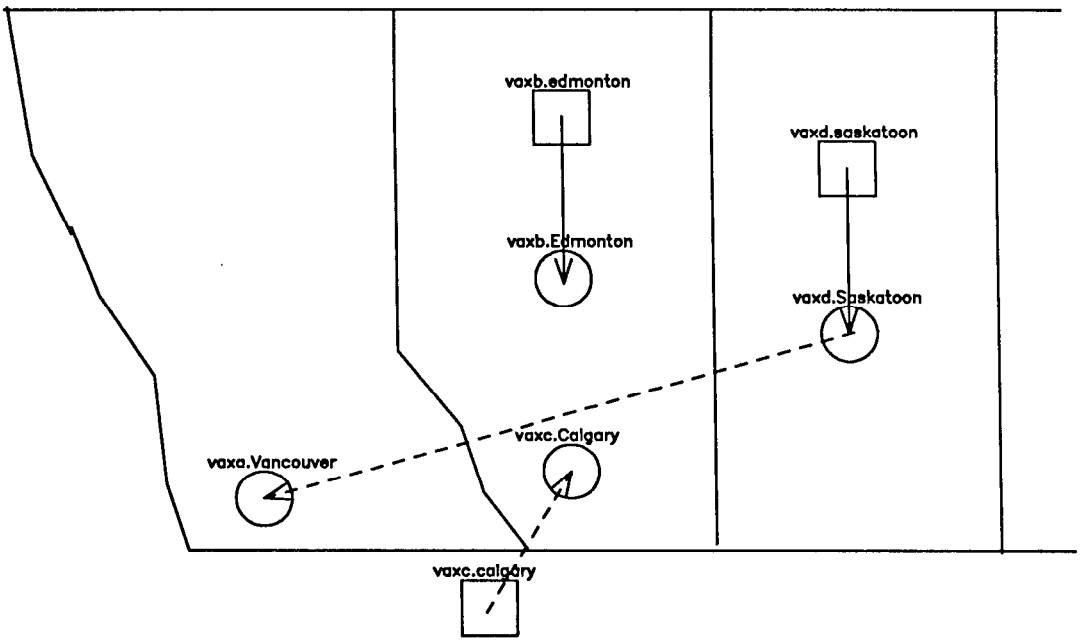


Fig. 8. Mona frame of the distributed banking system.

### 3.3 Display Management in Mona

The design of a distributed system is often structured hierarchically so that a collection of processes provides a single service or implements a single function. Furthermore, individual processes and groups of processes are often combined to form larger units. During application system development and debugging and the demonstration of the system's operation, the user will, at times, want to focus on the internal workings of a collection of processes and, at other times, will want to regard a collection as an indivisible unit. The display management facilities of Mona are able to reflect the system's structure so that its execution can be viewed at levels of abstraction above the IPC level. These facilities support monitoring and debugging after development has moved beyond the IPC protocol level.

In Mona, a *group* is defined to be a collection of entities in which each entity is either a group or a Jipc process. A group is created by using the mouse to define the opposite corners of a box that physically encloses the processes and groups that are to constitute the new group. Groups can be created, removed, and incorporated into other groups, and a group can be repositioned as an indivisible unit. The grouping of processes is discussed in [7] as a programming and kernel optimization aid, whereas here it is used as a mechanism to simplify a large, complex display.

A Mona group may be either *open* or *closed.* An open group is delimited by a dashed-line box; the interactions among the top-level entities of an open group are displayed. A closed group is delimited by a solid-line box. None of the internal

interactions among the entities of a closed group are shown, and internal process icons are not depicted. An open group corresponds to a collection of entities that the user wishes to view. A closed group encapsulates entities whose internal activities are not of current interest. The rule for displaying events in Mona is to depict all *visible* events, that is, those in which the participating processes are not in subgroups having a common closed ancestor group and those in which the participants are both on the screen.

*Zooming*, the counterpart of grouping, enables the user to focus on part of an application system. There are two types of zooming, physical and conceptual. In a physical zoom the mouse is used to define a rectangular area of the screen. In a conceptual zoom a group is selected. In both cases the area or group is enlarged to fill the entire screen. Successive zoom operations are placed on a stack, so the user can zoom in and out in a hierarchical manner. Zooming in on a closed group causes it to be opened (the assumption being that the user is zooming in on the group in order to see its internal activities).

*Shrinking* is a display management aid. When the user shrinks a group, it is scaled into a small box. This physical operation does not change whether the group is open or closed. Shrunken groups may be moved, opened, closed, removed, and included inside of new groups. When a shrunken group is expanded, it regains its former size. If a shrunken group is removed, its subcomponents are automatically expanded to their previous size. A *step mode* also helps the user manage the display by requesting confirmation before the depiction of the next visible event. This alleviates the problem of events being portrayed so quickly that they flicker past, leaving the viewer unsure of what just happened.

These ideas are illustrated in Figures 9 and 10 which show a system component that consists of two producers, a buffer, and two consumers. Figure 9 shows the two producers in an open group and the two consumers in an open group that has been shrunk. Figure 10 shows the display after both the producer and consumer groups have been closed, and the entire subsystem has been enclosed in an open group. From here the user can choose to view the producer/consumer subsystem as a buffer in a larger system by closing and shrinking this group.

