LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-392

MAM: A SEMI-AUTOMATIC DEBUGGING TOOL FOR DISTRIBUTED PROGRAMS

Lawrence Kenneth Kolodney

June 1987

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

MAM: A Semi-Automatic Debugging Tool for Distributed Programs

by

Lawrence Kenneth Kolodney

February, 1987

© Lawrence Kenneth Kolodney 1987

The author hereby grants to M.I.T. permission to reproduce and to distribute copies of this thesis document in whole or in part.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

MAM: A Semi-Automatic Debugging Tool for Distributed Programs

by

Lawrence Kenneth Kolodney

Submitted to the
Department of Electrical Engineering and Computer Science
on January 22, 1987 in partial fulfillment of the requirements
for the Degrees of Master of Science and Bachelor of Science

Abstract

Traditional debuggers, designed to examine single process serial programs, do not provide sufficient functionality for efficient debugging of distributed programs. There are a number of fundamental differences in the way in which a programmer understands the execution of a distributed programs, and a debugger must present data to its user in light of that fact.

MAM, A Message Abstraction Monitor, is described here. MAM provides a user with software tools needed to utilize a novel technique for debugging of distributed programs. MAM permits a user to define high level abstractions on a stream of messages transpiring between processes of a distributed program, using a Message Abstraction Language (MAL). MAM analyzes a post-mortem journal of such messages, attempting to impose user defined structures on them. The user may then view the analyzed journal in a sequential manner, with a graphical display indicating the relationships of various messages with respect to higher level abstractions and to processes of the distributed program..

MAM also provides "near-miss" detection allowing intelligent guesses to be made, for matches in an error-laden journal. This near-miss facility results in automatic detection of some programming errors.

The contributions of this research are a mechanism for the specification of correct abstract communications, the use of this in "near-miss" recognition, and the "play-back" nature of the presentation of this information for debugging purposes.

Thesis Supervisors: Dr. Karen R. Sollins
Title: Research Scientist, Lab. for Computer Science
Mr. R. Mark Chilenskas
Title: Research Staff, GenRad Inc.

Acknowledgments

I would like to thank my thesis advisor, Karen Sollins, for her patient, sympathetic and generally invaluable assistance that she has provided me over the past 17 months. Her editorial advice enforced needed intellectual discipline on my sometimes vague thoughts. Her optimism, in the face of my own self-doubt, provided me with faith to continue.

Richard Waters, who initially rescued me from the limbo of "advisorlessness," has my gratitude.

Franklyn Turbak, a gentleman and a scholar, kindly provided insightful criticism of my writing.

Mark Chilenskas, my supervisor at GenRad, was instrumental in helping me to develop the ideas in this thesis and in implementing them. His wealth of technical knowledge, his accessible manner, and his constant encouragement are all greatly appreciated.

To my parents, whose concern for and encouragement of my academic endeavors has been unflagging, I offer a heartfelt, if redundant, "Thanks!"

Finally, I would be remiss if I did not note the vital contribution of my numerous friends at Senior House dormitory, my on-campus home away from home, who, when times got trying, fed me, sheltered me, and provided me with the emotional support to continue work.

To my parents,

who taught me to give a damn.

Table of Contents

Chapter One: Introduction 9

1.1 General Motivation 9
1.2 Distributed Systems Improve Over Serial Ones 10
1.3 Debugging is Still an Art 10
1.4 A Message Abstraction Monitor 11
1.5 Previous Work 12
1.5.1 Event Description Languages 12
1.5.2 Runtime Stepping Control 13
1.5.3 Network Map 15
1.5.4 Movie Playback 15
1.6 Implementation Environment 15
1.6.1 Hardware Architecture 16
1.6.2 Software Architecture 16
1.6.2.1 The Message Server 16
1.6.2.2 Applications 17
1.7 Plan of Thesis 17
1.8 Definition of Terms and Abbreviations 18
1.8.1 Terms 18
1.8.2 Abbreviations 20

Chapter Two: Background: The Problem of Debugging 22

2.1 Learning from Experience with Serial Debuggers 22
2.2 Parallel Debugging Presents Qualitatively Different Problems 24
2.2.1 Time Ordering is Not Total 24
2.2.2 Too Much Confusing Data 26
2.2.2.1 The Problem: Complexity, and No Explicit Structures 26
2.2.2.2 A Solution: Behavioral Abstraction 29
2.2.3 Non-linear Data 30
2.2.4 Dealing With Unexpected Behavior 31
2.3 Automatic Error Detection 32
2.4 Summary 33

Chapter Three: User View of MAM 34

3.1 The Analyzer: Message Abstraction Language 35
3.1.1 Abstraction Descriptions 35

3.1.2 A Model for Understanding Data: MAL Semantics 41
3.1.3 MAL Limitations 48
3.2 JDM: Journal Display Monitor 49
3.2.1 Map 49
3.2.2 Control 50
3.2.3 Status 54
3.3 Examples 55
3.3.1 Example: A Fixture Test 55
3.3.2 Example: A Dead Process Causes Communication Breakdown 57
3.4 Summary 58

Chapter Four: Algorithmic Details 60

4.1 Analyzer Implementation 60
4.1.1 Overview of Analyzer Algorithm 61
4.1.2 Data Structures 63
4.1.2.1 Messages 63
4.1.2.2 Message Recognition Demons 66
4.1.2.3 Transaction Recognition Demons 68
4.1.2.4 Paths, Matching, and "Faulty Transactions" 71
4.2 Display Monitor Implementation 78
4.2.1 Unbounded Data/Finite Screen 79
4.2.2 Designing a Meaningful Display 80
4.2.2.1 What Should a Message Look Like? 80
4.2.2.2 How Should Messages Be Related? 81
4.2.3 Control in Terms of Abstractions 81
4.2.3.1 Keeping Track of Abstractions 82
4.2.3.2 Map Management 82
4.3 Summary 83

Chapter Five: Summary, Critique, and Future Work 86

5.1 Summary 86
5.2 Critique 90
5.2.1 Error Detection is a Powerful Tool 91
5.2.2 MAL Is Not Strong Enough, and Hard To Use 92
5.2.3 The JDM Lacks Sufficient Display Power 93
5.2.4 Control Mechanisms Are Not Ideal 94
5.3 Proposed Enhancements 94
5.3.1 MAL Language 94
5.3.1.1 A Syntax Checking Editor 94
5.3.1.2 A More Powerful Language 96
5.3.2 Monitor 97

5.3.2.1 An Enhanced Display 97
5.3.2.2 Control In Terms of Substructures 98
5.4 Conclusion 98

Appendix A: MAL Frames:A Hypothetical Transaction 99

Appendix B: MAL Frames: An Actual Scenario 105

Appendix C: A Message Journal 110

Appendix D: The JDM Display 111

Appendix E: A BNF Grammar for MAL 119

Table of Figures

Figure 3-1: A Sample Message Description 36
Figure 3-2: A Sample Transaction Description 39
Figure 3-3: A Journal to be Processed 43
Figure 3-4: A Message is Gobbled 44
Figure 3-5: A Transaction is Recognized 45
Figure 3-6: Journal is Processed 46
Figure 3-7: A Typical JDM Screen 52
Figure 3-8: JDM screen layout for Fig. 3-7 53
Figure 4-1: A Simple TRD, and MRD, Before Gobbling Message 64
Figure 4-2: A Simple TRD, and MRD, After Gobbling Messages 65

Chapter One

Introduction

1.1 General Motivation

Distributed, decentralized systems are rapidly becoming the architecture of choice for state-of-the-art computer projects. As the exponentially growing demand for computational throughput runs into the asymptotically limited power of computer hardware, distribution of computational load provides an increasingly attractive alternative to the striving for faster cycle times. Additionally, many applications today, such as array processing and graphics, naturally lend themselves to a distributed approach.

For these reasons, much work has been done recently on perfecting more sophisticated and elaborate distributed computer systems. (See for example [CCA80], [Hillis81], [Arvind80]) Unfortunately, while much progress has been made in augmenting the computational power available to distributed programmers, precious little has been done to insure that the ability to *control* that power has increased at a similar pace.

Specifically, there is a need for intelligent debugging and monitoring tools for distributed systems, such as have long been standard equipment for writers of serial programs. This thesis describes an investigation into one approach to that problem, and speculation as to future approaches.

1.2 Distributed Systems Improve Over Serial Ones

The term "distributed systems" encompasses a large class of computer system architectures, from tightly coupled highly parallel systems such as the Connection Machine [Hillis81], to networks with highly autonomous nodes, such as the ARPAnet [Cerf83, DARPA81]. All of them have in common the characteristic that multiple processes are active simultaneously, and are working in coordination with each other to achieve some global functionality.

Distributed systems provide potentially great increases in the throughput of computer systems, without drastic increases in processor speeds. They do this at a cost. By increasing the amount of activity in the system at a given moment, the complexity of these systems increases rapidly. Not only is there a linear increase in the amount of program data to be kept track of, but there is the newly added factor of *interconnections*. The number of potential interconnections increases quadratically as the number of nodes. Managing this complexity and reducing it to a manageable level presents a challenge for designers of debuggers for distributed systems.

1.3 Debugging is Still an Art

In many ways debugging is the step-child of computer science, a necessary evil.[1] Very little theoretical work has been done on it. While debuggers today are quite sophisticated by comparison to their predecessors of twenty years ago, there still exists today no detailed theory of debugging. This is perhaps understandable since, until recently, debuggers have done adequately without such a theory. In the absence of a good theory, serial debuggers have been designed by intuition.

---

[1] As Henry Leiberman has noted: Debugging is like sex, everybody does it, but nobody wants to talk about it.

The serial debuggers which have been written in the past had a straightforward task to perform: to provide a user with realtime control over the sequential operation of a process, and to give the user information about the state of the program. Debuggers today use the same basic techniques as those of the nineteen sixties, the main difference being the format of the display of information.

The execution of a distributed program involves types of data and interactions which are not handled well by serial debuggers. Examples of novel, poorly handled features include interprocess messages, simultaneous processes, and implicit higher level transactions. Creating a debugging system for distributed programs requires fundamental changes in the previous debugging paradigms. In order to do this, it is first necessary to state, in abstract terms, exactly what debuggers do. It is necessary to look at traditional serial debuggers, discover general principles of debugging, and apply them to the distributed problem. This will be examined further in Chapter 2.

1.4 A Message Abstraction Monitor

This thesis describes a Message Abstraction Monitor (MAM) which was designed with the above issues in mind. The MAM system was designed specifically for a distributed system under development at GenRad Inc. as the operating system for automatic test equipment (ATE).

MAM is a post-mortem analyzer which allows a user to inspect the message journal of a distributed program execution in terms of high level abstractions previously defined. This system uses the paradigm of *behavioral abstraction* [Bates81], understanding program flow in terms of the message passing behavior of individual processes, rather than program steps.

MAM consists of two modules, an Analyzer, written in Scheme [Abelson85], a

dialect of lisp, and a display monitor, written in C and utilizing the Sun workstation graphics system. The analyzer "digests" a raw journal, by parsing it in terms of abstractions provided by the user. The display unit uses graphics to provide a movie-like playback of the digested journal.

MAM was designed as a prototype system. It does not provide the clean user interface that would be required in a production debugging system. Rather it was used to explore exactly what the needs of such a system might be, in terms of analytical power and graphical display.

1.5 Previous Work

1.5.1 Event Description Languages

Various projects have used languages to describe interprocess communication on a higher level than the single message. All have relatively simple language descriptions, essentially extensions of regular expression descriptions.

The Event Definition Language (EDL) [Bates81, Bates82, Bates82a, Bates83, Bates86] was designed as part of a project to investigate techniques for programming on distributed systems. EDL provides a means of specifying, by means of regular expressions (with some extensions), a hierarchy of abstract event types on a space of primitive interprocess events. The authors introduce the notion of "Behavioral Abstraction", as an alternative to state based debugging. Their idea is to think of modules in a distributed system in terms of observable behavior (interprocess interaction), so that the state of the machine is defined in terms of this behavioral information rather than information about the program counter and variable bindings. In addition to providing descriptions of events, EDL allows the specifying of predicates on events so that filtering of uninteresting data can occur. EDL is intended as the

basis for a full debugger for distributed programs. Such a system has been built and is the subject of a thesis by Bates: [Bates86].

The MuTeam Debugger [Baiardi83] implements another language for describing events, this time in a distributed programming language. The debugging language is an extension of the programming language itself, and the authors give a rigorous analysis of the resulting semantics of this system.

Gertner [Gertner80] uses finite state machine descriptions to recognize interprocess events. This work allows hierarchical descriptions by allowing lower-level FSAs to be included as part of higher level descriptions. This system is primarily concerned with monitoring network behavior for performance analysis, rather than with debugging.

1.5.2 Runtime Stepping Control

A number of debuggers have been implemented which actually give the user breakpoint and stepping control of a distributed system at runtime.

Smith [Smith81] implements an interprocess debugger for processes communicating within a single processor. The debugging mechanism is an integral part of the operating kernel of the message system. Since messages are not being passed over a network, the kernel has complete control over the flow of messages. The system takes advantage of this fact by allowing a much finer grain of control than in other systems. Rather than have the transmissions of a message be the atomic type of event, this system considers the crossing of certain conceptual boundaries to be items of interest. For example, processes are modeled as having some number of *ports*, queues for receiving messages, which are internal to the process. Messages are sent to ports, rather than to processes. A message may be "inside" of a process, but still outside any particular port. The crossing of a message from the "ether" into a process is considered an event

in this system. Crossing from that process into a particular port is considered still another.

Smith's system allows the user to define demon recognizers, which monitor the message traffic, and fire a set of commands when they recognize certain behavior. These commands may set a new breakpoint, or insert "bogus" messages into the stream. The demon facility allows the user to run the system at normal speed, while still allowing control over execution.

[Schiffenbauer81] addressed the problem of global breakpoints in a physically distributed system. Smith's approach would not work in such a system because of the stochastic nature of network traffic. Setting a breakpoint at an arbitrary time might cause messages to be lost, or received in an unexpected order. Schiffenbauer used Lamport's [Lamport78] notion of logical clocks as a method for insuring transparent message delivery, even in the presence of breakpoints. Logical clocks insure that messages reach a program only at the same logical time (i.e. relative to other messages) in its execution history as they would have in a freely running system.

[Garcia-Molina84] describes a system by which local logs of process activity are kept, and then examined and coordinated later. A log is kept of the "interesting" activities of each process. Some of these activities are recorded by the system itself (such as process birth/death), while others are the responsibility of the process itself to record. Thus, this system allows arbitrary data to be stored in the process log, and avoids the problem of recognizing higher order events by having the process write them out explicitly. This system also uses logical clocks to coordinate the transaction logs.

After all the logs are recorded, they are treated as a distributed relational database, and a user may make queries into it. However no facility is given for reproducing program behavior from the records.

1.5.3 Network Map

[CCA80,pp. 113-122] describes a graphical system for monitoring network traffic in a distributed database system. It contains general ideas for the graphical representation of messages, including a network map format, use of color and arrows to indicate data flow.

1.5.4 Movie Playback

The idea for a movie playback of the message journal was inspired by [Balzer69], which describes a system for movie-like playback of a single process program, but without the use of graphics. The major idea in this system is to highlight program lines on the screen as they were being executed, while continually updating a display of program variables.

1.6 Implementation Environment

This section describes the specific debugging task for which MAM was created at Genrad. Although MAM was designed as a general purpose debugger, there are certain assumptions made about the nature of the types of programs that might need to be debugged, based on the design of the Genrad environment.

MAM was developed in response to a need at Genrad to understand the behavior of the control software for a new product, the 2750 Automatic Electronic Tester. The 2750 is controlled by three microprocessors connected via an ethernet, and running multiple concurrent processes. Each process is responsible for a specific well defined task, such as user interface, automatic test generation, or run time control of the test hardware.

1.6.1 Hardware Architecture

Processes in the Genrad system are distributed between a Digital Equipment Micro-Vax and a Sun Microsystems workstation based on a Motorola 68000 family processor. Systems run the UNIX operating system. All messages are transferred via ethernet hardware. No distinction is made between intra-processor communication, and process communication between physically distinct processors.

The Sun workstation acts primarily as a user interface. Its capabilities include an advanced windowing system, and high resolution bitmapped graphics. The Vax acts as the central processor of the system, handling processing-intensive jobs, and interfacing with the specialized hardware of the tester.

1.6.2 Software Architecture

1.6.2.1 The Message Server

All message traffic in the system is coordinated by a central message server. This process handles queuing of messages and spawning of new processes. Additionally, it generates a journal of all of its relevant activities. Specifically, this includes message transfer and process forking. Because a request for forking of processes is a behavior of equal importance to message passing from a behavioral abstraction viewpoint, process forks are stored in the journal in a way which makes them indistinguishable from message transfers. In this way, a process A requesting a fork to create process B is represented as a "pseudo-message" of type "fork" from process A to process B.

The journal output of the message server is stored in a frame-based database system known as The Navigator. Each frame in the journal corresponds to a single message transfer, and contains slots for relevant descriptive information of that transfer. These frames are described in more detail in Chapter 3.

1.6.2.2 Applications

A number of specialized modules make up the operating system in this tester. They include:

- A user interface task (UIT) for screen management.
- A number of user interface nodes (UIN) for individual window management.
- A run time system for the tester (RTS) to interface between the software and the specialized runtime hardware.
- A run time executive (RTE) to control the runtime hardware.
- An automatic test generation program (ATG), for generating test instructions, given descriptions of circuit boards.
- A diagnostic system (DIAG) for analyzing results of board tests.
- A test set development coordinator (TSD), a global program coordination module.

The interactions between these various modules vary greatly. The user interface uses asynchronous messages to other tasks at varying intervals. The run time system and executive are in close communication, with loads of up to 3 messages per second expected. In contrast, the automatic test generator may send one message in an hour.

1.7 Plan of Thesis

The rest of this thesis describes the motivation behind MAM, its functionality, and the issues that arose during implementation. Some realistic debugging case studies are also presented.

Chapter 2 discusses general issues in debugger design and how using an event

based view requires a modification of the traditional debugging paradigm. It introduces the notions of program *object data* and program *control data*, and highlights the overlap between the two in the behavioral abstraction paradigm.

Chapter 3 presents a user view of MAM, essentially a user's manual. It defines MAL, the Message Abstraction Language for describing behavioral abstractions, and describes the Journal Display Monitor, the graphical display interface. Examples of debugging practice are given with respect to some case studies.

Chapter 4 provides implementation details of the MAM systems, including the algorithms used by *demon recognizers* to analyze the input journal.

Chapter 5 critiques the current MAM implementation, and proposes future extensions. It discusses the results of the research conducted, particularly how the MAM performed with respect to the original expectations. A number of extensions are proposed, including an input editor for MAL, improved language features and improved display features.

1.8 Definition of Terms and Abbreviations

There are a number of terms and abbreviations used frequently throughout this thesis whose meaning may be ambiguous. They are defined below:

1.8.1 Terms

Control Data Information describing the state of a program's computation which is not generally involved in the computation itself. Typical examples of control data are the current program line number and pending procedure calls on a stack. *See also* **Object Data.**

Debugger A software tool for debugging computer programs. Not to be confused with the person doing the debugging, the user.

Distributed System A computer system in which two or more computer processes working in unison to provide some coherent service. Distributed systems take many forms, from very loosely coupled ones, such as the ARPAnet, in which nodes are relatively autonomous, to very tightly coupled ones, such as the Connection Machine in which each node is intimately involved in the workings of the other. The distributed programs of interest in this work are those which fall somewhere between those two extremes. In particular, it is concerned with systems which utilize a local area network (LAN) and a central message server, and involve autonomous processes working to provide a single service to a user.