Schwan and Matthews describe other research into graphically displaying parallel programs [22]. They present a graphical tool that permits the user to construct and display multiple static views of a parallel program. The tool constructs a view by extracting information from a database through the use of binary relationships between objects, as in [23]. These objects represent a program component or a group of components and are similar to groups in Mona. Schwan and Matthews's tool is more flexible than Mona in the types of relationships it can show (Mona only displays the current interprocess communication state); however, their tool is unable to display information that shows how the system evolves at run time.

### 3.4 A Comparison of the Text Console and Mona

Both the text console and Mona display an event trace. The primary difference is that Mona shows the current state of interprocess communication, whereas the text console shows the last $N$ events that occurred in the system (where $N$ is determined by the size of the screen).

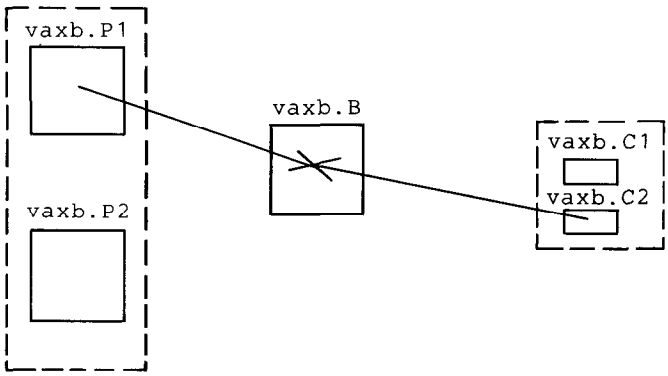


Fig. 9. Two open groups.

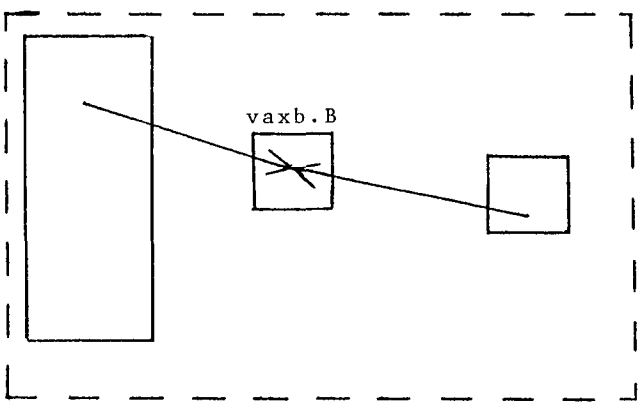


Fig. 10. Two closed groups within an open group.

When watching a system execute with the text console, all the user sees is lines of text being written out and the screen scrolling. Usually this takes place so rapidly that it is difficult for the user to read and make sense of the trace while the system is running. Thus, it is very difficult to determine the current state of an application system by watching the text console. The related problem of recognizing particular events in the event stream is handled by using filters and breakpoints.

In the text console, all events are initially displayed in a single place, whereas in Mona, events are displayed all over the screen. This means that it is difficult to know where the next event will be displayed by Mona but, at the same time, Mona gives processes and groups an identity they do not have in the text console. For example, since each process, or group of processes, has an identified site on the display, the user can determine the state of a process by looking at this site.

The text console has been used primarily for debugging systems at the IPC level. This is because the trace it produces contains all the events that have occurred during monitoring. When this trace is combined with close scrutiny of

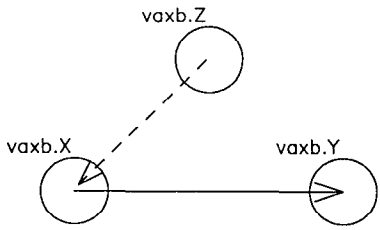


Fig. 11. Blocked system state.

the source code, the user is provided with a complete record of the system's execution that can be used in finding and correcting errors.

Mona, in contrast, always presents an up-to-date view of the system state but does not display all events and provides no indication of the sequence of events that led to a particular state; you may not know how you got there but at least you know where you are. For example, to determine why a particular process has failed to receive a message sent to it, it is necessary to look back through the textual trace to see if the process has been left blocked by a previous event. Mona, on the other hand, provides the current state to the user with a single picture that depicts all process interactions. This is illustrated in Figure 11. The Mona display clearly shows that $X$ cannot receive the message from $Z$ because it is blocked on a send to $Y$, and $Y$ has not yet replied to $X$.

Mona shows more than discrete events; general patterns of activity are communicated to the user. Indeed, these general patterns are often more informative than the details of each event. For instance, an anomaly in a general pattern may pinpoint a problem much sooner than a detailed, time-consuming analysis of each event in a textual trace. The importance of recognizing patterns as a debugging activity is discussed in [18].

Our use of graphics is based on an assumption that an observer employs mental imagery to understand system activity. Thus, a line of text describing an event is harder to understand because it must be translated into the user's mental model of the system. Mona's depiction of such an event seems to be closer to the user's mental model and requires little or no translation. This is supported by Model:

> These facts also show that sensory information is highly organised before it reaches the parts of the brain associated with abstraction, analysis, and other components of thought. The significance for monitoring facilities of these information processing characteristics of the human organism is that the pictorial, or analogical, presentation of information is often *more effective than presentation in more abstract, symbolic modes [such as text]*. [18, p. 12]

Text is an inherently more sequential medium than graphics: English text must be read left-to-right and sequentially to make sense, whereas a pictorial representation submits to direct access and focusing on the part of the picture that is of interest. This leads us to believe that it is easier for the user to assimilate the information presented by Mona than to assimilate the same information presented by the text console.

We note that some of the problems with textual traces are due to the line-by-line style of communication with users. A display terminal allowing direct modification of any part of the screen could be used to develop a better monitoring

tool that is not necessarily graphical. For example, events for different processes could be placed on separate portions of the screen, thus relieving some of the problems resulting from having these events interleaved. Finally, the textual trace, by listing one event after the other, often suggests a temporal ordering of events when no such ordering exists or at least when the order is not significant. This putative ordering may influence a developer to make unfounded assumptions or to reject reasonable hypotheses.

Whereas Mona has proved to be useful as a debugging tool, we have found that its primary value is in communicating the structure and dynamic behavior of distributed systems. The movie produced by Mona is a more effective means of explaining the structure of a system, the relationships among processes, and the general pattern of process interactions than a written or verbal explanation. This function is enhanced by the fact that the user can move and group icons representing processes in such a way that the resulting organization better reflects the user's mental model of the system. For this reason, we consider Mona to be a powerful tool for dynamic documentation.

In a discussion of pedagogical uses of computer animation, Mincy suggests that the key to understanding is visualization and proposes computer animation as a means of achieving this visualization [17]. We believe that Mona can be a valuable aid in teaching distributed systems concepts to students [5]. Also see [4], [10], [13], and [29] for further examples of the use of animation in understanding the behavior of programs and simulations.

The text console and Mona were both written for the same reason: to display a trace of the events occurring in a distributed system. We expected them to represent two different ways of doing the same thing. In fact, Mona and the text console fulfill different roles: Mona has been most useful in facilitating the understanding of a system's structure and behavior whereas the text console has been used primarily as a postmortem debugging tool.

### 3.5 An Event Line Console

Neither the text console nor Mona is able to display simultaneously both the current state of the system and the sequence of events that led to that state. In response, we have recently developed a console that displays process evolution versus events.

The *Event Line Console* displays the current state and history of each process in a compact form and, at the same time, defines the relative ordering of events. Figure 12 shows the display produced by the event line console after it has processed the events listed in Figure 6. The display is divided into three sections:

(1) On the right-hand side of the display, the name of each process is listed along with a single-letter abbreviation for that process.The abbreviation is also repeated on the left and is used to identify the process in event descriptions.

(2) In the middle section there is one row for each process. Each event line is divided into an equal number of *event intervals.* An event interval demarcates adjacent events in the console event stream; it has no relationship to the passage of real time. Each event interval displays an event; events are inserted at the right of the display and scroll to the left.

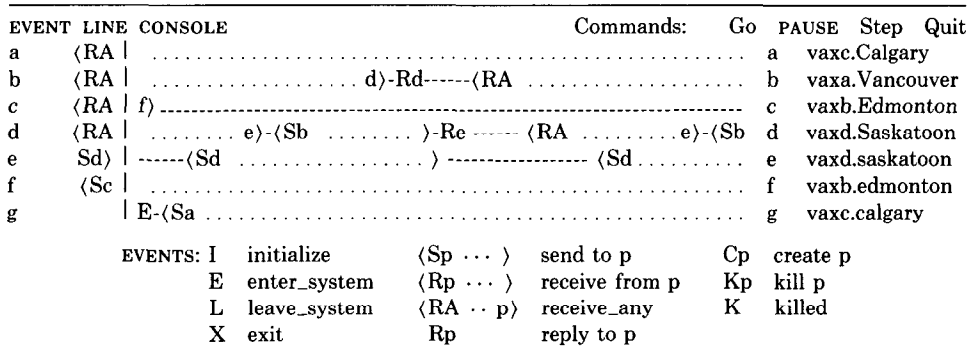


Fig. 12. Event line console trace of events listed in Figure 6.

(3) The section on the left (to the left of the vertical bar) shows the last event to scroll off the left-hand side. This event is retained so that the current state of a process is always available, even if a process has had all of its events scrolled off of the middle section (because the process is blocked or has not generated an event recently).

In the middle section, the relative ordering of events is shown by their location on the horizontal axis. This information is lost for the events displayed to the left of the vertical bar.

A process's event line is blank before it enters the Jipc system or is created and after it leaves the Jipc system or is killed. While a process is executing and not generating monitorable events, its event line is dashed (---). A dotted (...) line signifies that the process is blocked by a Jipc call. Events generated by a process are shown on its event line using the symbols listed in the legend of Figure 12 and the identifying letters of other processes. This event line console could be extended in several ways: (1) a history function could be provided by permitting the user to scroll the event lines both left and right, (2) a breakpoint facility could be provided, (3) the user could point to an event and have the message or parameters associated with that event displayed, and (4) the user could move event lines vertically so that the event lines of related processes are adjacent or grouped.