Element A message or transaction which is part of a larger transaction.

Event A message or a transaction.

Event Notification A message sent from one daemon in the MAL analyzer to another, indicating that an event of a certain type has been recognized.

Faulty Transaction A sequence of messages which deviates from the description of some previously described transaction by some permitted margin of error. It is assumed that a faulty transaction is a failed attempt at completing a valid transaction.

Message A data structure which is transferred from one process to another in a distributed system, usually over a network. Messages generally have as properties an id, a sender, one or more receivers, a timestamp, and contents.

Message ID A number associated with a message class *as understood by processes within a distributed program*. This information indicates how the recipient of a message should interpret the data in that message.

Message Name An unique identifier assigned to an *instance* of a message type that has been recognized during a journal analysis. Such names are constructed by concatenating the message type with the timestamp of the message instance.

Message Type A class of messages *defined for debugging purposes.* A message type is described by an MD. Characteristics which define a message type may include the message id, the sender and the recipients of a message.

Object Data Information on which a computer program is explicitly operating. Typical examples of these include program variables and data structures. *See also* **Control Data.**

Serial Debugger A program designed to aid in the debugging of serial programs. Serial debuggers typically provide the user with runtime control over a program, allowing the use of *tracing, breakpoints,* and *data dumping.*

Transaction The sending of one or more messages between two or more processes to achieve some particular purpose. Transactions can be classified into types using a Message Abstraction Language.

1.8.2 Abbreviations

AD Abstraction Description. The basic unit of a MAL input file. Either a MD or a TD.

ATG Automatic Test Generation. One of the program process types in the GenRad system.

DIAG Diagnostics. One of the program process types in the GenRad system.

JDM Journal Display Monitor. The MAM module which displays processed journal data on a graphics screen.

MAL Message Abstraction Language. The language for describing transaction and message types.

MAM Message Abstraction Monitor. The program described in this thesis.

MD Message Description. A MAL construct describing an abstract message type.

MRD Message Recognition Demon. A SCHEME message passing object which handles the recognition of an instance of a message type described in an MD.

NFA Non-deterministic Finite State Automaton. A model of computation useful in pattern matching.

RTE Runtime Executive. One of the program process types in the GenRad system.

RTS Runtime System. One of the program process types in the GenRad system.

TD Transaction Description. A MAL construct describing an abstract transaction type.

TSD Test Set Development. One of the program process types in the GenRad system.

TRD Transaction Recognition Demon. A SCHEME message passing object which handles the recognition of an instance of a transaction type described in a TD.

UIN User Interface Node. One of the program process types in the GenRad system.

UIT User Interface Task. One of the program process types in the GenRad system.

Chapter Two

Background: The Problem of Debugging

This chapter describes the theoretical issues which must be faced by the designer of a debugger for distributed programs, and gives brief descriptions of how MAM addresses them.

2.1 Learning from Experience with Serial Debuggers

Before setting about the task of designing a debugging tool, it is useful to ask what the process of debugging is all about. Programmers have quite a bit of experience debugging serial programs, as well as an intuitive understanding of that process. With distributed programs, however, there are a number of issues which make for a qualitatively different (and harder) problem. On a high enough level, though, the goals and methods in both domains are essentially the same. This section examines the general methods of debuggers for serial programs (serial debuggers).

The primary goal of a debugging session is to determine if the program is "doing the right thing", and if it is not, to discover the internal mechanism that is failing to operate in the desired manner. The test for "doing the right thing" is frequently just an informal comparison, by the user, of expected input/output behavior with that which is observed. Once an anomalous behavior pattern is noted, the programmer makes an hypothesis (possibly a very vague one), as to the internal cause of this problem, and attempts to make the necessary program modifications.

In this scenario, the debugger acts as a passive tool in the hands of the user. The debugger provides *access* to the program, but little or no *interpretation* of the results of running it.[2] Three features of a debugger allow the user to access the internal world of the program to check and further refine his hypothesis:

•*Data Dump/Display/Alteration*: A program might be executing the right statements, but those statements might not be performing the correct action on data, (i.e. some procedure is being used in the wrong way because of a misunderstanding by the programmer). Monitoring real-time changes in variable state, or analyzing a post-mortem dump of variable values can help spot the defective program line(s).

•*Stepping/Breakpoints*: It is frequently useful to run through a small section of code, and then to examine the partial results of the computation. A breakpoint feature allows the user to put a marker on a particular instruction, causing an interrupt of the program whenever that instruction is reached. A stepper can be thought of as a degenerate case of the breakpoint, where a marker is placed on every instruction.

•*Program Line Execution Trace*: A breakpoint leaves the user with partial computation data, but does not indicate that program control path which led to it. Sometimes this information can be inferred from the resulting partial computation, but often it cannot. Since a program may execute millions of instructions in a single run, it is often impractical to go through large portions of it at single-stepping speed in order to follow the exact thread of control. Tracing is a next-best attempt to infer the actual thread of control. Key instructions of the program can be marked so that their execution is flagged.

---

[2]Perhaps the word *debugger* is an inappropriate name for this type of program, since it is the human user that actually does the debugging.

The first of these features gives access to program *object data* , while the second and third give access to program *control data* information.[3] A typical debugging session involves first using the control functionality of the debugger to maneuver to a certain logical point in the program, and then observe program data. This cycle may be iterated a number of times at different levels of detail until the fault is found.

2.2 Parallel Debugging Presents Qualitatively Different Problems

2.2.1 Time Ordering is Not Total

In a distributed system, using the serial debugging paradigm is not always feasible. The program control operations (stepping, breakpoint and trace) all depend on a total ordering of program statement execution times. The idea of a step or breakpoint is meaningful only if there is a unambiguous successor to any given program step. Steppers and tracers are useful insofar as they give the user an idea of the logical sequence of instructions executed.

In a distributed message passing system, there may be no total ordering of program statements, only a partial ordering. In the logic of distributed programs, it is sometimes impossible to predict, and irrelevant to know, the relative ordering of two program statements in separate processes [Lamport, 1978]. For example, if two processes, running on physically separate systems, execute program statements at times very close to each other, it is impossible to determine remotely which occurred first. The reason is that the time for a signal

[3] *Object data* refers to the data that the program is computing on (e.g. variable values, data structures), while *control data* refers to the "hidden" information which is maintained by the computer to actually run the program (e.g. subroutine returns, program counter, procedure call stacks)

indicating such a statement execution to reach a monitoring process is a random variable, affected by the statistical characteristics of the message transmission medium. However, as Lamport has shown, this relative ordering is important only among the class of program statements whose relative ordering can be detected in (i.e. has some meaningful effect on) the resultant functional behavior of the program. Fortunately, this class of instructions is limited to those program statements which are involved in interprocess communication.

The solution, then, to the problem of controlling a non-total ordering of program steps, is to raise the view of the debugger to a level at which only interprocess communication actions are visible. Schiffenbauer [Schiffenbauer81] has offered one possible solution to this problem by recognizing that at a high enough level of abstraction, processes can be seen as interacting in a coordinated fashion. By thinking of individual processes as black-boxes, and treating them solely in terms of their message-passing behavior, Schiffenbauer has created a system where single stepping a distributed program, in terms of a logical clock determined by message dependencies, is possible.

Schiffenbauer's system is complex and not general. Because it operates as a realtime debugger, it must be concerned with debugger *transparency*, the maintenance of the illusion that the debugger does not exist. Nothing that the debugger does should have any effect on the logical behavior of the program. However, if a debugging program is merely one process among many competing for the resources of the distributed system that it is debugging, it cannot help but have some effect on the runtime environment of the other processes, and thus on their behavior. This problem can only be overcome, as Schiffenbauer demonstrates, by making the debugger an integral part of the message passing system. All messages pass through and are routed by the debugger. Parts of the debugger reside on every processor in the system to maintain control over individual processes under control of the central debugger.

MAM avoids the issues of timing and transparency by examining the distributed process in a *post-mortem* fashion. MAM does no observation of actual messages on the network; rather, it examines the journal produced by a central message server. This journaling is performed as a side effect of the interprocess communication facility, and thus entails no additional expense. The problem of transparency is no longer an issue, since MAM does not have any affect on the program at runtime. Since MAM is a general pattern recognizer, it can easily be customized for a variety of distributed environments; it requires no fundamental modifications to the system it is examining.

MAM addresses the problem of ordering somewhat differently than does the Schiffenbauer system. Schiffenbauer avoids the need to "arbitrate" the timing of program steps in separate processes by abstracting processes and only looking at their external behavior. But it is still necessary to arbitrate the external timings, and this is done by logical clocks. With MAM the arbitration issue does not occur, since a *de facto* ordering is automatically imposed on interprocess communication by the journaling mechanism. Since this mechanism is an integral part of the system being debugged, there is no loss of transparency as a result.

2.2.2 Too Much Confusing Data

2.2.2.1 The Problem: Complexity, and No Explicit Structures

An important goal of serial debuggers is to limit the amount of program information presented to the user, and to present it in a manner meaningful to the user, in terms of the abstractions in the user's model of program behavior[4] .

[4]For example, a LISP debugger that traced programs in terms of machine language instructions would be providing complete information, but it would not fit the programmer's model, which is that of a LISP interpreter environment.

This problem of controlling information overload is compounded in a distributed environment. The amount of relevant data is multiplied by the number of processes, and the program state information is now much more complex than a simple stack of procedure calls and program counter number.

Data overload can come in two forms: too much *object data*, which makes *data dumping* difficult, and too much *control data*, which makes tracing or breakpointing harder. In a debugger such as MAM, which adopts Schiffenbauer's notion of process-as-black-box, the problems of tracing and of data dumping are, however, essentially identical. The reason for this is that, if the finest grain of program steps are thought of as being the time between messages passing between processes, (these messages being the data of the program), the task of tracing a program is reduced to displaying the data being transferred between processes at a high enough level of abstraction.[5] It can be seen then that the MAM solution to the problem of partially ordered instructions, using message transmissions as instruction boundaries, also helps to limit to some extent the amount of data that it must process. All program behavior which is not discernible from I/O behavior is abstracted away, so that the only important information about the program is the list of messages which are sent between its component parts. However, if our goal is to display message traffic for a lengthy program run, distributed debuggers (debuggers for distributed programs) will run into the same problem that serial debuggers face, namely a glut of data, and no means to understand it.

---

[5]The distinction between control and data is somewhat fuzzy. Usually it is possible to tell where in the logic of a program a process is by observing certain landmarks. In most cases, these landmarks are single program step executions. In this case, since program steps are being abstracted away steps, the lowest level of landmark we have is the interprocess message. Seeing a message indicates where in the logic of the program a computation is. The lack of fine-grained observation causes all of the program behavior between messages to be treated as an atomic action.

Simply printing out identifiers for all messages that were passed during a program run would give the user a behavioral description of the program execution, but the sheer amount of uninterpreted data would make further analysis difficult. It would be analogous in a serial system to printing out every machine language instruction executed. In serial debuggers, this problem is avoided by giving the debugger knowledge of the higher level abstractions used in the programming language. Often, special object code is generated which tells the debugger of the relationship between machine language instructions and lines of source code. This allows the trace facility of a debugger to represent the execution of hundreds of lines of machine code by a single source program statement. Most debuggers today go one step further, allowing the user to treat the execution of user-defined procedure calls as atomic actions, allowing control in terms of high level abstractions.

Unfortunately, the high level abstractions in a message passing system are often not explicit in the code of the program. For example, a very common transaction in any system is a request for data. Process A sends a message to Process B, requesting a certain piece of data, Process B sends that data to Process A and (optionally) awaits an acknowledgment. This sequence of two (or three) messages represents a logical, functional unit. The programmers of the system expect it to happen at certain times, and its absence (or malfunction) would indicate a program bug. But there is not enough intrinsic information in these three messages to indicate that they are necessarily related in a logical way. If there are many processes using the system, it is unlikely that these messages will occur serially without interruption. While it is true that they will all have the same sender and return addresses, these do not necessarily give enough information to distinguish between adjacent or intertwined transactions.

2.2.2.2 A Solution: Behavioral Abstraction

Peter Bates has developed an approach to distributed debugging called *Behavioral Abstraction*, which addresses some of the issues mentioned above. Behavioral abstraction involves viewing a program solely in terms of the message passing behavior of its component processes, specifically inter-process communications. These interprocess interactions, usually in the form of messages, can be used as the basis for building up abstract transaction types.

A standard set of transactions will be associated with many distributed computing environments. A transaction is the message passing analogue of a procedure. That is to say, a transaction is a set of primitive actions that can be thought of, functionally, as a single unit, corresponding, on some level, to a single service or action. When a programmer is analyzing the behavior of a complicated message passing system, it will be these high-level transactions which will form the basis of understanding. It is only natural then, that in a debugging system, transactions should form the evidence of program control flow.

Since transactions are not explicit, in that they depend largely on the users own model of program behavior, a language for describing and identifying them is needed. Regular expressions are a natural means of description for transactions, as they describe classes of strings which do not require recursive descriptions (self-calls). Many distributed systems lend themselves to this form of description. In particular, those systems which utilize a synchronous paradigm in which processes either block for acknowledgments or ignore them, thus eliminating the possibility of a backlog of messages, can be described in such a fashion.

There is also a need in transaction descriptions for hierarchical descriptions. On the highest level, transactions may contain hundreds of individual messages. Once a user has isolated a problem within one of these top level transactions, a finer grain of viewing may be required. Thus, the user will need access to lower

level (but not necessarily primitive) events which make up the transaction. Eventually, it is likely that a single message (or lack thereof) will be found to be the culprit; finding that message by way of iteratively increasing the amount of detail is a natural way to go about that task.

In order to present message traffic in a useful way, MAM uses high level abstractions (transactions) to encapsulate detail. These abstractions are specified using expressions in Message Abstraction Language (MAL). Judicious use of this Message Abstraction Language (MAL), will allow the user to be shown only that data which is of immediate interest, and only in an informative format.

2.2.3 Non-linear Data

Most serial debuggers simply print status messages on a terminal in some straightforward way. The very nature of a serial program makes the use of a linear stream of messages quite a natural representation of program dynamics. With a distributed debugger it is essential to be able to represent a situation in which many different things are happening at once. Certainly, there are schemes which could do this by printing out messages on a terminal, but experience and current understanding of human cognition[6] would indicate that a graphical display of such data would be more useful.

The MAM display format is that of a network map. The various processes of the distributed program that are being monitored are displayed as nodes on a network, with connecting lines indicating the flow of various messages. Messages

---

[6] [Model79], p. 12:"...human physiology and psychology ... reveal a strong visual bias in the human organism ... sensory information is highly organized 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 brain is that the pictorial, or analogical, presentation of information is often more effective than presentation in the more abstract, symbolic modes."

are displayed in the context of the higher level transaction of which they are a part.

2.2.4 Dealing With Unexpected Behavior

As mentioned previously, using behavioral abstraction has the effect of making the *object data* of the debugger view the same as the *control data.* The user depends on the values of object data to determine the state of the control mechanism of the program. This has the unfortunate effect of tying the ability to understand the control mechanisms of the program to the ability to recognize valid data. The monitor can only describe what is happening within the program insofar as it can recognize the sequence of events that it observes.

Since MAM is designed to be used as a debugging aid, it is to be expected that some of the data to be analyzed by it will be faulty. High level transactions may be incomplete, or have extraneous messages in them. Messages may be sent which have unrecognizable types. Coordination problems may cause an incorrect ordering of messages. However, only in a pathological case will what appears be totally uninterpretable.

MAM has a language for describing recognizable events. It is expected that there will only be perhaps a few dozen such events of interest, and maintaining a library of them would be simple. However, it is also necessary for MAM to recognize the aforementioned "faulty" transactions, which are frequently caused by the very bug for which the user is searching. Yet to create a library of "bad" transactions would seem a Herculean task. Given the variety of ways in which high level transactions might be "corrupted" it would seem to indicate orders of magnitude difference in the amount of information needed.

The "faulty transaction problem" is solved by making certain assumptions about the appearance of faulty transactions. One can think of transactions (essentially

strings of messages) as having locations in a metric space. It can be expected that in the space of strings of messages, those points indicating valid transactions are sufficiently distant that one could (conceptually) draw large error circles around those points without overlap. Simple algorithms can be used to generate points in those "error margins", solely from data about valid points or, better, to determine whether an unrecognized pattern fits within the valid scope of one of the error margins of an *bona fide* transaction.

In more concrete terms, there are a number of ways to make the message specifications more "fuzzy", such as permitting transaction recognition in the presence of some limited number of unrecognizable messages, or of missing ones. MAM currently recognizes transactions that are faulty in one subcomponent of the defining pattern of the transaction. This missing component may consist of single missing message, or, in the case of a transaction built up of smaller transactions, a large number of messages. Since messages within a functional transaction tend to be causally related to each other, it seems much more likely that multi-message faults would occur within such units, rather than across them, thus making the single sub-transaction tolerance of MAM likely to catch common errors.

2.3 Automatic Error Detection

The functionality in MAM for *recognition* of faulty transactions permits, as an obvious side effect, the automatic *detection* of program errors by the system. This provides a significant service that is not available in serial debuggers. In serial debugging systems, the only errors which are explicitly flagged by the system are those which cause runtime errors, typically involving bad data types. Most types of semantic errors, or errors in logic, are left to the user to detect.

In a journal processed by MAM, any transaction which seems to be faulty will be

flagged. This immediately indicates for the user the general location of the problem. By then observing the situation on progressively lower levels of abstraction, the user can eventually pinpoint the exact cause of the problem.

2.4 Summary

In this chapter the theoretical and practical motivation for a new approach to distributed debugging, have been laid out. The process of debugging is first characterized in a general way, taking experience with serial debugging as a basis. The roles of *breakpoints*, *tracing*, and *data dumping* are examined.

From this starting point the problems encountered in distributed debugging are examined, and the areas in which the serial debugging paradigm breaks down are discussed. The major problems discussed include: maintaining debugger transparency in a distributed environment, displaying complex data in a manageable and meaningful way, and (given a "black box" solution to the to previous problems) how to handle unexpected program behavior gracefully.

Two major innovations were introduced to address these problems. *Behavioral Abstraction*, the method of understanding program behavior only in terms of interprocess communication, serves both to avoid transparency and timing issues, and to keep the amount of data down to a manageable size. A *network map* further reduces the problem of data glut by presenting the results of behavioral abstraction in an intuitive and simple manner.

In addition, a third innovation is introduced as a necessary side effect of the prior two. This is *automatic error detection*, which allows the debugging program to detect semantic errors and errors in logic, which were only manually detectable in serial systems.

Chapter Three

User View of MAM

This chapter describes the "user view" of MAM. "User view" is a more general term than "user interface", encompassing not only the operational details of that interface, but also the underlying model which the user must possess in order to utilize MAM properly. This includes the semantics of the Message Abstraction Language, as well as the "graphical semantics" of the network map display.

Using MAM is a two step-process, involving first the *analysis* of a message journal, followed by the subsequent *interactive display* of the result of that analysis. Two separate programs were written to accomplish these discrete tasks.

The Analyzer, which was written in Scheme, a dialect of Lisp, takes two inputs: a message journal, in the form of a Navigator[7] library file, and a Message Abstraction Language (MAL) input file, describing message abstractions. It operates in a non-interactive fashion, and outputs a modified library file, containing the original journal, plus information about the abstractions it has detected there. The Analyzer scans the journal for instances of transactions that have been defined in the MAL input file, marking their elements in the journal. Additionally, faulty transactions, those which come close enough to matching specifications to be considered "near-misses", are also marked.

Once the analysis is done, the user may actually begin to use the journal data to debug the program at hand. This is done using the **Journal Display Monitor**

[7]The Navigator is a frame representation language data base system system available at GenRad.