We have little experience using the event line console. Its development was motivated by the apparent preference of most users for the text console during debugging. With extensions such as those listed above, we anticipate that the event line console will be more useful than the text console.

## 4. EVENT MANAGEMENT AND ANALYSIS

The role of monitoring in the development of distributed systems can be extended by mechanisms that perform computations on an event stream, that enable nondeterminism to be controlled, and that utilize application-specific information to interpret an event stream. Mechanisms able to operate without any knowledge about the specific application system being monitored include the analysis of

interprocess communication patterns and the control and re-creation of specific execution paths. Examples of consoles that implement such mechanisms are described in Sections 4.1 and 4.2.

During the monitoring process it is inevitable that the system developer will need to observe behavior that is specific to the current application. Information can be presented to a monitoring system that enables it to interpret an event stream in a way that is relevant to a particular application distributed system. Two approaches are presented in Sections 4.3 and 4.4.

## 4.1 Communication Analysis

The consoles described in Section 3 simply display a stream of events. Tools that accept a stream of events, perform computations on this stream, and then present computed results to a user can also be implemented as consoles. One that we have implemented collects statistics on interprocess interactions, and another determines whether the state of communication among a set of processes is deadlocked.

4.1.1 *A Statistics Console.* The number and type of events that occur during the execution of an application system, as well as additional information available to the monitoring system such as message lengths, can be recorded for each process. At any time, the *Statistics Console* can be interrupted and data can be displayed either for individual processes or for the entire system. For IPC events, the statistics can be separated into local calls (the initiating and destination processes are on the same machine) and remote calls (the initiating and destination processes are on separate machines). The type and number of errors generated by each process are also recorded.

This console assists in optimizing a system at the interprocess communication level. Statistics concerning which processes communicate, how often they communicate, and average message length can aid in making decisions about system and process decomposition and the assignment of processes to processors.

4.1.2 *A Deadlock Detection Console.* The *Deadlock Detector* is a debugging tool that uses the event stream to maintain a model of the state of a Jipc system. As the deadlock detector receives each event, it updates the model and checks to see if any cycles of blocked processes exist in the model. When deadlock is detected, the user is informed, information regarding the current state of the deadlocked processes is displayed, and the system's execution is halted.

The advantages of the deadlock detector are (1) it actively monitors for deadlock; Mona, however, depends on the user to recognize deadlock; (2) in a distributed system with many processes, it can detect and identify deadlock among a small subset of the processes even though the rest of the system is operating normally; and (3) it requires no attention from the user until deadlock is detected.

## 4.2 Controlling Nondeterminism

One of the difficult problems in developing distributed systems is their inherent nondeterminism. Since events that are independent can occur in arbitrary order, a correct execution of an application system corresponds to a partial ordering of the communication events. Our monitoring system can be used to control the

order of events. This control can be used to automatically re-create a specific execution path from a recorded trace.

4.2.1 *Interactive Control.* As described in Section 2.3, a controller enables the user to determine the order in which pending events occur. Whenever a Jipc primitive is invoked by a monitorable process, a message is sent notifying the local channel. The channel first distributes this information to all consoles and then allows the monitored process to continue; that is, the monitored process is blocked until it receives a reply from the channel. If a controller is used, it is able to delay this reply to the application process, thereby preventing the monitored process from continuing.

At any given time there will be a set of pending events that are being delayed by the controller. The user can direct the controller to release a pending event or to continue receiving event information from other monitored processes and delay these events in a similar manner. At any point, the user may release a pending event that will result in the controller distributing the event to the consoles and then allowing the monitored process to continue. Thus, the system executes normally, but no event can be completed until the user allows it. Using the controller, the user is able to produce any sequence of events that the processes could possibly generate. This permits the user to observe how a system behaves in states that are improbable or erroneous.

A form of controller can also be used to produce an event ordering based on *logical* or *simulated time.* A logical clock can be associated with each process. Messages sent by a process can be time stamped with the value of this clock at the time the message is sent [15]. A process may also arbitrarily increase its own clock to represent, for example, the passage of simulation time. The controller can then use these time stamps to select the event with the smallest time stamp from the set of pending events as the one that is to occur next. This mechanism can be used during the development process to simulate the operations of unimplemented components and the interaction of these components with processes that are currently being debugged and evaluated [16].