(JDM), an interactive Sungraphics[8] based system, written in the C programming language. The JDM permits the user to display sequentially the behavior represented in the journal, at different speeds, at different grains, and at a number of different levels of abstraction.[9] The rest of this chapter gives a detailed description of the use of MAM.

3.1 The Analyzer: Message Abstraction Language

A standardized user interface for the Analyzer was never developed. It was assumed that syntactically and semantically correct MAL expressions would be prepared offline in a form readable by the program. This "no frills" approach to interpreting MAL expressions made implementations easier, but, not unexpectedly, caused many problems in actual debugging sessions. A proposal for a more intelligent interface is discussed in Chapter 5. The aspect of the analyzer of greatest interest, then, is MAL itself. The rest of this section describes MAL, its syntax, its semantics, and the process of building up data descriptions.

3.1.1 Abstraction Descriptions

MAL is a descriptive language. A MAL input file consists of a series of *Abstraction Descriptions* (ADs). An abstraction description is a list of items which characterize an abstraction that has been defined on the space of interprocess messages, giving the analyzer sufficient information to recognize unambiguously those abstractions. MAL allows for two distinct types of ADs, Transaction Descriptions (TDs), and Message Descriptions (MDs)

[8] Sungraphics is a registered trademark of Sun Microsystems Inc.

[9] "Speed" refers to the realtime speed at which the data is shown, "grain" refers to the amount of detail shown about a particular abstraction being displayed, and "level of abstraction" indicates which level in the hierarchy of abstract transaction types the user is interested in examining.

An MD (See Figure 3-1 for an example[10]) is the basic building block of a high level AD. An MD pattern is matched by a single message which appears in the input stream, although there may be more than one particular type of message which would satisfy a particular MD. An MD allows the user to focus upon those aspects of individual messages which are important in the recognition scheme, while abstracting away from other details. For instance, it may be useful to define a certain message type as having a particular message id, as well as being sent by a particular process type, but having a recipient of unspecified type.

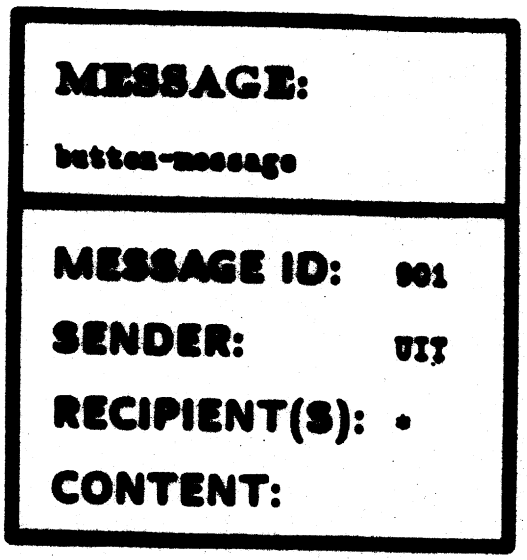


Figure 3-1:A Sample Message Description

[10]The representation in figure 3-1 provides an easy to read format for display of MD data, but does not represent the actual text that is provided to the Analyzer. See Appendix E for details on the actual input format.

A MD frame consists of the following slots:

Name: to identify the abstraction for use in higher level ones. Name is a unique identifier, selected by the user. In the figure, the name of the MD is **button-message.**

Message ID: a number which is the minimal form of type identification for a journal message. Every message has a message id, and it is this number which indicates to a recipient of a message how to handle it. The current implementation of MAL only allows for a single message id to be specified in an MD, so that message types are closely associated with message ids. However, a more general system would allow null or multiple entries in this slot, thus allowing a message type to include messages with a varieties of ids.

In the figure the message id is 901. Note that there is not necessarily a one-to-one mapping between message ids and message types. In the figure, the abstract message type **button-message** must have a message id of 901, *and* a sender of type UIT (User Interface Task). A message with an id of 901 but with a different sender type is not a message of this type. It is conceivable that the message id 901 might be used in the protocol between two other process types with a different meaning associated with it.[11]

Sender: constraints on the identity of the sender of the message. This slot may contain a '*', indicating that any process type is acceptable, or it may contain a list of process types, indicating that the sender must be from among those. In the figure, the sender is constrained to be a UIT process.

---

[11]Understanding the difference between **message id** and **message type** can be confusing. The message id is an artifact of the message, determined by protocols used by the designer of the program being debugged. The message type is associated with a class of messages which are described by an MD, and is the textual name for that class.

Recipients: constraints on the identity of the recipients of the message. Similar to sender, except that multiple expressions are allowed, one for each recipient. In the figure there is one allowed recipient, and it has no constraint on its identity.

Message Contents: constraint on the actual contents of the message. This slot is currently unimplemented. Possible constraints might include specification of the contents of certain positions in the message, length of the message, etc.

Higher level abstractions, which are composed of multiple messages, are described by TDs (See Figure 3-2 for an example[12]). A TD describes, in terms of other TDs and MDs, a pattern of message traffic associated with a particular transaction. TDs are the primary construct for allowing the user to impose a structure on a message stream. Since a TD can refer to other TDs, a hierarchy of abstraction levels can be built up.

A TD consists of the following frame slots:

Name: as in the MD case, a unique identifier, selected by the user. In the example shown in Figure 3-2, the name is **atg-interaction.**

Level: an integer assigned to the abstraction by the user. In an abstraction hierarchy, individual abstractions can be thought of as having a level number, such that all abstractions on a particular level only refer to abstractions on a lower level. Levels are primarily important in display playback, allowing selective attention to abstractions on an appropriate level. Since MAM has no way to infer the level intended by the user, it must be explicitly named.

Actors: a list of process parameters (actors) and associated constraints. Like an

---

[12]The representation in figure 3-2 provides an easy to read format for display of TD data, but does not represent the actual text that is provided to the Analyzer. See Appendix E for details on the input format.

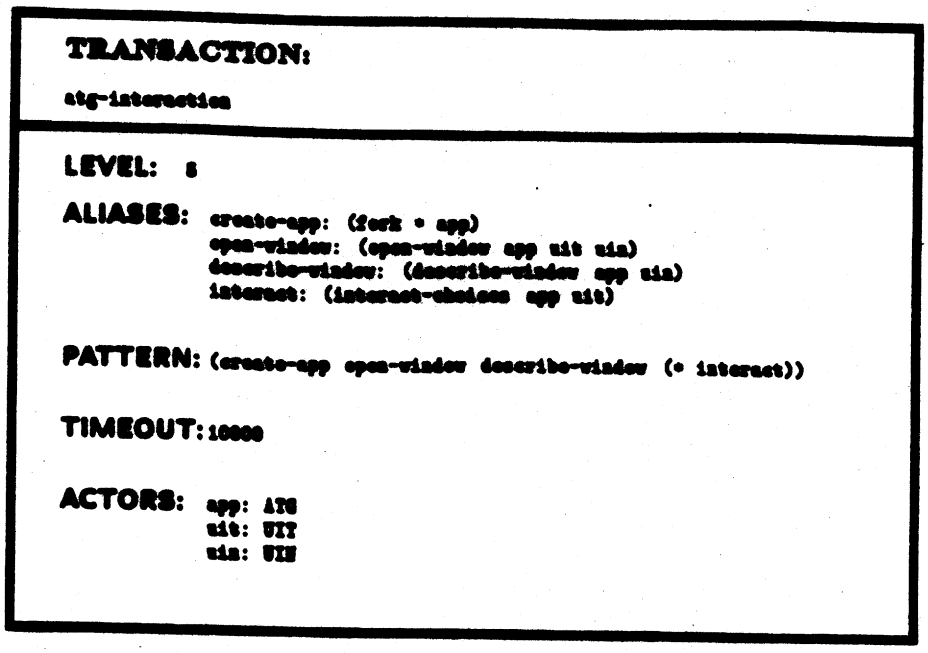


Figure 3-2:A Sample Transaction Description

MD, a TD can place constraints on the identity of the processes involved in it. In an MD, the role of processes in the description of constraints is relatively simple, each process being either the sender or recipient of the single message to which the MD refers. In the case of a TD, a more complex role is played, since a single process can act as a sender and recipient of many messages in the course of a transaction. By thinking of each of these processes as an actor (with a particular role to play), and assigning it a name, it can be "tracked" throughout the course of the transaction. Giving an actor an "identity" insures the consistency of processes in a transaction. In the example figure there are three actors defined: **app**, **uit**, and **uin**. Each is constrained to be a process of a particular type, **ATG**, **UIT**, and **UIN**, respectively. Note that the actor names may or may not be identical to the types which they are constrained to

represent. The choice depends on how the user views the role of the process. In the example, the **app**(lication) actor is viewed not as an **ATG** process *per se*, but rather as an application task attempting to establish a window. The **uit** and **uin** tasks are viewed as tasks performing special functions associated with their types, and are thus named accordingly.

Aliases: a list of the elements out of which the pattern for the transaction will be composed. They are called aliases because they assign a symbol or alias to represent a complex "call" to another TD. Aliases do not add any descriptive power to the TD abstraction, but do increase readability by allowing listing and naming of the elements involved, and by permitting simplification of the syntax of the regular expression specification. In the example, there are four aliases. The first **create-app**, expands to a **fork** request by an unspecified process of the **app** task. In the other three cases, the parameters are constrained to be one of the actors enumerated in the **actors** slot. Note how constraint comes into play here: the second parameter to the **create-app** must be identical to the first parameter to the **open-window** transaction.

Pattern: a modified regular expression made up of the elements defined in the alias section. In their canonical form, regular expressions consist of an alphabet of symbols, and some combining operators: Kleene start, disjunction, and parentheses. In MAL patterns, the regular expressions do not consist of patterns of atomic symbols. Rather, they consist of alias symbols, which are expanded out into parameterized subpatterns. This makes the task of pattern matching more complex, since each potential matching element is constrained not only by the current symbol in the pattern, but also by the instantiated parameters in previously matched elements. The pattern represented in Figure 3-2 consists of a **create-app** transaction, followed by a **open-window** transaction and **describe-window** transaction, and zero or more **interact** transactions.

Timeout: an integer representing the maximum number of milliseconds that a transaction might take to complete. This provides addition constraint on the recognition of valid transactions.

3.1.2 A Model for Understanding Data: MAL Semantics

MAL syntax provides a method for description of static relations between data abstractions, and this gives a good flavor of what can be described by the language. However, this information is not enough to permit the user to write full system descriptions. An analogy can be made between MAL and PROLOG. Both are descriptive languages, which can be used to describe formal relations among data. However, to understand the behavior of a PROLOG program, it is necessary to know more than the rules of first order logic (the static relations). It is also necessary to understand theorem proving and the unification algorithm (the dynamics of the engine).

The MAL recognizer is based on a demon model of recognition. Each AD is used as a template to spawn demons, specialized recognizers whose job it is to scan the world for instances of its associated abstraction. Demons do this by "gobbling up" instances of transaction components as they occur (in time, as the journal is scanned), and keeping track of what other items need to be recognized to form a completed abstraction. When a recognizer demon does find itself with a finished item, it signals this to the world, by announcing its completion to other demons. In this way, demons which might use this particular abstraction as an element can gobble it up in turn.

One instance of every demon is spawned as each message is read in. Demons which accept a sub-part continue to scan for the rest of the journal. Those which do not are immediately killed. When the end of the journal is reached, those demons which have found completed transactions then mark their sub-parts as members of the larger entity.

In Figures 3-3 through 3-6 a schematic representation of the transaction recognition process is given. The details of demon structure have been eliminated for clarity.[13] In Figure 3-3, a schematic diagram of the actors involved in journal processing, including transaction demons, message demons, and the messages themselves, is presented. In Figure 3-4, a Message Demon "gobbles" a message which matches its specification, and marks that message with its tag. It in turn is gobbled by a transaction demon which creates a link to it.

In Figure 3-5, the second journal message has been gobbled, and its corresponding message demon has in turn been gobbled by a transaction demon. The transaction demon, having completed its pattern, now is gobbled by a transaction demon of a higher level. Finally in Figure 3-6, the entire journal has been scanned, and all patterns are complete. Note that all abstraction information is contained both in the tags on messages, as well as in a global list.

Those demons which start but never finish are considered to represent near misses of the type "incomplete transaction." This consideration actually involves an assumption, namely that different transaction types are sufficiently orthogonal so that no pattern recognizer would inadvertently recognize some part of a valid transaction as an incomplete instance of some other transaction. This assumption can easily be justified by the observation that any sequence of messages that is ambiguous to a recognizer demon would also have been

---

[13]The two most important omissions in these diagrams are a description of the birth and death of demons, and a proper representation of the method by which messages are marked for membership. The diagrams show a static set of demons awaiting input. In actuality, many demons are created each time a new message is read in from the journal. Only those which actually accept input remain alive to continue pattern matching. It is only those "survivors" which are represented in the figures. Marking of message membership in a transaction, which conceptually occurs as soon as a message is scanned (as is shown in the figures), does not actually occur until journal scanning has been completed. This allows arbitration between competing partial recognitions.

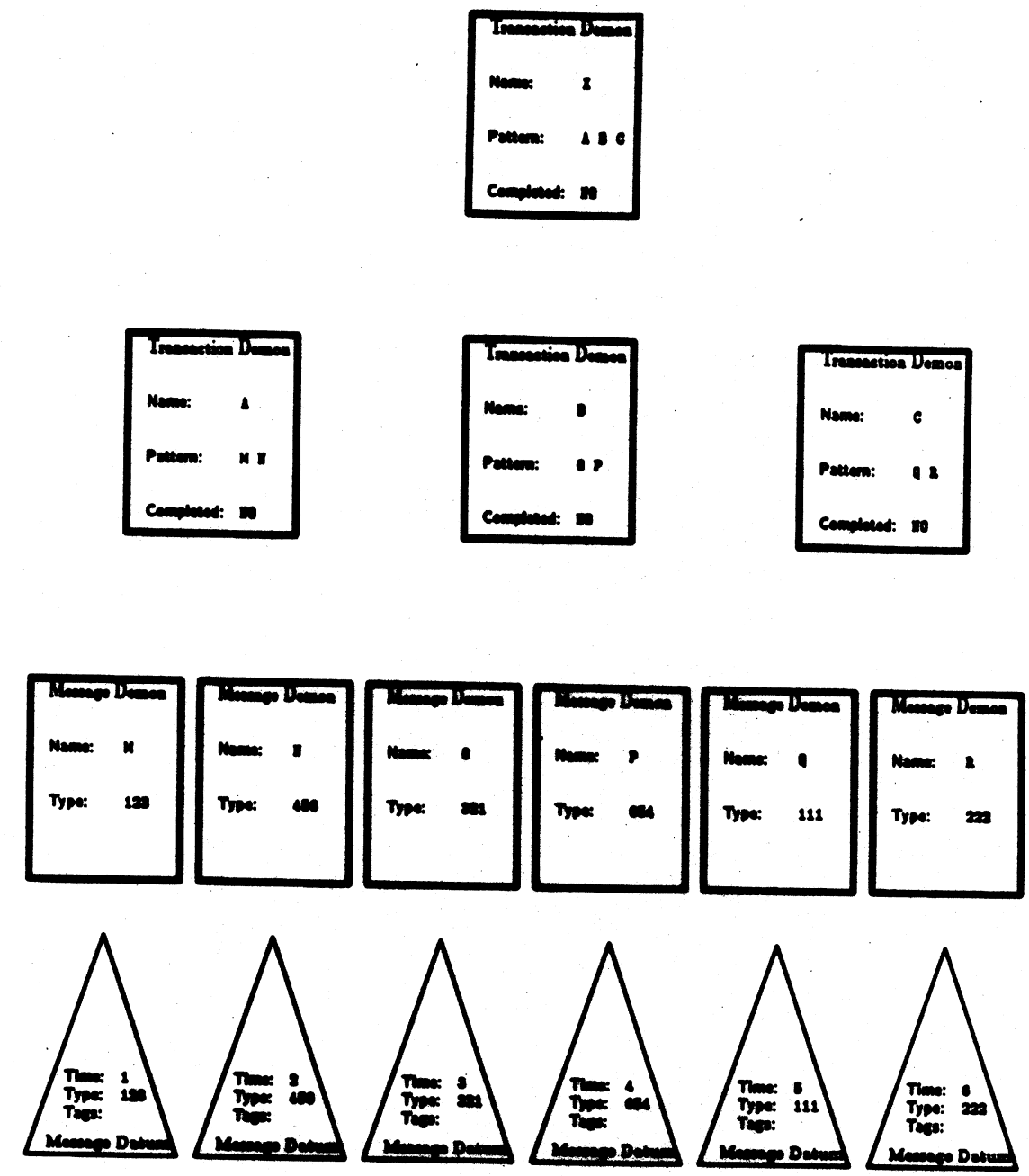


Figure 3-3:A Journal to be Processed

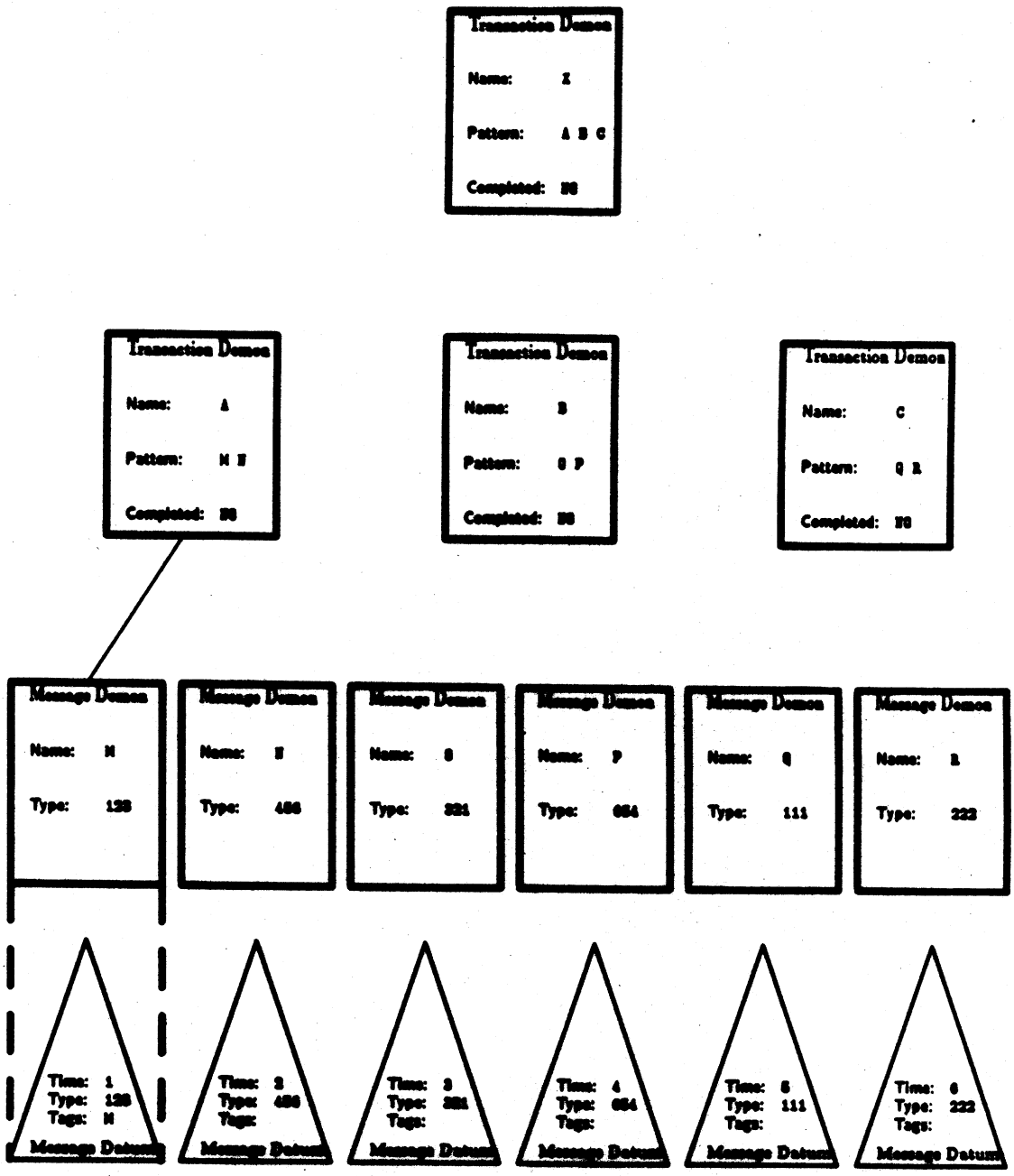


Figure 3-4:A Message is Gobbled

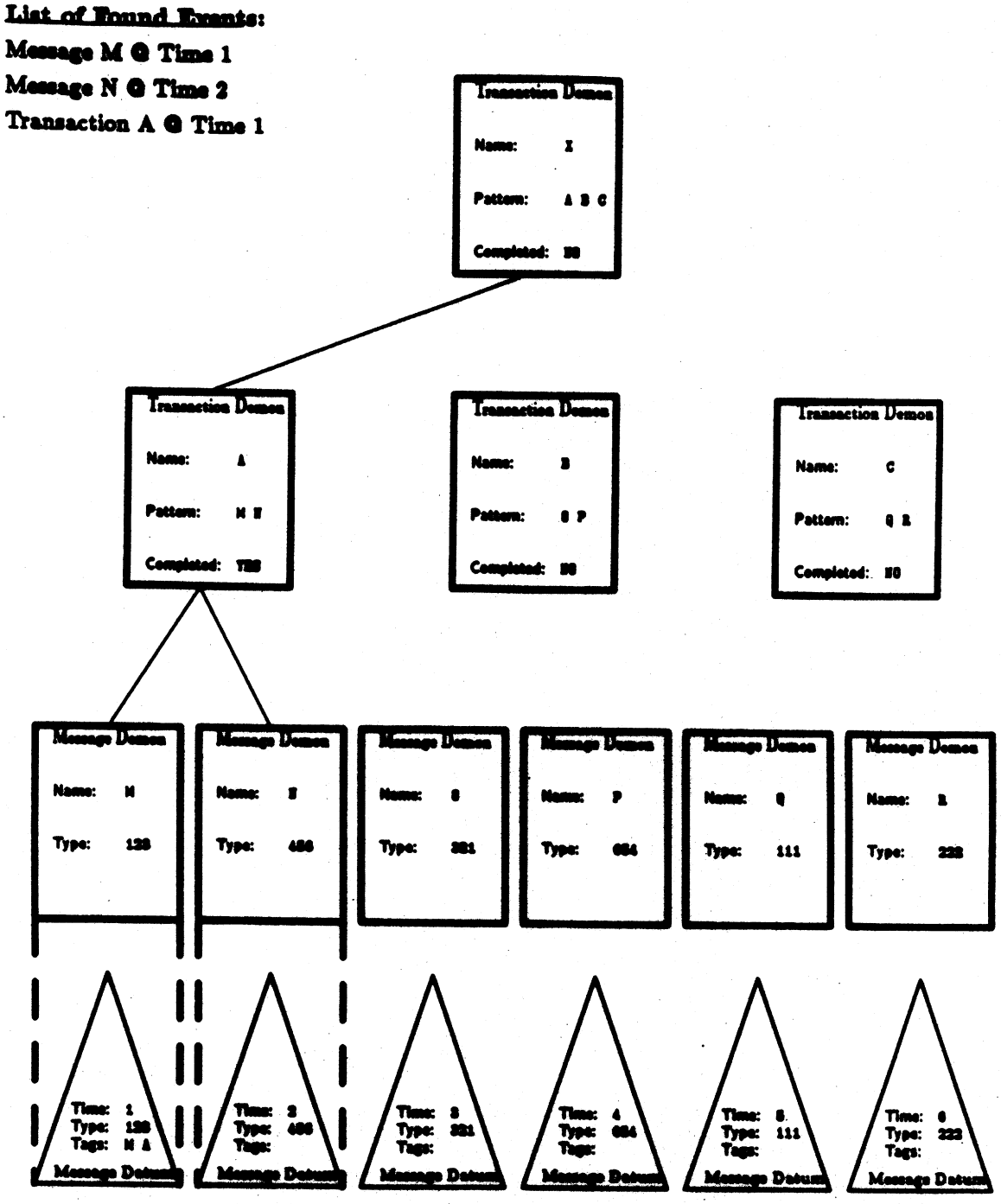


Figure 3-5:A Transaction is Recognized

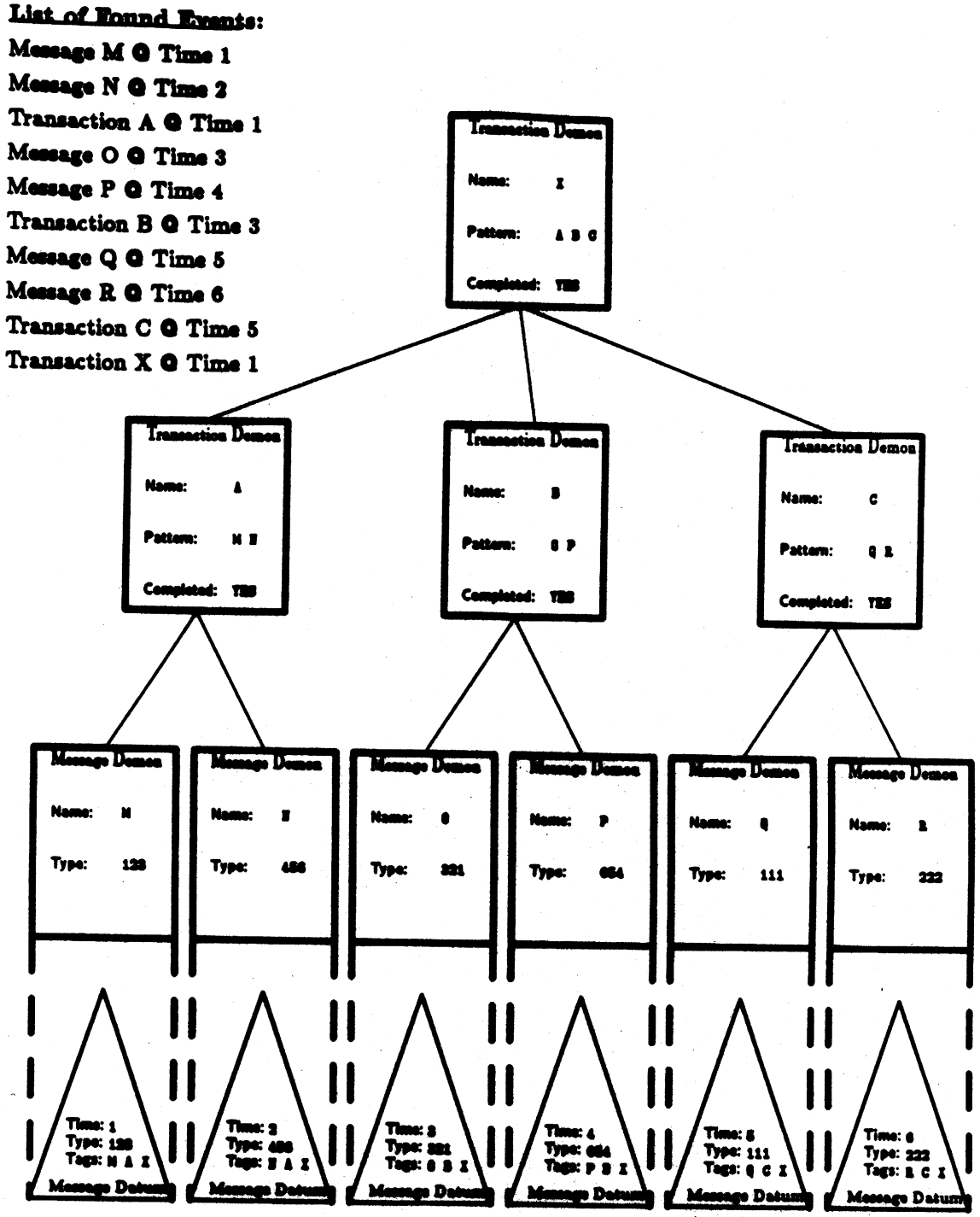


Figure 3-6: Journal is Processed

ambiguous in the original message stream on the network. Since it can be assumed that a programmer is not going to develop an ambiguous message protocol, it can also be assumed that message streams resulting from such a program will not be ambiguous.

Although the patterns that each transaction recognition demon recognizes are described by regular expressions, the class of patterns that MAM can recognize cannot be fully understood using only the terminology of regular languages. Because each item in the input stream is parameterized by process types, there is, in effect, an infinite input alphabet of symbols, which is divided into a finite set of classes. Recognition is not simply a matter of pattern matching, but also of constraint propagation. Each element of a pattern that is read in and accepted constrains which elements of the classes specified in other parts of the pattern may be accepted. This is done by requiring that parameter names shared by elements of a pattern have consistent bindings.

MAL patterns also differ from regular expressions in that they specify, not only a regular language, but also, implicitly, a language of "near misses" that are associated with elements of the "true language." These near misses include not only incomplete transactions, as already noted, but also "missing element" transactions. A MAL recognizer demon will recognize, as a near miss, any stream of sub-parts which would have resulted in a complete transaction had one more sub-part been present in the stream.

Explicit descriptions of such error patterns are generally quite complex. A missing element can be not only a single message, but an entire sub-transaction. Thus, patterns which are arbitrarily different from those described by the true language may be valid error patterns, depending on how sub-transactions were specified. Degenerate cases of totally meaningless error languages are unlikely to occur, however, since a programmer is likely to specify transactions in terms of

sub-transactions which are functionally meaningful (since this is the easiest way to describe a transaction). A language generated by the absence of one meaningful piece of the true language is likely to be meaningful as well.

3.1.3 MAL Limitations

Although MAL can describe a large class of transaction types that are likely to be encountered in a distributed system, there are certain classes of transactions which MAL is too weak to handle. Because MAL is essentially a regular expression recognizer, it is unable to recognize patterns which require a remembering of unbounded state information. Typical of this class of expressions is the form: A^nB^n. In a network message stream, such a pattern could correspond to the output of a process which queues its input, processing messages and replying to them in a FIFO fashion. A transaction involving queued input could not be described in MAL.

MAL is also restricted in describing dependencies between transactions. In the present system, the only constraints that can be invoked are those which are implied by a sharing of parameters (processes) by events within a transaction. A transaction recognizer has no access to global information that might predicate the validity of the existence of a transaction. For example, it might be useful to describe a certain transaction type as being valid only after some other transaction had finished executing, or involving a process that had NOT been a party to some certain other transaction.

Such limitations do not prevent the use of MAM in the situations described, they merely limit the correspondence between the descriptions that can be created and the user's model of the behavior of the system, since certain natural mental representations are not describable.

3.2 JDM: Journal Display Monitor

The JDM provides a meaningful and easy to use dynamic representation of a message journal, in terms of the abstractions described in a MAL file. In the JDM user view, the message journal can be thought of as a motion picture, and the JDM as a viewing system. The user may play-back the journal record at various speeds, rewind and start over, and zoom in for a more detailed look at the action.

Figure 3-7 shows an example of an actual JDM display screen. Figure 3-8 describes the layout of that screen. The screen is divided up into three sections: **map** (lower left), **control** (at top), and **status** (at right).

The **map** section contains a network map representation of the flow of messages in the journal. The **control** section contains various switches and meters. The **status** section is used to display explanatory status messages as the display progresses.

3.2.1 Map

The primary component of the screen is the network map. It provides a graphical representation of the message traffic that is captured in the journal being displayed.

Nodes in the network represent processes that are "active." An active process is one that is participating in a transaction on the current display level that has not yet completed. A process node first appears on the screen when the transaction of which it is a part begins in the journal, and remains displayed until that transaction finishes.

The relationship between nodes is represented by lines connecting them. Two nodes get connected if the transaction which caused them to be displayed

involved passing a message between their associated processes. When the message is actually encountered, the line is highlighted and labeled with the message type. Since there can be multiple recipients for a single message, more than one line may be labeled at one time.

The entire structure of a transaction can be gathered from the connections of its nodes. It can be assumed that the transaction of interest is composed of the transitive closure of n(a), the relation that maps nodes to other nodes to which they are simply connected. This assumption is justified by the intuitive notion that transactions involve communications between members, so if two processes are in the same transaction, there will be some chain of communication between them. Of course it would be possible to define transactions without such a chain, but it would not be a very useful thing to do.

In the example, two transactions are being displayed on the map simultaneously. A transaction called **describe-window16**, involving the processes **UIN666** and **TSD22** started at time 16, while the transaction **open-window17** involving the three processes **UIN11**, **ATG2**, and **UIT10** started at time 17.

3.2.2 Control

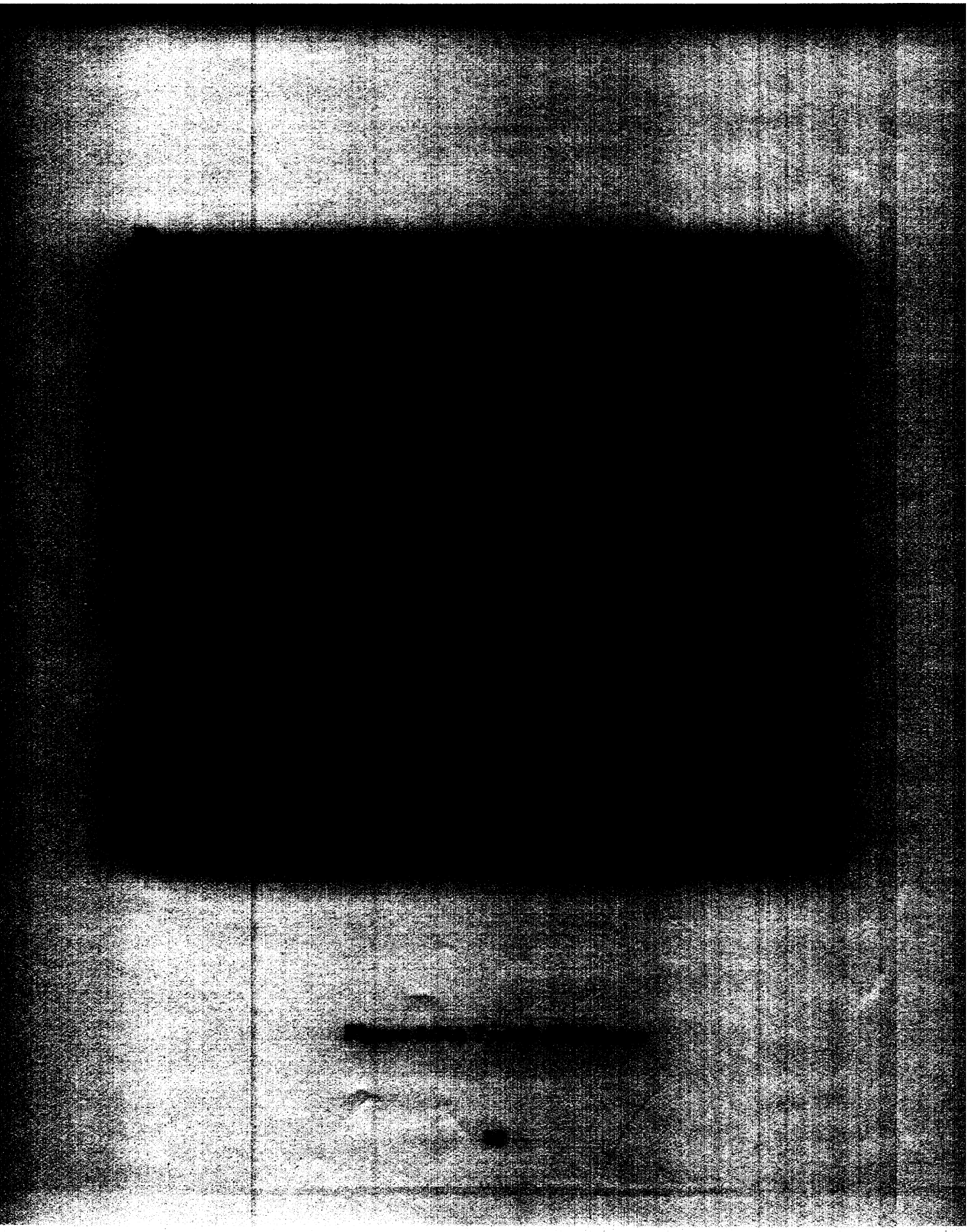
In addition to the main map display, the JDM provides the user with some additional graphical information.

There are two binary mode switches. These control the *continuity* and the *grain* characteristics of the display. If the *continuity* mode is **step**, then the user must manually advance the display after each step, by selecting the **step** switch with the mouse. If it is in **run** mode, successive steps are displayed until the mode is switched to **step** again, or until the end of the journal is reached, at which time the system reverts to **step** mode automatically. The speed of display in run mode may be varied by the user, as described below. In the example, the continuity mode is **step**.

The *grain* mode controls the amount of detail that is displayed in the **map** and **status** sections by determining the smallest time unit for each step. If the mode is **message** then the journal is stepped through message by message. Each message line is highlighted as a message passes along it, and a name label (indicating message type) is flashed over the line. The **status** area displays starts and ends of transactions at the appropriate level. Transaction structures are still drawn in the **map** display. In **transaction** mode, the grain is determined by the current transaction level. At each step, a search is made for the next transaction of the current level in the journal and it is displayed. Its name is listed in the status area, along with the names of its component parts. The transaction is displayed until the display clock passes the death date of of the transaction. A valuable extension to this would be to flash the individual transactions as they occurred, but this was not implemented. In the example, the grain mode is **transaction.**

In addition to the mode switches, the control area also contains two settable meters, for controlling *virtual time display clock* and *display delay.* The journal position meter is scaled in units of time determined by the journaling mechanism, and spans the period encompassed in the actual journal. Its value is referred to as the journal virtual time since it represents the current time in the JDM's virtual recapitulation of journal activity. By setting the meter to a particular time, the user can have the scanner begin displaying the journal at an arbitrary point. As the journal is displayed, the meter is constantly updated to give the user a graphical indication of the position of the scan relative to the start and end of the journal. The display delay meter determines the rapidity of display in **run** mode. Delays can range from approximately one-half second to five seconds per transaction. In the example, the delay is set to 51 (out of 100), while the journal virtual time is 17.