4.2.2 *State Re-creation.* An important requirement for debugging is repeatability or system state re-creation. The ability to faithfully repeat a distributed system's execution permits errors that only manifest themselves on selected execution paths to be isolated and identified. This facility is implemented by a console and a controller. The console records, in a transcript file, all events that occur in an application system and any commands issued by the user that can affect the system's execution. After a system has finished executing, the controller can use this transcript to guide the system's reexecution so that it re-creates the system's original execution. It does this by taking events from the transcript and comparing them against the actions in the system. At all times the controller knows what event must be executed next to ensure that this execution matches the original execution. The controller waits for this event to occur or waits until it knows that this event cannot occur. Re-creation continues until (1) a process does something different from what the transcription indicates; (2) the user enters a command which would change the system's execution; or (3) the user turns re-creation off. In all of these cases the system can subsequently be allowed to continue execution.

```
MESSAGE BUFFERS:
  ⟨acct_number⟩     ::= ⟨I: range 0 . . . 499⟩
  ⟨acct_type⟩       ::= ⟨S: "savings" | "checking" | "visa"⟩

  ⟨null⟩            ::= ⟨ ⟩
  ⟨error⟩           ::= ⟨S: "error"⟩
  ⟨success⟩         ::= ⟨S: "success"; I⟩

  ⟨connect⟩         ::= ⟨S: "connect"⟩
  ⟨security check⟩  ::= ⟨S: "security check"; acct_number⟩
  ⟨deposit⟩         ::= ⟨S: "deposit"; acct_number; acct_type; I⟩
  ⟨withdrawal⟩      ::= ⟨S: "withdrawal"; acct_number; acct_type; I⟩
  ⟨disconnect⟩      ::= ⟨S: "disconnect"; I⟩

  ⟨result_buf⟩      ::= ⟨null⟩ | ⟨success⟩ | ⟨error⟩
  ⟨request_buf⟩     ::= ⟨connect⟩ | ⟨security check⟩ | ⟨deposit⟩ | ⟨withdrawal⟩
  ⟨all_buf⟩         ::= ⟨request_buf⟩ | ⟨result_buf⟩
```

Fig. 13. Example message definitions.

A transcript can also be used to play back the execution of a distributed system on a set of consoles. Instead of receiving events from monitored processes, the channel takes events from a transcript file. The source of the events is transparent to the consoles. Playback is useful when repeating a system's execution is expensive. These transcript and playback facilities provide a form of dynamic documentation by allowing test runs and demonstration runs to be archived for later use.

### 4.3 Run-Time Protocol Checking

When the correct patterns of interprocess communication within an application system can be specified, this specification can be used by a monitoring system to recognize erroneous patterns. Many techniques have been defined for specifying communication protocols; a survey is given in [21]. We have implemented a console in Prolog called the *Protocol Checker* that accepts a specification of the allowable process interactions for a particular distributed system. The protocol checker receives events in the same manner as any other console but, instead of displaying the events, it checks them against the specification. If the event is permissible, then nothing happens; otherwise, the discrepancy is reported to the user. The user specifies Jipc level interactions to the protocol checker in three parts: (1) the types of messages used in the system, (2) the classes of processes in the system, and (3) the processes or process classes that interact and the types of messages they use for each type of interaction.

The first part describes the acceptable message formats that can be used in the system being monitored. A Backus-Naur form (BNF)-like grammar is used to describe the messages in terms of the primitive Jipc data types (integer, character, string, floating-point, process id, byte-block, and atom) and previously defined messages. Besides specifying the order and type of data items in a message, the value of a data item can also be restricted to a range of values or be one of a list of values. Several example message definitions for the banking system are shown in Figure 13. The second line defines the message type ⟨acct_number⟩ that consists of an integer (I) in the range 0 through 499. A ⟨withdrawal⟩

```
PROCESS_CLASSES:
  ⟨database⟩      ::= Calgary | Edmonton | Saskatoon | Vancouver
  ⟨bank_machine⟩  ::= calgary | edmonton | saskatoon | vancouver
  ⟨all⟩           ::= ⟨bank_machine⟩ | ⟨database⟩
```

Fig. 14. Definition of process classes.

```
INTERACTIONS:
  ⟨bank_machine⟩ send ⟨database⟩          ::= ⟨request_buf⟩
  ⟨bank_machine⟩ rec_reply ⟨database⟩     ::= ⟨result_buf⟩

  ⟨database⟩ receive ⟨bank_machine⟩       ::= ⟨request_buf⟩
  ⟨database⟩ reply ⟨bank_machine⟩         ::= ⟨result_buf⟩
  ⟨database⟩ send ⟨database⟩              ::= ⟨request_buf⟩
  ⟨database⟩ receive ⟨database⟩           ::= ⟨request_buf⟩
  ⟨database⟩ reply ⟨database⟩             ::= ⟨result_buf⟩
  ⟨database⟩ rec_reply ⟨database⟩         ::= ⟨result_buf⟩
```

Fig. 15. Process/message interactions specification.

message type is comprised of the string "withdrawal," the submessages ⟨acct_number⟩ and ⟨acct_type⟩, and an integer representing the size of the withdrawal. The second part of the system description associates specific processes with general process classes. In Figure 14 the process class ⟨database⟩ consists of the processes named Calgary, Edmonton, Saskatoon, and Vancouver.

The third part of the system description defines the permissible types of interactions between classes of processes and the types of messages they can use for each type of interaction. Each definition in this section consists of an interaction specification and the message(s) that can be used during that interaction. Each interaction specification is further made up of a subject process class followed by an interaction name and another process class. The interaction names are *send*, *receive*, *reply*, and *rec_reply*; the first three correspond to the Jipc interprocess communication primitives, and rec_reply refers to the types of messages a process can expect to receive as replies.

An example specification for the banking system is shown in Figures 13–15. The first and second lines in Figure 15 indicate that a process of class ⟨bank_machine⟩ can send messages of type ⟨request_buf⟩ to processes of class ⟨database⟩ and that it can expect to receive messages of ⟨result_buf⟩ as replies.

Some uses for the protocol checker and its associated input protocol specification are

(1) The specification provides a consistent notation for describing the communication interfaces among a set of application processes. Developers can use this mechanism to define interfaces that are checked at run time.