Finally, there is the *level choice* feature. This consists of a set of labeled



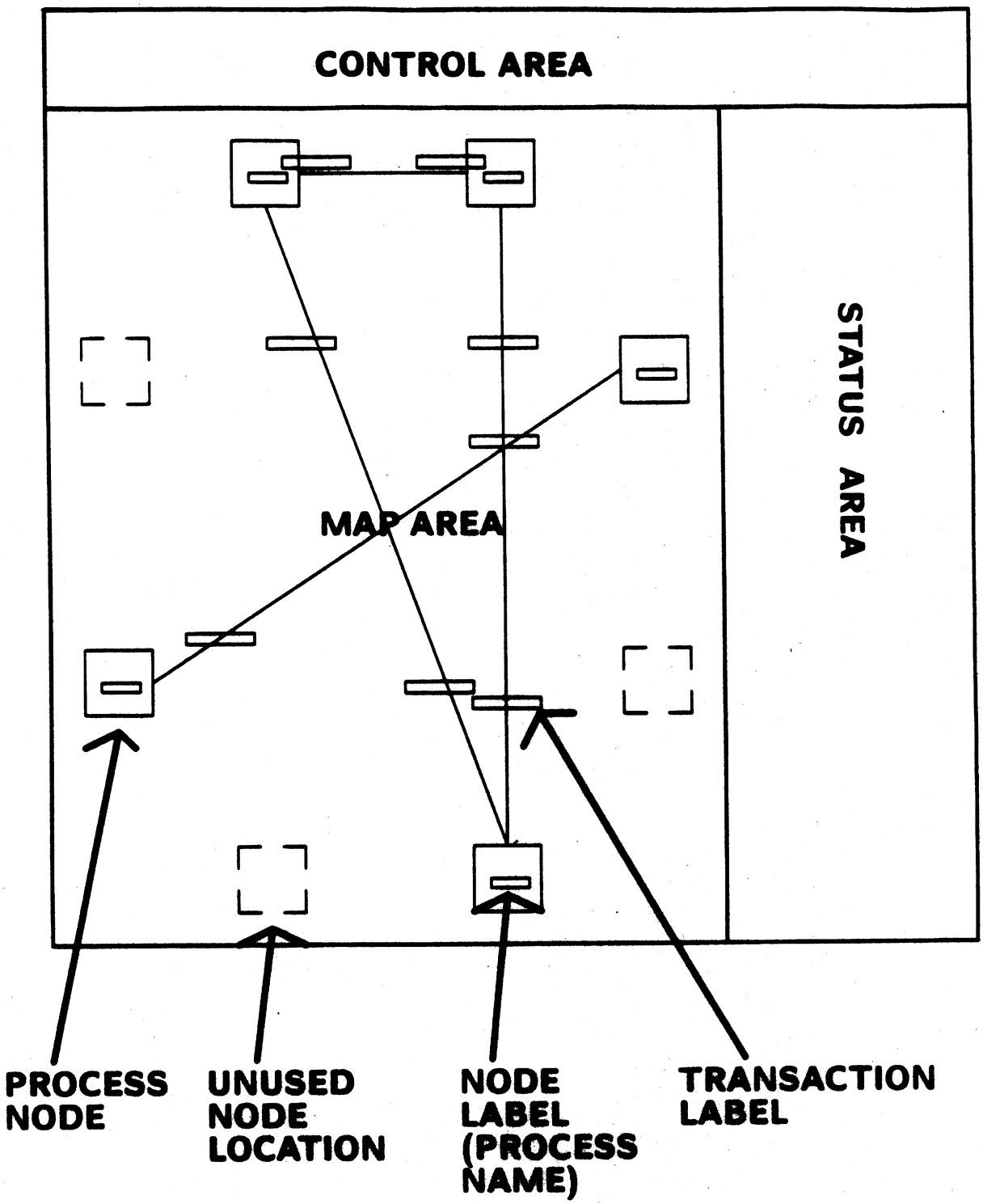


Figure 3-8:JDM screen layout for Fig. 3-7

graphical switches, each corresponding to one of the transaction level numbers represented in the journal, plus a **0** level for single message display . By selecting one of these, the user determines the level of transactions that will be displayed. In this example, the level choice switch is set to level **3**.

3.2.3 Status

The **status** section consists of a sidebar for messages which are used in conjunction with the map display. As data gets displayed on the map, notations for them get written to the status area. The map acts as a dynamic movie playback of the journal. This has the advantage of providing a realistic feeling for the behavior of the program. Its drawback is that the data it displays is ephemeral. The **status** area provides a more permanent record of what has recently transpired, thus augmenting the user's understanding from the map.

Status messages consist of four parts: a name, a time, an auxiliary message, and a structural outline. A status message is generated each time an abstraction is displayed on the map. The name describes a transaction or message that has appeared in the journal, the time refers to the time at which the starting message of the event was recorded in the journal during the run of the program, and the auxiliary message gives information concerning any irregularities in the transaction (such as missing messages). The fourth part, the structural outline, only appears when the JDM is in **transaction** mode. It lists, in the order encountered, the elements which make up the transaction. Status messages also appear when the end of a transaction is reached, announcing that end.

In the example, the side bar contains five status messages. The first three are for a transaction called **ATG-INTERACTION15**, which was an incomplete transaction, and which ended at timestep 21. It appears on the screen three times because the user went over the section of the journal containing the

transaction three times before continuing. The first two times were in **message** mode, so no structure is displayed. Because **ATG-INTERACTION15** is incomplete due to having reached a dead end, no ending status message is displayed for it. The last two status messages, for **DESCRIBE-WINDOW16** and **OPEN-WINDOW17** correspond to the two transactions displayed on the screen, and so there are no ending status messages for them either.

3.3 Examples

In this section two examples from the GenRad environment are described. The first example represents a contrived example, meant to demonstrate the descriptive power of the MAL language. The second example represents a real instance in which an error, to which behavioral abstraction techniques could be applied, occurred. For this second example, an actual debugging session is presented.

3.3.1 Example: A Fixture Test

In this scenario, the 2750 tester is performing one of its functions, a "fixture test." A fixture test is a process by which a printed circuit board is tested, using a variety of methods, to make sure that it works properly. Since the primary task of the 2750 is testing printed circuit boards, this is a very frequently occurring event in its normal operation.

A process known as "test set development" (TSD) serves as the system monitor, and interacts with the machine operator. When the operator indicates that a board needs to be tested, he indicates the type of board to be tested, and initiates the board test sequence. It is that sequence that is described below:

Board-Test-Sequence: The TSD initializes the system to begin board testing. It then activates the board "fixture."[14] Following this, one or more test programs[15] are run on the board, until enough test data has generated. An exit sequence then returns the test hardware to its idle state.

Although this scenario can, in the simplest case, consist of only four major actions, its smallest manifestation involves 30 separate messages being passed. In between the top level description and the message level description, there are a number of intermediate levels on which the scenario can be described in increasing detail.

As might be expected, a complex transaction can be viewed from a variety of perspectives, high level and low level. The exact form of the translation of an informal description of a distributed program scenario into a formal description depends in large part on the particular debugging needs of the user. One particular formalization of the above scenario is presented in Appendix A.

In this example, some of the general practical rules for designing MAL input files are demonstrated. The file is headed by a single top-level procedure **Board-Test-Sequence**, which describes the entire series of events which the user expects to transpire. Because all of the other events "hang" off of the **Board-Test-Sequence** transaction, certain global constraints can be enforced, such as consistency identities for the actor processes: **tsd, rte, rts, diag, ui.**

Another feature to be observed is that while some transactions, such as **Open-Window** might be described as general purpose transactions (in that they

---

[14] A fixture is a custom made device that interfaces between a particular type of board and the tester.

[15] Test programs are suites of physical tests on the board. They are run by a special purpose run time processor, and are not considered independent processes that are part of the distributed program.

can apply in a variety of contexts), others, such as **Load-Fixture** describe a very specific sequence of events. In general there will be, for some distributed program environment, a collection of general purpose transaction descriptions, which can be re-used between debugging sessions. It will, however, generally be necessary to make descriptions specific to particular debugging tasks.

3.3.2 Example: A Dead Process Causes Communication Breakdown

One of the tasks that is performed by the distributed program described above is window management. The system works with a bitmapped Sun workstation, and a variety of windows, corresponding to separate subtasks, may be active at any given time. From time to time, a process may require that a new window be created. This is done by a sequence of messages between the requesting process and special user interface processes.

In the scenario outlined below, problems were being encountered in a specific instance of establishing a window. A particular task, the *Automatic Test Generator* (ATG), was attempting to establish a window, but was experiencing a breakdown in communication with that window after it was generated. To investigate the problem, a set of transaction descriptions were created which dealt only with that part of the program which was of interest, namely the window interactions.

The sequence is as follows:

> **ATG-Window-Interaction: The ATG application is created. It requests that a window be created. That window is created, and informs that calling process (ATG) of its characteristics. A sequence of zero or more window interaction messages are sent from the *User Interface Task* (UIT) to the ATG.**

In practice the above sequence did not occur, and so MAM was used to determine the cause. In Appendix B the MAL input file for the debugging session is shown,

Appendix C shows the actual journal of messages of interest, while in Appendix D, screens from the actual debugging session can be found.

What did happen in practice is the following: In the process of creating a window for an application, the user interface task (UIT) created a special task, a *User Interface Node* (UIN), specifically to handle communication between the application task and the window being created. The UIN created in this case, due to a bug in the software, was exiting itself unilaterally. The UIT, seeing that the UIN task it had created had died, restarted it, *but with a new process id number.* As a result, this new UIN task continued to operate the window, but its messages to the application task were ignored, because it had the wrong process id number. From the user's point of view, this made it seem as though the ATG task was ignoring user input.

In the debugging process, the MAM was used to first detect the fact that the top level transaction was not being completed. This indicated that the problem was not in the ATG ignoring input, but in the input not reaching the ATG in the first place (i.e. the creation of a logical channel between the window and the ATG was never accomplished). This incomplete ATG interaction was detected automatically as a dead-end interaction by MAM. Then the view was shifted to lower levels, until the actual problem was determined.

3.4 Summary

This chapter presents the user's view of MAM, consisting of a description of the Message Abstraction Language (MAL), and the Journal Display Monitor. The syntax and semantics of MAL were described as well as examples of its usage. The graphics screen of the JDM was described, giving a full description of its features and functionality.

The MAL language permits the user to describe a hierarchy of transaction to be scanned for in a message journal. Transactions are described in terms of modified regular expressions, the components of which may consist of other transactions or individual messages. Because transaction descriptions are parameterized according to the processes involved, transaction descriptions may constrain the identities of processes involved.

The MAL analyzer acts by scanning the message journal for message types that it recognizes. When a message is recognized, a constraint is propagated through a system of transaction demons each of which is scanning for a particular type of transaction. When a transaction is recognized as complete, a downward propagation of this information ensues, resulting in a marking of all appropriate messages as members.

MAL is limited by its regular pattern expressions. Certain useful classes of transactions, such as those which involve queueing, cannot be represented. Additionally, more general constraints, such as numerical size or the absence of certain elements, cannot be expressed.

The JDM is the means by which MAM conveys its results back to the user. It consists of a graphical system for displaying a network map corresponding to the distributed program being debugged. The display is divided up into three main areas, the map itself, a control area by which the user manipulates the display, and a status area for explanatory messages.

Debugging with the JDM, as with serial debuggers, involves an iterative narrowing of the scope of observation. During the scan of the journal, a faulty transaction will be noticed in some high level. The journal can then be rescanned locally at progressively lower levels until the actually location of the fault is found.

Chapter Four

Algorithmic Details

This chapter discusses the details of the implementation of the MAL Analyzer, and the Journal Display Monitor (JDM). The MAL Analyzer is a program written in Scheme, a dialect of Lisp, and uses a message passing demon-based approach to recognizing patterns in the input stream. The JDM is a C program which utilizes heavily the SunWindows package for the Sun workstation. The two systems interface through a query database system called the Navigator.

4.1 Analyzer Implementation

The MAL Analyzer operates on a stream of input messages, marks them with data which identify them as members of higher level abstractions, and generates global information about the stream, including length, start and end times, abstraction levels present, and transactions present.

The bulk of the work in the Analyzer is done by message passing objects [Abelson85, pp. 140-142], which scan the incoming messages and, through their interactions, constrain the possible patterns that the input stream could match, until a final one is determined.[16]

[16] In Scheme a message passing object is implemented by means of a procedure with local state. Messages are sent to an object by applying its associated procedure. Thus, the only overhead required to maintain non-active objects is the small amount of space required to store code and a local environment frame.

4.1.1 Overview of Analyzer Algorithm

The objects that exist in the Scheme world created by the Analyzer are: a *list* of *transaction descriptions*, a *list* of *message descriptions*, *transaction recognition demons (TRDs)*, *message recognition demons (MRDs)*, a *journal* of *messages*, *recognition paths*, and *abstraction descriptions (TDs and MDs)*. The following is the top level algorithm for processing a journal:

```
FOR each message in JOURNAL:
  Instantiate TRD for each TD.
  Instantiate MRD for each MD.
  FOR each MRD instantiated:
    Ask each MRD to accept the current message.
```

In this way, each journal message is inspected in order by the analyzer.

Each MRD has the following algorithm for accepting messages:

```
IF message id matches constraint in MD, AND
    sender matches constraint in MD, AND
    recipients match constraints in MD, THEN:
  Accept message-datum.
  Set message-datum type to type of MD.
  FOR all existing TRANSACTION DEMONS:
    Notify TRD of message datum acceptance by MRD.
```

With a properly written MAL file, each message will be accepted by a single MRD, the one corresponding to its abstract type. Only the one MRD which accepts the current message remains an actor during the rest of the recognition process. The rest are discarded.

Each TRD has the following algorithm for accepting notifications:

```
Attempt to extend all current acceptance paths,
using the notification datum.
IF any path is extended THEN:
  IF any PATH represents a possibly complete recognition THEN:
    FOR all existing TRANSACTION DEMONS:
      Notify TRD of transaction recognition
      by this TRD.
ELSE
  IF TRD was created on current timestep THEN:
    Kill TRD.
```

During the scan of the journal many TRDs will be created. Only those which accept an element (by extending an acceptance path), remain alive. Those which actually complete a pattern, (by coming to the end of an acceptance path), inform other TRDs, so that they, in turn, may be subsumed as part of larger transactions, as the MRDs were before them.

When the end of the journal is reached, a process of committal takes place, in which the tentative relationships that have been developed are finalized. This works as follows:

```
FOR each extant TRD:
  IF TRD has a tentative winning path THEN:
    Mark all elements of transaction with
    membership tag.
  ELSE IF TRD has a "faulty path"
  with a missing element THEN:
    Mark all elements of transaction with
    "faulty transaction: missing element"
     membership tag.
  ELSE
    Make longest partial path the winning path.
    Mark all elements with
    "faulty transaction: dead end" membership tag.
```

Element marking is the means by which message data are marked for membership in larger transactions. Those elements of a transaction which are transactions themselves (as opposed to messages), simply "forward" the marking messages they receive, thus guaranteeing that messages are eventually marked.[17] While announcements of path completion occur as soon as a tentative completed path is discovered, marking of elements is delayed until the end of the journal scan. This ensures that all sub-elements, including those discovered after a tentative completion, are appropriately marked.

[17]For an example of how this works, refer to Figure 3-6. In this example, the TRD labeled **X** sends marking messages to its three elements, the TRDs labeled **A**, **B**, and **C**. These three in turn send a message to their children, the six MDs, indicating that those MDs are part of the transaction recognized by the TRD labeled **A**. In this way all messages which are part of a transaction ultimately get marked by that transaction, even though the TRD itself has no direct knowledge of those messages.

After all messages have been marked, global data is generated, by scanning all of the marked messages, and extracting birth and death information about all found transactions, and the elements of those transactions. This allows the JDM later to use both bottom-up and top-down information in its display algorithm.

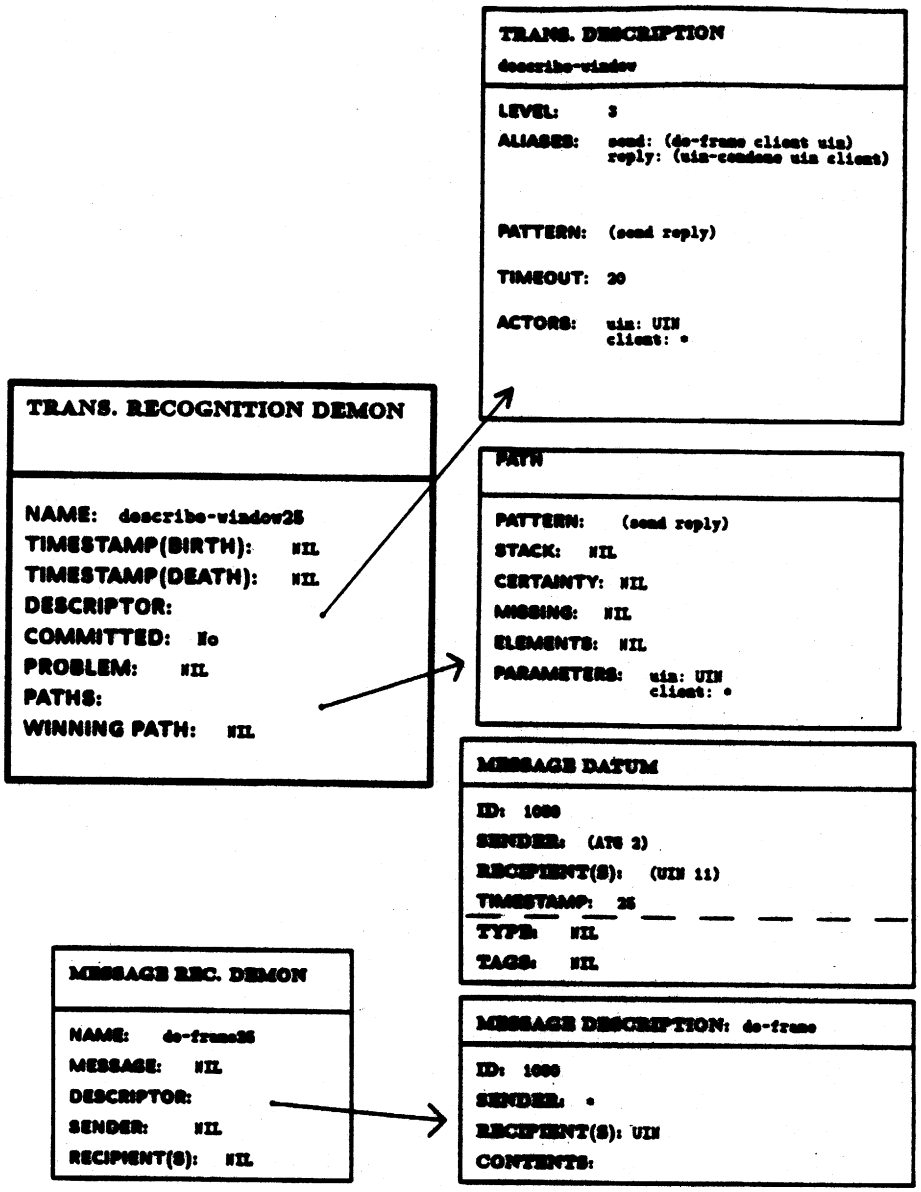
4.1.2 Data Structures

Figures 4-1 and 4-2 show some demons and associated data structures, both before and after a message acceptance. Figure 4-1, represents the state of those objects before the MRD has attempted to accept the message datum. Figure 4-2 shows the resulting relationships after acceptance. The TRD shown, **describe-window25**, and the MRD shown, **do-frame25** were created on timestep 25, just as the message datum was about to be scanned. Since the message datum matches the constraints of the MRD's associated MD, namely that it have a **message id** of 1080, and a **recipient** of type UIN, it is "gobbled" by the MRD. The MRD is in turn gobbled by the TRD, since it matches the leading element in the TRD's pattern, namely a **do-frame** whose second parameter is of type UIN.

4.1.2.1 Messages

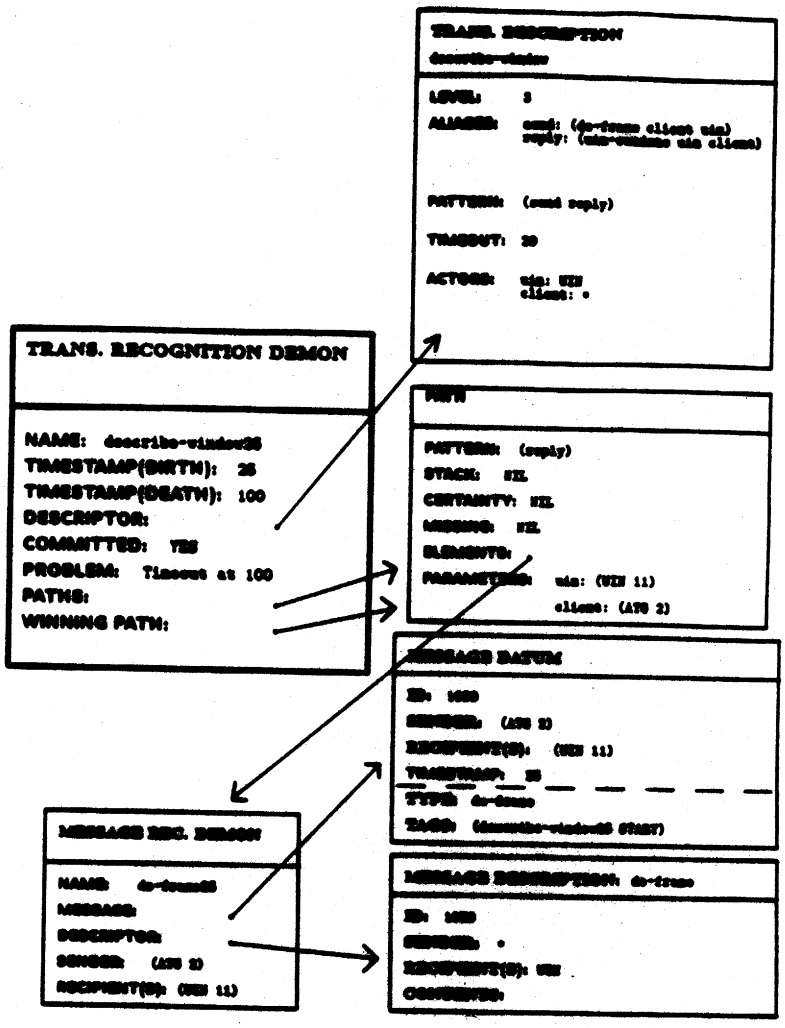
Messages are Scheme data structures which capture the important information about the journal message entries provided as input to the Analyzer. Initially, messages contain information corresponding to that represented in the unprocessed journal. During the course of the analysis, messages are marked by recognizer demons, as to their membership in various abstract structures. At the end of the analysis, these marked messages are written back out as a processed journal.

A message initially consists of a **message id**, a **sender**, a list of **receivers**, a **timestamp**, and its **contents**. This is all of the data that is typically found in



This figure represents the state of the **describe-window** TRD created at timestep 25 in the scenario presented in Appendix B, immediately before the current message datum is checked. Both the TRD and the MRD are in their virgin states, with initial parameters determined by their associated TD and MD respectively. The TRD has a virgin path associated with it which at its starting point.

Figure 4-1:A Simple TRD, and MRD, Before Gobbling Message



This figure represents the state of the **describe-window** TRD created at timestep 25 in the scenario presented in Appendix B, at the end of journal analysis. The **do-frame25** MRD has been gobbled by the path of the **describe-window25** TRD. The state represented indicates that a faulty transaction has been recognized, with a timeout at 100. In the TRD, the timestamps have been set, indicating that the first message of the transaction was seen at timestep 25, and the timeout was recognized at 100. The **committed** flag is set to YES, indicating that the demon is no longer open for modification. The path pattern has been modified to indicate a traversal past the **send** element in the pattern. The path parameters have been modified to indicate the new constraints for specific processes.

Figure 4-2:A Simple TRD, and MRD, After Gobbling Messages

the journal of a distributed system. A message also contains four writable slots which contain information about how the message relates to the rest of the journal. These are: **starts**, **ends**, **memberships**, and **message-type**. **Starts**, **ends**, and **memberships** are lists of transaction labels, indicating, respectively, transactions of which the message is the initial element, terminal element and intermediary element. The **message-type** is a slot filled in by a Message Recognition Demon (see below), which labels it with a name, consisting of a user defined type and a timestamp. This user defined type is independent of any intrinsic type information the message may contain.

An example of a message datum is represented in Figures 4-1 and 4-2. In the figure 4-1, the message can be seen in its virgin state, with the externally supplied characteristics, **id**, **sender**, **recipient**, and **timestamp** represented. Since the datum has not been marked yet, its **type** and **tags** slots remain empty. In 4-2, the figure has been marked, both by the MRD **do-frame25**, and the TRD **describe-window25**. The **type** slot indicates the type of MRD which accepted the datum. The **tags** slot indicates that this message is the starting element of a the transaction called **describe-window25**.

4.1.2.2 Message Recognition Demons

An MRD is an object whose job it is to recognize a particular type of message object, as defined in a MD. The data contained in an MRD includes the full MAL MD for its associated message type (see chapter 3). It also includes writable slots for the **message name**, consisting of its type and timestamp, the **message object** itself, and a **parameter list** consisting of the processes which were involved in the sending of its **message**.

In recognizing a message type, an MRD performs two major tasks: it marks individual messages with user-defined names and it initiates a propagation of

constraints which eventually results in the matching of user-defined transaction patterns. As each message is encountered in the scanning process, an instance of each type of MRD described in the MAL input file is spawned. Typically, all of these demons will "die" except one. Although there is no mechanism for preventing more than one demon from accepting a particular message, this would indicate a poorly specified MAL file, and the ensuing behavior is not well defined (i.e. the information contained in a message should indicate its type unambiguously, MAL specifications which indicate two possible types for a message are clearly faulty.) Avoidance of such conflicts is discussed further in Chapter 5.

A demon remains alive by "gobbling" a **message**. The message datum becomes a part of the internal data of the demon and is marked with the demon's unique name. The demon then broadcasts its existence to all of those TRDs which have indicated an interest in the message type of that demon.[18] TRDs may then in turn gobble the MRD, thus incorporating it into the transaction.

When an MRD broadcasts to higher level demons, it transmits not only its type, but also a **parameter list**, containing the names of the processes involved in sending or receiving the message. This parameter list is essential to the acceptance process of the TRDs, since it allows a demon to determine the relationship between the current message and ones that it may have already accepted as part of a pattern. Thus a message which fits into a transaction pattern by virtue of its type may still be rejected because, for example, the sender does not match that of a previous message related to it in the pattern.

---

[18]The system is optimized in a way such that only those TRDs which may potentially accept an event notification are notified of it. This potential is determined when the TRD is created. A global list, associating event types with current demons, is updated so that the TRD is listed as being "interested" in all event types which are mentioned in its pattern.

Parameters are determined by the specific processes involved in sending and receiving the MRD's **message**. The MRD binds each parameter to a process. The **sender** process is bound to the sender parameter. However, since there may be more than one receiver of a message, and thus more than one receiver parameter, the match of receivers with parameter names requires use of constraints to eliminate possible false matches.

In the definition of the MRD, each parameter is associated with a set of constraints on the type of process to which it may be bound. While a particular parameter might be able to be bound to more than one process from the message, it can be assumed that any meaningful MRD definition will be such that the ordering of the receiver parameters is irrelevant, or sufficient constraint is provided to eliminate all but one possible match. In the case of sufficient constraints this is done using an algorithm which generates "buckets" of all possible matches for each parameter, and then empties the buckets of already matched items, until each bucket has one unambiguous result remaining.

In Figure 4-1, an MRD labeled **do-frame25** is represented in its virgin state. In Figure 4-2, the results of its scan are shown. The **message** slot now points to the message it has accepted. The **sender** and **recipient(s)** slots now contain the process names culled from the message datum.

4.1.2.3 Transaction Recognition Demons

The *transaction recognition demon (TRD)* is, in many ways, analogous to the MRD. It starts out with a user-provided pattern data, and data slots for containing objects which match the pattern. It too searches for a certain class of data to pass before it, and when that occurs, informs other demons of its success.

As with MRDs, all TRD types are instantiated as each item is read from the message journal. A TRD can get killed immediately, if the input that spawned it

does not indicate a possible pattern start. However, those TRDs that survive this "birth" process, remain active for the rest of the journal scan.

Unlike MRDs, TRDs never directly access message objects. All information that a TRD receives come either from MRDs, or from other TRDs. In this way, TRDs act as redirectors and modifiers of the flow of information initiated by an MRD recognition.

A TRD exists as long as there is a possible completion of its pattern. For the purposes of transaction pattern recognition, the sole criterion for this standard is whether the demon's pattern recognizer gobbled any element on the timestep that it was created on. This rule is based on the assumption that there will be little overlap among element types between transactions, especially for initial pattern elements. Since every demon type is instantiated at every timestep, a demon which had not gobbled anything previously would be redundant.

TRDs keep track of potential pattern matches by means of data structures called *paths*, which are described in the next section. Each path represents conceptually a possible route through a finite state machine diagram for recognizing the regular expression associated with the TRD. All possible paths are maintained for the life of the demon.

When the journal has been completely scanned, every extant TRD is sent a "commit" message. On receiving this, a demon examines its possible paths and determines one which becomes its "winning path." Paths are ranked in the following priority: completed paths, paths representing "near misses", and incomplete paths. An unambiguous transaction description should not allow for more than one complete path. If multiple near misses or incompletes are at issue, a random choice is made. The commit sequence is necessary so that partially completed patterns are not counted as incomplete paths until the journal is exhausted.

If a demon recognizes a valid complete pattern, it sends a message to all TRDs indicating its recognition, thus allowing this information to propagate up to higher levels. There is no need for the demon to wait for the commit message, since the recognition of a complete, non-faulty, transaction is unambiguous. Higher level TRDs now have the opportunity to gobble this completed TRD and make it a part of a greater pattern.

The TRD also transmits a *parameter list* in a similar fashion to the MRDs. There is no positional ambiguity about the parameters in this case, however, since the parameters of a TRD are determined explicitly in the TD, and do not depend on the raw, randomly ordered recipients list of a message datum.

Figures 4-1 and 4-2 demonstrate how a TRD is modified during the recognition process. In this particular instance, the TRD involved ends up matching a faulty transaction (see Section 4.1.2.4). After the MRD, **do-frame25**, accepted the message datum, it signaled its acceptance to the TRD, which attempted to expand its path (see Section 4.1.2.4) using that MRD. Since the path did expand, the TRD remained alive, however no further elements were accepted. Thus, when it came time to commit, this demon had no completed paths, however its timeout interval had expired, and so it returned its longest path as a "timeout" faulty transaction.[19]

[19]The careful observer will note that the TRD was "born" at time 20, "died" at time 100, yet had at timeout interval of 25. The reason for this discrepancy is due to the mechanism by which timeouts are registered. A TRD will timeout if either: 1) it attempts to gobble an event when its timeout interval has passed, or 2) it receives a "commit" message after its timeout interval has ended, and has no complete paths. The death date is chosen to be the journal time at which such an event occurs. This time is likely not to be the birthdate plus the timeout interval.

4.1.2.4 Paths, Matching, and "Faulty Transactions"

A TRD is initialized with a single empty path and proceeds to accept event-notifications. At each event-notification, attempts are made at generating extended paths by an algorithm which attempts to fit the current input into one of the current paths. When a path is successfully extended, the algorithm is said to have *matched* the input to a path.

The matching algorithm recognizes regular expressions by simulating a modified nondeterministic finite state machine. Each **path** data structure represents a state in the NFA that might have been reached by the current input. When an input is checked, all possible routes from the current state are checked. If any of them can be traversed with the current input, then a new path is generated. Some states may have two valid outgoing paths (as in the case of a disjunction), so the number of paths tends to increase as the age of a TRD increases. This increase tends, however, to be fairly small, as the number of possible interpretations of partial message data tends to be small as well.

The following section describes the algorithm by which TRDs match an input stream of events to a particular pattern. It describes the **path** data structure, which holds the result of a partial match, and the matching procedure itself, which creates new paths.

The Path Data Structure

A **path** consists of the following substructures: a list of **elements**, a **pattern**, a **stack**, a list of **actors**, a list of **parameters**, a **certainty**, a **missing element**.

When a virgin **path** is created, it contains only a **pattern** and associated **parameters**, which are copied directly from the TRD data. The matcher takes the path, and an event notification to be matched, and generates a set of possible successors to that path which include that event. Each new path may differ in the following ways:

•**Elements** will now be updated to include the event tested. **Elements** consists of the actual demons gobbled, in the order of appearance. This information is used in a completed path for marking membership in lower level demons.

•The old **pattern** will be transformed into a new one, the matching of which would be consistent with the matching of the prior pattern before encountering the current event (i.e. corresponding to the "rest" of the pattern to be matched). In a simple pattern consisting of a sequence of symbols, this would mean removing the symbol just matched. In a pattern containing a Kleene star operator, two paths are generated, one with a pattern minus the first expression (indicating a "zero repetition match"), as well as one with the argument to the Kleene star operator substituted for the operator expression (preparing to match at least one instance of the argument).

•The **stack** might be pushed or popped. The **stack** contains pairs of "test points" and "return points" and is used to allow backtracking in the case of a Kleene star expression. When a Kleene star expression is encountered, the current position (return point) in the path is pushed on the stack, along with the entire pattern which succeeds the Kleene star expression (test point). A new pattern is generated with the Kleene star argument substituted for the Kleene star expression. When the test point is reached in this new path pattern, it causes a pop of the stack, and the original return point is restored as the pattern, thus allowing repetitive matches of the (possibly complex) Kleene star expression.

•The **certainty** of a path can be set to one of three possible values: **true**, **maybe**, or **nil**. It initially starts out as nil, and does not get set until its entire pattern has been matched. It is set to **maybe** it the stack is non-empty (i.e. it is still possible to accept more input via a Kleene star expression) and the pattern is empty. If both the pattern and stack are empty, then it is certain that the path is finished, and **certainty** is set to **true**. A path which is **true** or **maybe**

can make a TRD broadcast of its completion to the world. A path which is **maybe** may also continue to gobble data which fit its pattern.

•The **missing element** slot may be filled by a pattern symbol which, if matched, would have completed the transaction. This only happens if the path expression succeeding it was matched, thus indicating a possible "near miss" match (see section on "Atomic Symbol Matching" for details on how this works).

Matching Algorithm

A pattern consists of a list of subexpressions (see BNF grammar in Appendix E), each of which may be one of the following: an *atomic symbol*, a *disjunction*, a *Kleene star expression*, or a *zero/one expression*. Each of these, save the *atomic symbol* case, indicates some choice of possible matches, and can be analyzed by being broken down into constituent parts, and generating new paths for each alternative. The matcher works by checking the first subexpression of the pattern. If it is an atomic symbol, then a *symbol match* is attempted (see below). If the subexpression is complex, then it is expanded into a set of simpler expressions corresponding to the possible matches represented by that operator. This is repeated until all generated patterns contain atomic symbols as their first expressions, at which time symbol matches are attempted on the newly generated paths.

When the analyzer attempts to expand a pattern headed by a complex expression, it first divides the pattern up into three parts. The *operator* of the complex expression, the *argument(s)* of the complex expression (i.e. the thing on which the operator is operating), and the *rest* of the pattern (i.e. everything in the pattern list except the first expression). The complex expressions are expanded as follows:

The **disjunction** operator **$OR** indicates that either of two subpatterns may be

matched at the current point in the matching. It is expanded by generating two new paths, each one containing one of the two arguments of the disjunction appended to the rest of the pattern.

The **zero/one** operator ? indicates zero or one instance of the subexpression argument may be matched, and this is expanded by generating two paths, one missing the zero/one expression, and one having the argument appended to the rest of the pattern.

The **Kleene Star** operator * indicates that zero or more instances of the argument may be matched. In a nondeterministic finite state automaton (see for example [Lewis81]), this would be represented by having a "loop" back to the state at which the argument begins to be recognized, as well as a null transition edge which permits skipping of the recognition entirely. This is implemented in the path model by using a stack. (A stack is necessary because Kleene star expressions may be nested.) A Kleene star operator causes the matcher to generate two new paths, one with the expression eliminated, and one with the argument of the expression appended to the rest. At the same time, a pair consisting of the current position in the pattern and a pointer to the rest of the pattern is pushed onto the path stack. Using this "end/return pair" the matcher generates, whenever the endpoint specified by the stack pointer is reached, a new path identical to the one that existed before the argument to the Kleene star operator was matched. This has the effect of simulating the looping seen in NFAs.

Atomic Symbol Matching

It is at the moment of atomic symbol matching where decisions as to whether an event becomes a transaction element take place. It is also at this point in the pattern matching process that faulty patterns are checked.

If the initial subexpression of a pattern is an atomic symbol then it is tested against the current input event. If it matches, then a new pattern is generated, minus that initial subexpression. If an empty path results, then a complete match has occurred. If no match is made against the first expression, then the pattern is expanded once again, only with the first expression missing. If this new expansion results in an atomic match, then the resulting path become a *faulty match* path, with the missing expression noted in its **missing element** slot.

Symbols are matched against subexpression notifications by first expanding them into the subexpression calls for which they are aliases. A subexpression call consists of either a TRD or an MRD type name, followed by a list of parameter names. If the type of the demon (i.e. the type of transaction or message it represents) matches the type of the subexpression call then the parameters of the call are checked against the parameter list of the calling demon.

Those parameters which are as yet uninstantiated are tested by comparing the process type of the potential matching process to the constraints detailed in the constraint expression for that parameter variable. If there is a match the parameter is "instantiated" by replacing its original constraint with a constraint "binding" it to the matching process. Later in the pattern match (in this particular path) this parameter is required to match the same process. This insures parameter consistency across subexpressions within a pattern.

Already bound parameters are simply checked against the previously matched process. If the potential matching process is the same as that which previously matched the parameter than there exists a valid match.

If the type of the potential matching event is of the correct type, *and* all of its parameters satisfy the constraints, then a symbol is considered to be matched.

When this matching occurs, a new path is generated, consisting of the old path, *minus* the symbol just matched. This corresponds to the rest of the pattern to be matched.

Matching "Faulty Transactions"

The recognition mechanism in each transaction demon takes into account the fact that not all transaction descriptions will accurately describe the behavior represented in the journal. Some will describe expected behavior that never materializes. Others describe behavior, which materializes, but in a way slightly different from what is expected. It is analysis of these "faulty transactions" which presents the hardest recognition problem.

Because "faulty transactions" are derived from the definitions of *bona fide* transactions, there is a high degree of correspondence between the technical presence in a journal of a *bona fide* transaction and its faulty versions. That is to say, any time that a transaction can be recognized in a journal, any number of faulty versions of that transaction could certainly be incorrectly recognized as well (e. g. the transaction minus one of its elements could be recognized as a faulty transaction by ignoring the actual presence of that element). Thus it is necessary to allow these faulty recognitions to stand only in the absence of a "better" alternative.

Incorrect faulty recognitions are prevented by making the following assumption; no transactions of different types begins with identical messages types.[20] This can be done in the GenRad environment because of a protocol in which each

[20]Note: This is not the same as identical message ids. The message *type* is defined by a Message Description (MD) in the MAL input file, and may be valid only for certain types of sending or receiving processes.

transaction type begins with a unique message type.[21] Given this *non-ambiguity assumption*, it can be assumed that on any given level, only one transaction recognition demon (TRD) will remain alive for any given creation time, since any triggering event at that time should be accepted as an initial element by demons of only one certain type. Given this result, it can be seen that a particular partial path recognition need only be checked against other paths within a demon to see if it should be allowed to stand.