(2) While the system is executing, the protocol checker is able to detect some errors in the implementation of the protocol. Without the protocol checker, an illegal data item can be inadvertently passed among several processes

before the error causes one of them to fail. The protocol checker detects the error as close to its source as possible, thus minimizing the damage done by its propagation.

(3) As the system evolves and changes are made to the specification, the protocol checker ensures that modules conform to the updated specification.

### 4.4 Event Abstraction

The events of interest during system development change from IPC events to complex patterns of IPC events that correspond to high-level operations in an application. By recognizing a sequence of lower level events as a composite event, information that is not currently relevant can be eliminated. For example, producer, buffer, and consumer processes may form a simple component of a larger system. Whereas several events are required to transfer a message from the producer through the buffer to the consumer, only the higher level event of an item being consumed may be of interest to the user. The purpose of event abstraction is to support system debugging after development has moved beyond the IPC protocol level.

Just as an abstraction can be constructed in terms of primitive events, yet higher level abstractions can be defined in terms of events specified at the previous level. A composite event console could be developed that accepts a specification defining several levels of abstraction and reports events that occur at each level. The levels of abstraction and events at each level would form a tree with IPC events occupying the leaves. The composite event console is analogous to grouping in Mona; however, Mona groups processes, whereas a composite event console would group events.

As an application system executes, IPC events would be collected by the composite event console and, at any given time, several partially completed composite events could exist. The low-level events that comprise a composite event need not be contiguous in the stream of events. The composite event console must also be able to recognize events that can never be completed, for example, when processes required to complete a composite event have died.

Besides being able to report events in terms of abstractions, the composite event console must permit the user to switch among the levels at which the system is being monitored. When debugging, the user is continually observing the system, forming hypotheses, and testing these hypotheses. The ability to change the level of monitoring becomes relevant when the user is observing a system at one level, notices some event in the system that suggests a particular hypothesis, and wants to check it at a more finely focused level.

Event abstraction can be illustrated using the bank system. The design of this system can be decomposed into four distinct layers with different types of events at each level:

(1) primitive Jipc events.

(2) *remote procedure call* events that consist of Jipc send/receive/reply events.

(3) database *transaction* events that consist of either querying or updating the database. In either case, a transaction will be composed of a sequence of remote procedure calls.

1. vaxc.Calgary enters the system
2. vaxa.Vancouver enters the system
3. vaxb.Edmonton enters the system
4. vaxd.Saskatoon enters the system
5. vaxd.saskatoon enters the system
6. vaxd.saskatoon calls "connect" in vaxd.Saskatoon
   arguments ⇒ NONE
7. vaxd.saskatoon returns from "connect"
   results ⇒ NULL
8. vaxb.edmonton enters the system
9. vaxb.edmonton calls "connect" in vaxb.Edmonton
   arguments ⇒ NONE
10. vaxc.calgary enters the system
11. vaxc.calgary calls "connect" in vaxc.Calgary
    arguments ⇒ NONE
12. vaxd.saskatoon calls "security check" in vaxd.Saskatoon
    arguments ⇒ 200
13. vaxd.Saskatoon calls "security check" in vaxa.Vancouver
    arguments ⇒ 200
14. vaxd.Saskatoon returns from "security check"
    results ⇒ "success" 5
15. vaxd.saskatoon returns from "security check"
    results ⇒ "success" 5
16. vaxd.saskatoon calls "withdrawal" in vaxd.Saskatoon
    arguments ⇒ 200 "savings" 100
17. vaxd.Saskatoon calls "withdrawal" in vaxa. Vancouver
    arguments ⇒ 200 "savings" 100

Fig. 16. Trace of banking system at the transaction level.

(4) database *session* events that consist of a sequence of transactions beginning with the establishment of a connection to the database, followed by a series of query and update transactions, and finally, a disconnection transaction.

Figure 16 shows a trace of the banking system at the transaction level that corresponds to the events shown in Figure 6. Composite events could also be displayed graphically by using a version of Mona that accepts a specification of composite events and a description of how each event is to be animated. Just as Mona frees the user from having to deal with the system by reading a textual trace, the composite event console would free the user from having to deal with the system at the IPC level.

Bates and Wileden [3] describe previous work related to event abstraction. They propose behavioral abstraction as a tool for debugging distributed systems. Behavioral abstraction provides a means of transforming the stream of primitive events produced by a distributed system into a stream of composite events that correspond to the user's view of the system. Clustering and filtering are two essential mechanisms in behavioral abstraction. Clustering gathers together one or more primitive events into a single higher level event. Filtering removes primitive events from consideration as candidates in the formation of a higher level event. They also discuss some of the problems involved with recognizing

composite events out of a stream of low-level events, that is, filtering out noise, handling the relative timing of processes, sharing a primitive event between more than one composite event, and time and space considerations.

Baiardi et al. [1] present a debugger that is also based on levels of event abstraction. This debugger allows knowledge of the semantic model of the programming language to be included in event definition. Event abstraction is also discussed in [18].

## 5. DISCUSSION AND CONCLUSIONS

Our work on monitoring evolved from previous research that focused on the simulation and prototyping of computer systems [26]. The role of distributed systems in this work led to the development of Jade and its subsequent application to simulation and prototyping [16]. The display and animation of simulation traces [4, 10, 13] was a natural outgrowth of this work and resulted in the development of an initial version of the Jade monitoring system. This version was embedded within the implementation of Jipc.

As a result of the lessons learned from this simulation research and the initial Jade monitoring system, a new monitoring scheme was designed and implemented in 1984 and 1985. Two of the key objectives of the new implementation were to build the monitoring system on top of Jipc, to increase its modularity and portability, and to create a small set of basic tools that could be extended in different directions to facilitate experimentation with alternative monitoring schemes. This monitoring system was released as an integral part of the third release of Jade in the Fall of 1985 and forms the basis of most of this paper. During 1985 and 1986, a number of application systems, including a relatively large distributed simulation system [28] have been built using Jade and its monitoring system.

Our experience with the development and use of the monitoring system has led us to the following conclusions:

(1) *Collecting and distributing monitoring information with the same IPC mechanism used by application processes offers several advantages.* The design, debugging, and maintenance of the monitoring system is simplified. Also, when an application program is ported to a target network of machines, support for the monitoring system is available without having to port a second IPC mechanism.

(2) *The detection and collection of monitoring data should be separate from its analysis and display.* This separation supports the development of an integrated set of tools that share a common implementation and that can work together effectively (e.g., textual traces and deadlock detection). This approach also permits a wide range of monitoring tools to be implemented efficiently because the writer of each new tool does not have to become familiar with the low-level details of how monitoring information is gathered.

(3) *A wide variety of different monitoring views and interpretations is needed.* This is because no single tool can display all the information the user requires. The spectrum of tools required has at least two dimensions. First, it is necessary

to be able to move from low-level IPC views through increasingly higher levels of abstraction. Second, the use of complementary views, such as textual traces and graphical state displays, is very effective.

(4) *Textual and graphical displays offer roughly orthogonal views into a system's execution.* The former provides a complete record of a system's execution whereas the latter provides a relatively clear picture of the system's state. Our experience with the text console and Mona strongly suggests that the former is more useful for tracking down the cause of an error, whereas the latter provides more insight into the system's overall operation.

(5) *The ability to interactively control nondeterminism and to reproduce specific computation paths is crucial.* This permits better test coverage because improbable, but nevertheless possible, execution paths can be explicitly tested, and erroneous executions can be reproduced easily.

(6) *The ability to control nondeterminism supports system prototyping.* The use of a combination of real and simulated time to automatically determine the order in which independent events occur enables the execution of an application system to be coordinated with simulations of, as yet, unimplemented or unavailable components.

(7) *Animated, graphical state displays provide a very effective form of dynamic documentation.* In practice, Mona has been used most often to demonstrate a system to those who are unfamiliar with the system's structure and operation.

(8) *A monitoring system should be able to exploit semantic information about an application system.* Except when the user is monitoring the system at the IPC level, the monitoring tools must be able to interpret and display information in a way that reflects the structure and dynamic behavior of the system. This is not possible unless the monitoring system is able to recognize patterns of lower level interactions as logical operations in the application system.

Our work has focused on monitoring at the interprocess communication level. The processing and interpretation of large amounts of monitoring data, a problem common to the monitoring of both sequential and distributed systems, remains unresolved. For example, it is possible to collect a transcript of a system's execution that leads to an error and to use this transcript to recreate the erroneous execution. However, this is often not practical when the system must execute for a long period of time before the error occurs. Further work is also needed on the specification of higher level system activity in ways that can be exploited by both graphical and textual monitors.

## ACKNOWLEDGMENTS

We would like to thank the faculty, students, and staff associated with the Jade Project for contributing ideas and for providing an excellent environment in which to explore these ideas. Special recognition is due John Cleary and Radford Neal for their contributions to this work. We would also like to express our appreciation to the Natural Science and Engineering Research Council of Canada for supporting this research.