This disambiguation is performed at the end of the scanning processes. All demons are sent a "commit" message. When such a message is received, the demon checks all of its paths, checking first for complete recognitions. If no recognitions are found, than a check is made for "missing one element" matches. Finally, if none of these are found a search is done for incompletes and timeouts.

As soon as one of these searches turns up a match, that match becomes the winning match of the demon. If the match is a complete match, then a completion announcement is triggered, as well as a marking of elements. If a faulty match is found, only the marking takes place, as transactions are considered not to include faulty events in their matches.[22] Since completely matching paths are checked for first, faulty matches only manifest themselves in the absence of a complete match.

Example Path Sequence

---

[21]A more general system, to handle cases where such an assumption cannot be made, can be easily constructed with minor changes to the recognizer system. The primary change would require the addition of a means of choosing between multiple completed demons at a particular abstraction level, whose patterns start at the same timestamp. This could be accomplished by a check of pattern lengths of successful demons, and accepting only that demon with the longest path (indicating that it was a more complete representation of the events).

[22]However the absence of such a event could cause a "missing one element" faulty transaction recognition in a higher level TRD.

In Figures 4-1 and 4-2 a path for a simple pattern is represented. This pattern has two elements, **send** and **reply**, which alias to **(do-frame client uin)** and **(uin-comdone uin client)** respectively. When the MRD is accepted, it matches the **send** symbol, since the alias pattern contains the same message type and the parameter constraints allow for the parameters of the MRD. Thus, in Figure 4-2, the resultant path shows a new pattern, with the **send** symbol (having already been matched), removed, and with the parameters instantiated to be those of the MRD accepted. Since there are no *-operators in the pattern, the stack is not used, and since the path never completes (it is a dead-end), **certainty** and **missing** slots remain nil.

4.2 Display Monitor Implementation

This section describes the implementation details of the Journal Display Monitor (JDM), the graphical system for displaying the results of journal analysis. The task of the JDM requires that it display data in a fairly detailed manner, but this goal is constrained by the need to provide rapid data access and display update, as well as by the limited size of the graphics screen. Most of the information needed for data display is generated by the MAL analyzer, however there is still a significant amount of run-time computation that must be done by the JDM. This fact constrains the detail with which data can be displayed, making the JDM less effective than it might otherwise be. The rest of this section discusses a number of specific design issues which came up in the development of this system. As well as the solutions used in this work. These solutions provide a step in the direction of addressing the user-interface issues here, however what constitutes an ideal display is still an open question.

4.2.1 Unbounded Data/Finite Screen

A transaction journal might be arbitrarily large, yet because of the finite size of the screen, only small portions of it may be displayed at once. Human factors considerations also dictate that too much data on the screen at one time would decrease to user comprehension.

Data may be large in two ways. There may be many things going on at once, or many things may happen over an extended period of time. The former problem is the more vexing of the two, since a "movie-playback" paradigm requires that all data for a particular time slice be displayed simultaneously. If a single transaction has many processes (nodes) participating in it, or many transactions occur simultaneously, the ability of the system to display the state of the computation intelligently would be limited.

This problem is addressed in the JDM by assuming arbitrarily that no more than eight nodes would be active at any given time. Eight was chosen as the maximum number of node locations that could be exist on the screen in order to retain legibility and a comfortable user interface.

The length of the journal is also of concern. Specifically, as more processes are referenced in the journal, the task of assigning node slots to them in a consistent manner becomes more difficult. Although process nodes only appear on the screen when they are part of a current transaction, they may continue to exist in the interim between two transactions. When a node is subsequently redisplayed, as part of some new transaction, it is desirable that its slot position be the same as it was previously. This maintains the integrity of the visual representation.

Unfortunately, this is not always possible, since other intervening nodes may have taken the slot in the interim. The slot allocation algorithm is designed to minimize this possibility. When a node needs to be displayed, a table of

previously displayed nodes is first checked to see whether the node has been displayed previously. If it has, than the most recent display slot is reused. Only if that location is already taken, or if the process has never been displayed before, is a random location chosen.

If no more than eight nodes are represented in the course of the journal, then this system is guaranteed to place each node in the same unique slot each time it is displayed. If more than eight nodes are displayed during the course of a display session, then there is a small possibility that nodes will get "bumped."

4.2.2 Designing a Meaningful Display

As Model [Model79] has noted, graphical display can be, potentially, a much more powerful means of displaying complex data to a user than straight text. Unfortunately, there is very little in the way of a systematic method for determining the ideal display format for a particular type of data.

4.2.2.1 What Should a Message Look Like?

A message has two properties, *vis a' vis* the display. It is a transient occurrence, yet it is a building block of a larger object (a transaction), which is less transient. As a result, there are conflicting goals: to indicate the transient nature of the message by displaying it for only a short period of time, yet also to keep its relationship to the larger transaction known for the lifetime of the transaction.

The solution in the JDM is to display transactions in terms of their messages, and to highlight the individual messages as they occur. This is done by flashing the name of the message type over the line in the network that represents it, and making that line momentarily bolder.

4.2.2.2 How Should Messages Be Related?

The entire structure of a transaction is based on the actual messages which pass between processes. These are the glue which bind processes to each other, and it is in terms of message passing relations that processes are connected in the display screen. The JDM uses a single bi-directional line connecting two process nodes to indicate that, at some point in the currently displayed transaction, a message is sent from one of the pair to the other. Entire transactions are recognized by a collection of nodes linked together.

This method is easy to understand and requires relatively little computation. The journal is scanned for the start of a transaction, and when one is found, each process involved in that transaction is assigned a node on the screen. Then each process is checked to see with which other processes it will communicate during the life of the transaction, and a line is drawn for each case.

4.2.3 Control in Terms of Abstractions

Aside from the actual generation of graphic images, the major computational task of the JDM is to keep track of the dynamic status of messages with respect to the data currently displayed on the screen. Ideally, a control mechanism for a display system will allow specification of controlling commands in terms of a mental model that the user has of the data being displayed. It should not be necessary to perform any on-the-fly conversions of data in order to converse with the debugger. That is to say, if the user understands the journal to be a sequence of certain high level transactions, he should be able to control the flow of the display of that journal, in terms of those transactions (e.g. a command to go to the start of the next transaction), without having to compute any lower level information about them (e.g. their start or finish times).

Although the "flow" of the journal is determined by the stream of individual

messages encountered, what the user expects is a stream of abstractions. The JDM attempts to maintain an illusion of "movie-playback" on the selected abstraction level by providing a rapid sequence of transactions displayed and maintaining the map structure across transaction boundaries (i.e. there is no repainting of the screen).

4.2.3.1 Keeping Track of Abstractions

Part of the data generated by the Analyzer is a list of *event data structures*. Each structure describes an instance of some event found in the journal by the analyzer. The information included is: level, start time, end time, the elements of the transaction, its size, and, in the case of a faulty transaction, an error message. Using this information, when the start of an abstraction is encountered in the journal, the entire structure of the transaction can be displayed, without it having to be computed. That is to say, it can be displayed in terms of its elements.

4.2.3.2 Map Management

When an abstraction is displayed, many graphical objects are placed on the map screen. These must be tracked. In an n-actor transaction, n map nodes are displayed, as well as up to n(n-1)/2 connecting lines. Some of these nodes may disappear at different times than others in the same transaction (if, for example, transactions involving the same node are intertwined). There are two structures which keep the display accurate, the *actor display array* and the *pending transaction array*.

The actor display array maintains the status of each of eight potential nodes. This information includes whether or not the node is currently being used, what the last actor to use the node was and, if the node is in use, the number of transactions currently being displayed of which the node is a part. Whenever an

actor node is modified (either by activating it, or adding connections), its corresponding status structure is updated. When a node's number of connections is zero, it is erased. If an actor is about to be displayed, all nodes are checked to see where (if any place) it was last displayed, and that node is chosen, if possible.

The pending transaction array contains structures which keep track of all of the parts of a transaction currently being displayed on the screen. Each element contains a transaction data structure as described in the previous section, a list of map locations of the members of the transaction, and a list of messages which make up the transaction. As each message is encountered in the journal, pending transactions are checked to see if the message is a member. If it is, the appropriate line is flashed on the screen (determined by the locations of recipient nodes). If the message is the last message in the transaction, then all nodes involved in the transaction have their transaction count lowered, and those with a count of zero are erased.

4.3 Summary

This chapter presents a detailed look at the structure and implementation of MAM. MAM consists of two separate modules, an Analyzer for scanning a journal file and recognizing transactions in it, and a Journal Display Monitor, for displaying the results of the analysis. Understanding the implementation, at least on a structural level, is necessary because it provides the user with a more complete model of usage of the MAM tools.

The Analyzer uses a message passing model of programming in order to recognize user defined transaction in a journal file. Software *demons* are created which recognize particular types of abstractions which have been defined in a file of *abstraction definitions*. Abstraction definitions come in two flavors, *transaction definitions* and *message definitions*.

Recognizer demons are passive collectors of syntactic information. Demons receive messages from other demons, indicating events that have been recognized. The demons use those recognition messages to build up higher level abstractions, sending out their own recognition messages in turn. Such sequences of message passing are initiated by message demons, which scan the journal directly for messages of some class.

Transaction demons recognize patterns by simulating non-deterministic finite state machines. The inputs to these machines consist of event recognition messages, which are matched to pattern symbols. Matches are contingent not only on event type, but also on the parameters of the event (i.e. the processes involved in it), which must meet certain constraints. These constraints may be user specified, or they may be cause by the previous instantiation of a parameter.

Demons also contain mechanisms for recognizing so-called "faulty" events. These are events which contain most of the elements of a pattern, but fail by missing a given element, by not finishing in the time specified by a timeout field, or by simply partially completing. Checks are made for partial matches after the scanning of the journal has been completed. By using a "non-ambiguity assumption" partial matches need only be checked against complete matches within a particular demon, thus making the task of weeding out false faulty transaction matches relatively simple.

The Journal Display Monitor uses a movie-like graphics display to represent the flow of transactions read from a journal that has been pre-processed by the Analyzer. The primary issues involved in the JDM implementation were ones of ergonomics. The data to be displayed was readily available and required little additional computation. The primary problem was one of presenting the data in a way which was intuitively useful to the user.

The three major issues are ones of user control, display design, and screen management. A user's mental model of the behavior of a distributed program may be very complex, involving many levels of understanding, and control of the display should be possible at any of these levels. The actual representations are also important, as they must correspond in some useful way to the structure of the transactions being displayed.

The control problem is vexing because it involves a tradeoff, between speed and functionality. In order to give the user maximal control, a great deal of record keeping would be required, so that the program could keep track not only of transactions that were being displayed on the screen, but also of those that were not. This would allow a user to switch levels arbitrarily at any time in the display. However, the current implementation does not support such a scheme. Instead, only the current message and currently displayed transaction are tracked. This allows the user control on a per message basis, or by entire transactions on a given level.

The representation problem is difficult because there are very little in the way of hard design rules to follow. A simple map representation is used, augmented by a textual "commentary" of the abstractions being displayed. Nodes on the map correspond to processes of the distributed program, while lines connecting the nodes represent messages passing between them.

Map management involves not only keeping track of the various graphical objects on the screen, and keeping that screen display current, it also involves making sure that screen changes preserve the integrity of the screen representation over time. The primary task is to insure that the correspondence of screen nodes remains static over time. This is done by keeping a record of previous correspondences, and reusing them as much as is possible.

Chapter Five

Summary, Critique, and Future Work

The work presented in this thesis is the result of an investigation into improving the tools available to debuggers of distributed software systems. Because of the paucity of research into the field of distributed debugging, there was no firm foundation of tested strategies on which to build improvements. Instead, some ideas were taken from the few *ad hoc* approaches which had been tried in the past, but most of the work proceeded from scratch. The result, the Message Abstraction Monitor, provided an implementation with which to empirically judge the value of the approaches used.

This chapter is divided into four sections: The first is a summary of the work performed, the second is a critique of the MAM tool, the third suggests future improvements and extensions, the fourth one is a concluding statement.

5.1 Summary

This thesis presents the results of an investigation into debugging distributed programs. Debugging distributed programs is similar in some ways to debugging serial programs, but there are some important differences which must be addressed. A number of researchers have attempted different approaches to this problem in the past, using a variety of techniques, none of them particularly satisfactory. MAM, a program developed as part of this investigation, combines a number of techniques from previous works, as well as developing a novel approach to the problem of faulty data.

Serial debuggers are fairly well understood and all implement the same basic paradigm. They allow the user to trace, step and dump. It would be useful to be able to do the same for a distributed program, however those functions are less well-defined in that domain. One does not want to consider individual program line steps in a distributed program, because they are not well ordered. Instead, it is useful to look at processes as black boxes and debug them from a functional standpoint. The goal is to understand a program simply from the input/output behavior of individual processes. This introduces the complication that program object data and the indications of program flow control (control data) are one and the same, and understanding the flow of a program becomes dependent on interpreting the data that it produces. There are two new issues that this "behavioral abstraction" approach brings to the fore: 1) The data is a raw stream of messages, and usually not very meaningful to the user in this form. There is no "source code" to which to return in order to make sense of a dump the way a program line debugger does. 2) If there are mistakes in the data, that is messages which do not correspond to any program behavior that is expected, the debuggers grasp on the control flow of the program is lost. These two observation must be taken into account when designing a distributed debugging tool.

MAM is an attempt to address the issues brought up in the last paragraph. It allows the user to specify abstractions for which to scan in a journal of messages. MAM provides a program for scanning that journal and marking it according to the abstractions provided, and it provides a method for "playing back" that journal in a graphical display monitor, allowing the user to understand what happened in the journal in terms of his model of the behavior of the program, rather than in terms of the details of message passing interaction. It includes a method for automatic recognition of "near-miss" transactions, which are faulty in some way, but which are a close fit to the expected transaction.

MAM is a custom implementation for a particular task, namely debugging a distributed programming system which was being developed at Genrad. This system involved a suite of programs, with zero or more instantiations of each program running at any given time. These programs communicated by sending messages (synchronous and asynchronous) back and forth on an ethernet. A central message server coordinated communication between processes. There were no explicit layers or enforced high level conventions in the network, just a single protocol for sending and receiving messages. The central message server generated a log of events for use in debugging. These events consisted of messages, forking of tasks, and restarts. Using this journal as the means of getting "into" the program operation, as opposed to some real-time intervention, seemed the cleanest approach.

MAM consists of two parts: an Analyzer written in Scheme, and a Journal Display Monitor written in C using the SunWindows package. The Scheme program is non-interactive. It is given an input file of Abstraction Descriptions, written in MAL (Message Abstraction Language), and a journal produced by the message server. It outputs a modified journal, with messages marked according to their membership in various abstract entities, as well as some global information.

MAL allows the user to define message abstractions and build up higher level transaction types. This is done by specifying regular expressions whose components are "calls" to lower level abstractions and, ultimately, to single message. In this manner a whole hierarchy of descriptions can be created. Errors and near-misses are detected using simple rules for partially completed transactions and transactions missing one element. Such "faulty" transactions are included in the output file and displayed as events in the journal playback.

The display system takes the finished journal and displays it on a graphics

terminal. It can be thought of as an enhanced movie playback of the journal, with the user controlling both the speed of playback, and the "magnification" (level) at which it is viewed. The user can jump between levels, and move backwards in time, thus allowing him to focus on the problem. The various processes involved are displayed as nodes on a map, those which communicate are connected with lines, and those lines are highlighted as messages appear. A sidebar gives a running record of activity as it happens in the virtual time of the display.

Because of certain assumptions about the nature of the data being recognized, the semantics of the MAL are simpler than they might otherwise be. For example, it can be assumed that any pattern for which a search is performed must be recognizable by the processes that are listening for messages. This allows the Analyzer to ignore the mathematically possible but pathological cases for recognition. Even with such constraints, there are still possibilities for ambiguity, due to the nature of the language. Certain assumptions have been made about the most likely interpretations of ambiguous cases.

The notion of a near miss is not well defined. It is not just a matter of syntactic difference, but also of semantic difference. It was assumed that a faulty abstraction would, on some level, simply be a single missing abstraction, so the system merely recognizes abstractions minus no more than one of its expected components.

The abstraction recognizer of MAM was implemented using a message-passing programming model. Each abstraction is used to spawn demons, which scan the journal as it "passes by." When a demon recognizes an abstract entity, it informs the world, so that its information can be utilized by other demons. Each demon keeps track of the current state of its recognition by data structures called "paths." Each path kept by a demon corresponds to a possible

interpretation of the data. Since data is assumed to be unambiguous by the end of the journal, false paths can be eliminated. An algorithm for eliminating false paths was developed.

The Journal Display Monitor system had a number of interesting implementation issues. Specifically, determining the type of graphics display that would be most effective in giving the user some intuition about the nature of the journal was difficult, due to a lack of a strong theory on the subject. It was found that trying to keep processes at the same physical location in the map (even if there was a hiatus during which the process was not displayed) was very helpful. A system of priorities for node assignment was developed for this. There were also some questions about how to implement the two "modes", namely message and transaction. In message mode, the display proceeded one message at a time (although the entire transaction structure was displayed). In transaction mode, the grain was the entire transaction. The problem here is that this really is not the duality wanted. Transaction mode should display a transaction *in terms* of its elements (with the actual pattern displayed on the sidebar). This was difficult due to the way in which transaction data was stored.

5.2 Critique

MAM provides three primary innovations towards distributed debugging: a specification/description language for interprocess events, a graphical display system for journal play back, and automatic error detection. The MAL language analyzer provides a scheme for specifying debugging data structures which are not implicit in the program (i.e. they depend on the user's understanding of the program semantics, and cannot be automatically generated from the structure of program or its output). More specifically, it uses *behavioral abstraction*, treating interprocess communication as the finest grain of program control. The JDM

uses a graphical network map display as the means of representing the dynamic behavior of a distributed system. It also provides the user with control of the display in terms of high level abstractions. Because the MAL analyzer can recognize "near-misses", it is able to pinpoint the sources of unexpected behavior in a distributed system.

5.2.1 Error Detection is a Powerful Tool

MAM, as its name suggests, was originally envisioned primarily as a *monitor.* Users would observe a processed journal, using the JDM, and check what they saw against expectations. In fact, the most useful feature of MAM, in practice, appears to be the "near-miss" error detection capability. This is particularly true with the large class of errors which result from coordination and stray message problems.

Because of the hierarchical structure of MAL specifications, almost all "reasonable" message streams come close to matching user defined abstractions; (the more faulty messages in the stream, the higher the level of abstraction needed to abstract all faulty messages into a single faulty transaction). Thus, the user is likely to find that the MAM can pinpoint the exact location of most faulty behavior automatically.

It is important to note that useful errors were found with MAM using a very simple error checking scheme. One-missing-element, and transaction-dead were the only tests used by MAM. Perhaps with a more sophisticated error checking technique, even better results might be obtained.

5.2.2 MAL Is Not Strong Enough, and Hard To Use

The descriptive power of MAL is limited in part by its similarity to a regular language, and in part by features native to it. Like other regular languages, MAL expressions cannot express the need to keep track of unbounded state information. MAL expressions cannot capture the notion of something that is "pending." In a system such as the one implemented at GenRad, where every send must be acknowledged, issues of pending results generally do not come into play.

It is easy, however, to conceive of a system in which valid transactions could only be described in terms of nested calls. Such a system would involve a client and a server. The client sends, at random intervals, various requests for service, to which the server must, before the transaction ends, respond. A valid transaction would be one in which all requests eventually receive a response. MAL expressions would be unable to represent this.