## REFERENCES


1. BAIARDI, F., DE FRANCESCO, N., MATTEOLI, E., STEFANINI, S., AND VAGLINI, G. Development of a debugger for a concurrent language. *Softw. Eng. Not. 8*, 4 (Aug. 1983), 98.
2. BALZER, R. M. EXDAMS—EXtendable debugging and monitoring system. In *Proceedings of AFIPS Spring Joint Computer Conference*. AFIPS Press, Reston, Va., 1969, 567–580.
3. BATES, P., AND WILEDEN, J. C. An approach to high-level debugging of distributed systems. *Softw. Eng. Not. 8*, 4 (Aug. 1983), 107.
4. BIRTWISTLE, G. M., WYVILL, B. L. M., LEVINSON, D., AND NEAL, R. Visualizing a simulation using animated pictures. In *Proceedings of SCS Conference on Simulation in Strongly Typed Languages* (San Diego, Calif., Feb. 2–4, 1984). Society for Computer Simulation, San Diego, 1984, 57–61.
5. BROWN, M. H., AND SEDGEWICK, R. Techniques for algorithm animation. *IEEE Softw. 2*, 1 (Jan. 1985), 28.
6. CHERITON, D. R., MALCOLM, M. A., MELEN, L. S., AND SAGER, G. R. Thoth: A portable real-time operating system. *Commun. ACM 22*, 2 (Feb. 1979), 105–115.
7. CHERITON, D. R., AND ZWAENEPOEL, W. Distributed process groups in the V Kernel. *ACM Trans. Comput. Syst. 3*, 2 (May 1985), 77–107.
8. CORSINI, P., AND PRETE, C. A. Multibug: Interactive debugging in distributed systems. *IEEE Micro 6*, 3 (June 1986).
9. DEWAR, A. A graphical debugger for prolog. Master's thesis, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada. (1985).
10. DEWAR, A., AND UNGER, B. Graphical tracing and debugging of simulations. In *Proceedings of SCS Conference on Simulation in Strongly Typed Languages* (San Diego, Calif., Feb. 2–4, 1984). Society for Computer Simulation, San Diego, 1984, 68–76.
11. GARCIA-MOLINA, H., GERMANO, F., AND KOHLER, W. H. Debugging a distributed computing system. *IEEE Trans. Softw. Eng. 10*, 2 (Mar. 1984), 210.
12. HARRISON, M. D. Monitoring a target network to support subsequent host simulation. Res. Rep., Dept. of Computer Science, University of York, Toronto, Ontario, Canada (1984).
13. JOYCE, J. J., BIRTWISTLE, G. M., AND WYVILL, B. L. M. ANDES—an environment for animated discrete event simulation. In *Proceedings of United Kingdom Simulation Conference* (Bath, U.K., May 1984). United Kingdom Simulation Council, 1984.
14. JOYCE, J. J., AND UNGER, B. W. Graphical monitoring of distributed systems. In *Proceedings of the SCS Conference on AI, Graphics, and Simulation* (San Diego, Calif., Jan. 1985). Society for Computer Simulation, San Diego, Calif., 1985, 85–92.
15. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
16. LOMOW, G. A., AND UNGER, B. W. Distributed software prototyping and simulation in Jade. *Can. J. Oper. Res. Inf. Process. 23*, 1 (Feb. 1985), 69–89.
17. MINCY, J., THARP, A., AND TAI, V. Visualizing algorithms and processes with the aid of a computer. In *14th SIGSCE Technical Symposium on Computer Science Education. SIGCSE Bull. 15*, 1 (1983), 106–111.
18. MODEL, M. L. Monitoring system behavior in a complex computational environment. Ph.D. dissertation, Department of Computer Science, Stanford University, Stanford, Calif. (1979) Also available from Xerox Palo Alto Research Center, Palo Alto, Calif.
19. MYERS, B. A. Incense: A system for displaying data structures. *Comput. Gr. 17*, 3 (July 1983).
20. NEAL, R., LOMOW, G. A., PETERSON, M., UNGER, B. W., AND WITTEN, I. H. Experience with an interprocess communication protocol in a distributed programming environment. In *Proceedings of CIPS Session '84 Conference* (Calgary, Alberta, Canada, May 9–11, 1984). Canadian Information Processing Society, Calgary, 1984, 361–364.
21. NOUNOU, N., AND YEMINI, Y. Development tools for communication protocols. Res. Rep. CUCS-160-85, Department of Computer Science, Columbia University, New York, N.Y. (Feb. 1985).
22. SCHWAN, K., AND MATTHEWS, J. Graphical views of parallel programs. *Softw. Eng. Notes 11*, 3 (July 1986), 51–64.
23. SNODGRASS, R. T. Monitoring distributed systems: A relational approach. Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (1982).

24. SOFTWARE RESEARCH AND DEVEVLOPMENT GROUP. Jade User's Manual (4 vols). Res. Rep., University of Calgary, Dept. of Computer Science, Calgary, Alberta, Canada. (Oct. 1985).
25. TEITELMAN, W., AND MASINTER, L. The interlisp programming environment. *IEEE Comput. 14*, 4 (Apr. 1981).
26. UNGER, B. W., AND BIDULOCK, D. S. The design and simulation of a multicomputer network message processor. *Comput. Networks 6*, 4 (Sept. 1982) 263–277.
27. UNGER, B. W., BIRTWISTLE, G. M., CLEARY, J. G., AND DEWAR, A. A distributed software prototyping and simulation environment: Jade. In *Proceedings of SCS Conference on Intelligent Simulation Environments* (San Diego, Calif., Jan. 23–25, 1986). Society for Computer Simulation, San Diego, 1986, 63–71.
28. UNGER, B., CLEARY, J., LOMOW, G., LI, X., SLIND, K., AND XIAO, Z. Jade virtual time implementation manual. Res. Rep. 86/242/16, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada (Oct. 1986).
29. VAUCHER, J. Future directions in simulation software. In *Proceedings of SCS Conference on Simulation in Strongly Typed Languages* (San Diego, Calif., Feb. 1984).
30. WYVILL, B. L. M., NEAL, R., LEVINSON, D., AND BRAMWELL, B. JAGGIES: A distributed hierarchical graphics system. In *Proceedings of CIPS Session '84 Conference* (Calgary, Alberta, May 9–11, 1984). Canadian Information Processing Society, Calgary, 1984, 214–217.

Received July 1985; revised November 1986; accepted November 1986