Even in a system with acknowledgments, a similar problem could occur if, for example, some action in a transaction depended on the number of previous transactions observed. This observation leads to another perceived shortcoming, the inability to calculate. In a more general system, statistics about the messages observed, such as the number of them, could be put into variables, and used as predicates in recognizing transactions. This might be useful in a situation where a transaction is only valid in case a certain number of transactions of another type have already been observed.

Negation is not implemented in MAL. It is impossible to describe a pattern position as NOT being a certain type. Because regular expressions are defined on finite alphabets, NOT clauses in a regular language are just syntactic sugar[23] But

[23]Take for example the a regular language on the alphabet (A B C). To say that a symbol is NOT(A) is equivalent to saying that is is OR(B C).

the alphabet for MAL is arbitrarily large (since symbols are parameterized), and thus NOT is indispensable. It might be useful, for example, to indicate that a pattern position should not be matched by an event type *with particular parameters*, although the events of that type in general could match.

Another problem which was discovered, and which is inherent to all description languages, is the frequency of errors in the abstraction descriptions themselves. In debugging a large program, abstraction descriptions can be rather large, and subject to errors. In using MAM, it was found that as much time was spent debugging the MAL input file as was spent debugging the actual distributed program. This was due in large part to the lack of syntax checking in the MAL analyzer. A scheme for overcoming this problem is discussed in the next section.

5.2.3 The JDM Lacks Sufficient Display Power

The JDM is lacking in some respects as well. The control mechanisms are somewhat crude, relative to the complexity of the data. The screen display, while useful in interpreting the data, requires further work to make it maximally useful to the user.

The JDM allows two types of flow control modes. In **transaction** mode, transactions are flashed atomically, for a single instant, and then erased, the display moving on to the next transaction at that level. In **message** mode, transactions are displayed, and then each of the messages comprising that transaction is highlighted as the journal time progresses. Neither of these modes displays the transaction *as it is understood by the user*, as a sequence of abstract elements. This is due to the inability of the display system to represent that amount of information at one time.

5.2.4 Control Mechanisms Are Not Ideal

The lack of understanding of transactions in terms of events presented a problem for control, as well as display. Ideally, the control level and the display level should be independent. That is to say, the level of transactions that are being displayed on the map does not necessarily need to be the same as the level of transaction being used as the timestep in **transaction** mode. In the current implementation these levels are the same.

The control/display combination that might be the most useful addition would allow the user to display transactions on a particular level, and control the flow of the display in terms of events which are the lexical constituents of the transaction pattern, regardless of which level they are on.

Another problem with the JDM concerned changes of level. A frequently used debugging technique involves progressively lowering the abstraction level number, in order to home in on the faulty message. The JDM is incapable of maintaining a meaningful screen display across level changes. That results from the fact that the JDM does not keep track of the current transactions in levels other than the one that it is currently displaying, and as a result cannot display transactions at other levels without encountering them while at their level.

5.3 Proposed Enhancements

5.3.1 MAL Language

5.3.1.1 A Syntax Checking Editor

As mentioned previously, the major difficulty in using MAM effectively came from the difficulty in insuring the validity of the MAL specifications. In actual

test runs, detecting errors in MAL syntax and typographical errors in the input file were responsible for the majority of time spent in the debugging process. In complicated systems, with perhaps tens or hundreds of events defined, errors in the semantic content of ADs are likely to become significant as well.

One can imagine a number of safeguards that could be built into the Analyzer to recognize problematic abstraction descriptions. Simple syntax checking is easily implementable and quick, given the very simple structure of MAL expressions. More important, however, would be what might be called "consistency" checking. Many of the MAL errors encountered involved misspelled words or errors of omission in transaction patterns. Mistakes such as these cannot be caught using syntax checking alone. Instead, checks can be made, using certain heuristics, to insure that expressions appear "reasonable."

One such heuristic involves insuring that every alias defined in a transaction is actually used in the pattern associated with it. Another involves checking that elements are of a lower level than the transaction that encompasses them. Misspelled process types (which are used in aliases to constrain the identities of parameters) can be caught by maintaining a master list of valid process types. Even with error checking, the user is still presented with the problem of typing in a specification file, running it through the analyzer, finding the errors, and rechecking. Such a routine is likely to *add* time to the debugging process, rather than make it easier. Fortunately, the simplicity of the MAL language makes a better solution feasible.

A *syntax-directed editor* [Teitelbaum79] is a system which guides a user in generating code for a language, using knowledge of the syntax of that language. A syntax-directed editor is likely to prove very valuable in a future version of MAM, due to the simple and regular structure of MAL expressions. The structure of MAL expressions is essentially that of *frames*, with *slots* to be filled.

The content of each slot is at least partially dependent on the values of other slots. As a result of this fact constraints can be utilized while MAL expressions are being written, in order to force the user to write valid expressions.

Such an editor might also be used to enforce overall coherence in the input file. For example, a test could be performed on the **level** attribute of various transaction definitions. In the current implementation, the **level** attribute is set at the whim of the user, but this can lead to problems. For levels to be meaningful, there should be a partial ordering on the transactions, such that a transaction only contains elements on a lower level. This enforces a more important rule, namely that transactions should not have so-called "cyclical definitions." That is to say, an event should not refer to its parent transaction in one of its elements. The editor could check for this by constructing a graph of the relationships of transactions, and then checking for cycles.

Some form of machine assistance in the creation of abstraction descriptions seems vital to the creation of a feasible abstraction based debugging monitor. For any program of modest complexity, the abstraction descriptions will compare in size to the program itself. If creation and debugging of the abstraction descriptions takes more than a negligible amount of time, the entire project is for naught.

5.3.1.2 A More Powerful Language

The MAL language could be improved to make it more powerful. This would be done at the cost of more difficult journal analysis. Making a richer language would also entail more complicated pattern expressions, thus complicating the task of generating such expressions correctly.

The most obvious enhancement would be to abandon the restriction that MAL patterns be regular expressions. Instead, expressions with the expressive power of turing machines (i.e. a full-fledged programming language) could be used. Such

expressions could have local variables, perform arbitrary tests on the input (which would still be subexpressions). Less ambitious improvements might add features to the language such as stacking, negation, etc.

5.3.2 Monitor

5.3.2.1 An Enhanced Display

A major difficulty in designing the monitor display was the inability to display enough information on the screen at any given time. The information available about the state of computation is multi-layered and hierarchical. This aspect is very difficult to display on the current screen without creating unacceptable clutter.

A solution to this problem lies in the use of a color display monitor. As it stands now, the JDM has no way of displaying multiple levels of structure on the screen at once. A color coded display would help to solve that problem, using distinct colors to represent levels.

Alternatively, multiple abstraction levels could be displayed simultaneously by using a multiply-windowed display screen, with each window devoted to the display of a single level. The major impediment to this scenario is screen size. Having the five to ten windows that might be common in such a situation on single display screen would seriously limit the size of such windows, thus limiting the detail of display in those windows. However a limited system allowing a smaller fixed number of windows is quite feasible.

5.3.2.2 Control In Terms of Substructures

The current control mechanism of the JDM is primitive by comparison to the richness of description it is capable of displaying. The flow of control is managed in two modes, which allow a message by message time grain, or the display of entire transactions at the current level as atomic units. Ideally, such control should be more general, using more of the constructs generated by the analyzer.

Specifically, the following scheme might be implemented: Another mode **subtransaction** is added to the two, **message** and **transaction**, which already exist. In this mode, there exists a **current level**, just as in the other two. But in this case, it is not the transactions at the current level that are displayed one at a time, but the elements which make up the current transactions on the current level. In conjunction with a color display, the map could be further improved by highlighting those nodes which participate in a particular event, as it is occurring.

5.4 Conclusion

The MAM project was an empirical investigation into an approach to the debugging of distributed systems, a field almost devoid of previous theoretical work. The resulting work is significant in that it explores some of the areas for which a theory might later be developed, user interface, abstraction languages, error detection. Of these three areas, the work presented here on error detection is the most novel. The combination of the three areas provides a valuable contribution to the debugging of distributed systems.

Appendix A

MAL Frames:A Hypothetical Transaction

The MAL frames presented in this appendix represent transaction descriptions for a hypothetical debugging scenario, which is used to demonstrate some of the descriptive abilities of the MAM system.

There are seven TD frames listed, **board-test-sequence, init-fixture, init-rte, run-program, quit, test-result,** and **open-window,** as well as ten MD frames. Together, they serve to describe the following algorithm:

> **Board_Test_Sequence:** The TSD initializes the system to begin board testing. It then activates the board "fixture."[24] Following this, one or more test programs[25] are run on the board, until enough test data has been generated. An exit sequence then returns the test hardware to its idle state.

The frame **board-test-sequence,** at level **5,** serves as the top level or *root* frame and its pattern describes the entire transaction from a high level view. Notice how the *-operator is used to indicate that one or more run-program transactions may occur as part of this transaction, as noted in the description above.

The children transactions, assigned to level **4,** are those which are elements of **board-test-result,** including **load-fixture, init-rte, run-program,** and **quit.** The remaining transactions, **open-window** and **test-result,** are conceptually

---

[24] A fixture is a custom made device that interfaces between a particular type of board and the tester.

[25] Test programs are suites of physical tests on the board. They are run by the special purpose run time processor, and are not considered independent processes that are part of the distributed program.

elements of level **4** transactions, and so are given the still lower level of **2**.[26]

[26]Note that there are no transaction of level **3**. There is no requirement that level numbers be consecutive integers. Leaving certain level numbers unused allows addition of more levels later on, if more detail is needed.

TRANSACTION:

board-test-sequence

LEVEL: 5

ALIASES:
init-rte: (init-rte tod rte rte diag ui)
load-fixture: (load-fixture tod rte rte diag)
run-program: (run-program tod rte rte ui diag)
quit: (quit rte rte diag ui tod)

PATTERN: (init-rte load-fixture run-program (* run-program) quit)

TIMEOUT: 1000

ACTORS:
tod: TOD
rte: RTE
rte: RTE
diag: DIAG
ui: UIT

TRANSACTION:

init-rte

LEVEL: 4

ALIASES:
start-rte: (fork tod rte) open-win: (open-window rte ui *)
start-rte: (fork rte rte)
start-diag: (fork rte diag)
rr-ack: (ack rte rte)
dr-ack: (ack diag rte)
rt-ack: (ack rte tod)

PATTERN: (start-rte start-rte start-diag rr-ack dr-ack rt-ack open-win)

TIMEOUT: 100

ACTORS:
tod: TOD
rte: RTE
rte: RTE
diag: DIAG
ui: UIT

TRANSACTION:

load-fixture

LEVEL: 4

ALIASES:
tr-lf: (load tod rte)
rr-lf: (load rte rte)
rr-fi: (fix-is rte rte)
rd-fi: (fix-is rte diag)

PATTERN: (tr-lf rr-lf rr-fi rd-fi)

TIMEOUT: 100

ACTORS:
tod: TMD
rte: RTE
rte: RTE
diag: DIAG

TRANSACTION:

run-program

LEVEL: 4

ALIASES:
t-r: (run tod rte)
u-r: (run ui rte)
r-r: (run rte rte)
test-result: (test-result rte rte diag)

PATTERN: ((0or t-r u-r) r-r test-result (* test-result))

TIMEOUT: 100

ACTORS:
tod: TMD
rte: RTE
rte: RTE
ui: UIT
diag: DIAG

TRANSACTION:

quit

LEVEL: 4

ALIASES:
tod-quit: (quit-kill tod rte)
ui-quit: (quit-kill ui rte)
rr-done: (done rte rte)
rd-done: (done rte diag)

PATTERN: ((0or tod-quit ui-quit) rr-done rd-done)

TIMEOUT: 100

ACTORS:
rte: RTE
rte: RTE
diag: DIAG
ui: UIT
tod: TOD

TRANSACTION:

test-result

LEVEL: 2

ALIASES:
rr: (result rte rte)
rd: (result rte diag)

PATTERN: (rr rd)

TIMEOUT: 100

ACTORS:
rte: RTE
rte: RTE
diag: DIAG

TRANSACTION:

open-window

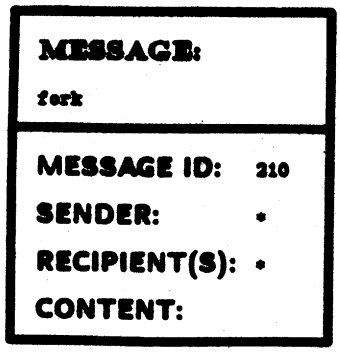
LEVEL: 3

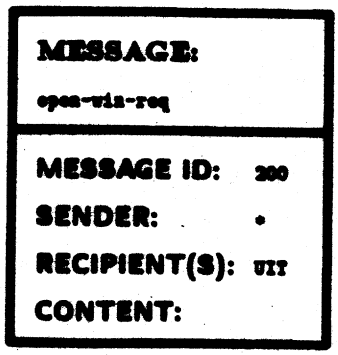
ALIASES:
req: (open-win-req client ochsd)
fork-req: (fork ochsd window)
ack: (o-win-req-ack window client)

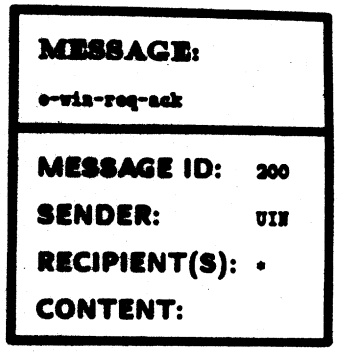
PATTERN: (req fork-req ack)

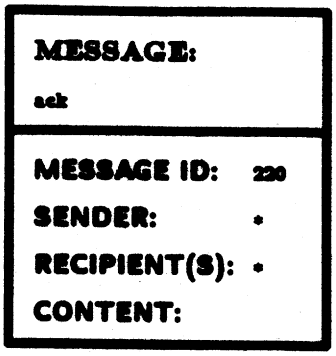
TIMEOUT: 20

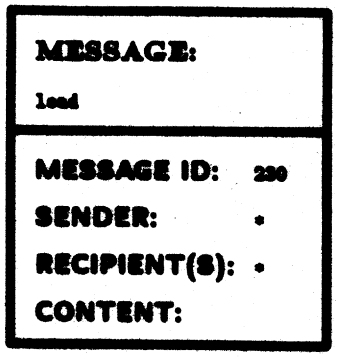
ACTORS:
client: *
ochsd: UIT
window: WIN

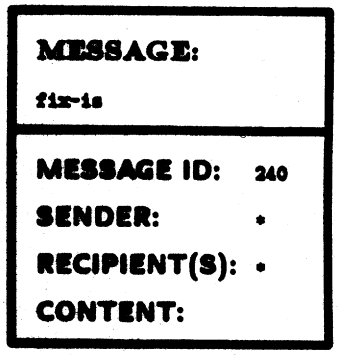

MESSAGE:
fork
MESSAGE ID: 210
SENDER: *
RECIPIENT(S): *
CONTENT:

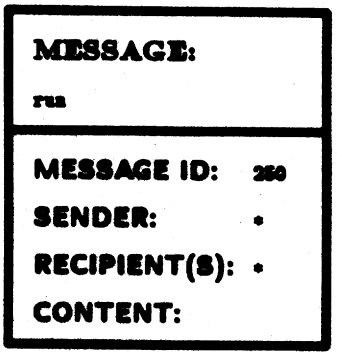

MESSAGE:
open-win-req
MESSAGE ID: 200
SENDER: *
RECIPIENT(S): UIT
CONTENT:

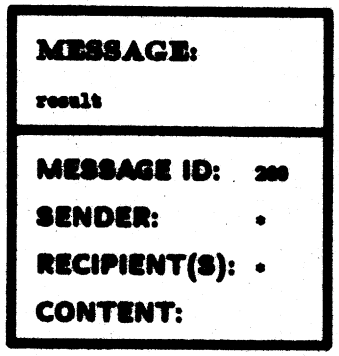

MESSAGE:
o-win-req-ack
MESSAGE ID: 200
SENDER: UIH
RECIPIENT(S): *
CONTENT:

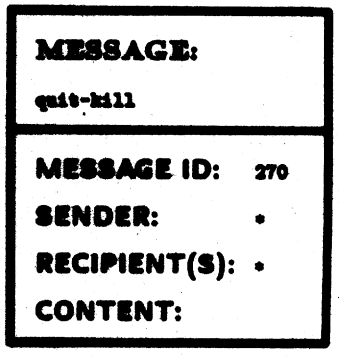

MESSAGE:
ack
MESSAGE ID: 220
SENDER: *
RECIPIENT(S): *
CONTENT:

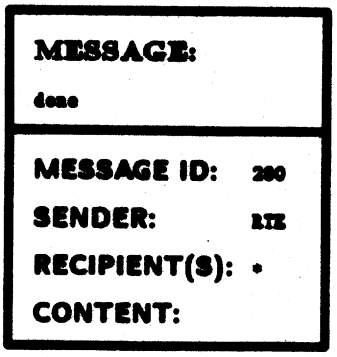

MESSAGE:
load
MESSAGE ID: 230
SENDER: *
RECIPIENT(S): *
CONTENT:


MESSAGE:
fix-is
MESSAGE ID: 240
SENDER: *
RECIPIENT(S): *
CONTENT:


MESSAGE:
run
MESSAGE ID: 250
SENDER: *
RECIPIENT(S): *
CONTENT:


MESSAGE:
result
MESSAGE ID: 260
SENDER: *
RECIPIENT(S): *
CONTENT:


MESSAGE:
quit-kill
MESSAGE ID: 270
SENDER: *
RECIPIENT(S): *
CONTENT:


MESSAGE:
done
MESSAGE ID: 280
SENDER: RIK
RECIPIENT(S): *
CONTENT:

Appendix B

MAL Frames: An Actual Scenario

The MAL frames presented in this appendix represent transaction descriptions for an actual debugging scenario, which was encountered during the development of the GenRad system. It describes transactions found in the journal described in Appendix C.

There are four TD frames listed, **describe-window**, **atg-interaction**, **interact-choices**, and **open-window**, as well as seven MD frames. Together, they serve to describe the following algorithm:

> **ATG_Window_Interaction:** The ATG application is created. It requests that a window be created. That window is created, and informs the calling process (ATG) of its characteristics. A sequence of zero or more window interaction messages is sent from the *User Interface Task* (UIT) to the ATG.

In this scenario, the root frame is **atg-interaction**, which describes the open ended pattern of three initialization transactions, followed by zero or more I/O interactions.

Since this transaction contains fewer parts than that presented in Appendix A, the "level" structure is not as clear. In this case, the elements of **atg-interaction** are "general purpose" transaction types, which are used in a variety of situations, and whose levels are determined by their functionality, apart from their usage in any particular context. Thus two transactions, **describe-window** and **open-window** are assigned levels of **3**, which in this case was used for transactions involving setting up user interface windows. **Interact-choices**, however, was assigned to level **2**, which was used for

transactions involved in lower-level real-time communication between a window and an application.

TRANSACTION:

atg-interaction

LEVEL: 8

ALIASES: create-app: (fork * app)
open-window: (open-window app uit uin)
describe-window: (describe-window app uin)
interact: (interact-choices app uit)

PATTERN: (create-app open-window describe-window (* interact))

TIMEOUT: 10000

ACTORS: app: ATG
uit: UIT
uin: UIN

TRANSACTION:

open-window

LEVEL: 3

ALIASES: req: (open-win-req client sched)
fork-req: (fork sched window)
ack: (o-win-req-ack window client)

PATTERN: (req fork-req ack)

TIMEOUT: 20

ACTORS: client: *
sched: UIT
window: UIN

TRANSACTION:

describe-window

LEVEL: 3

ALIASES: send: (do-frame client win)
reply: (win-commens win client)

PATTERN: (send reply)

TIMEOUT: 20

ACTORS: win: WIN
client: *

TRANSACTION:

interact-choices

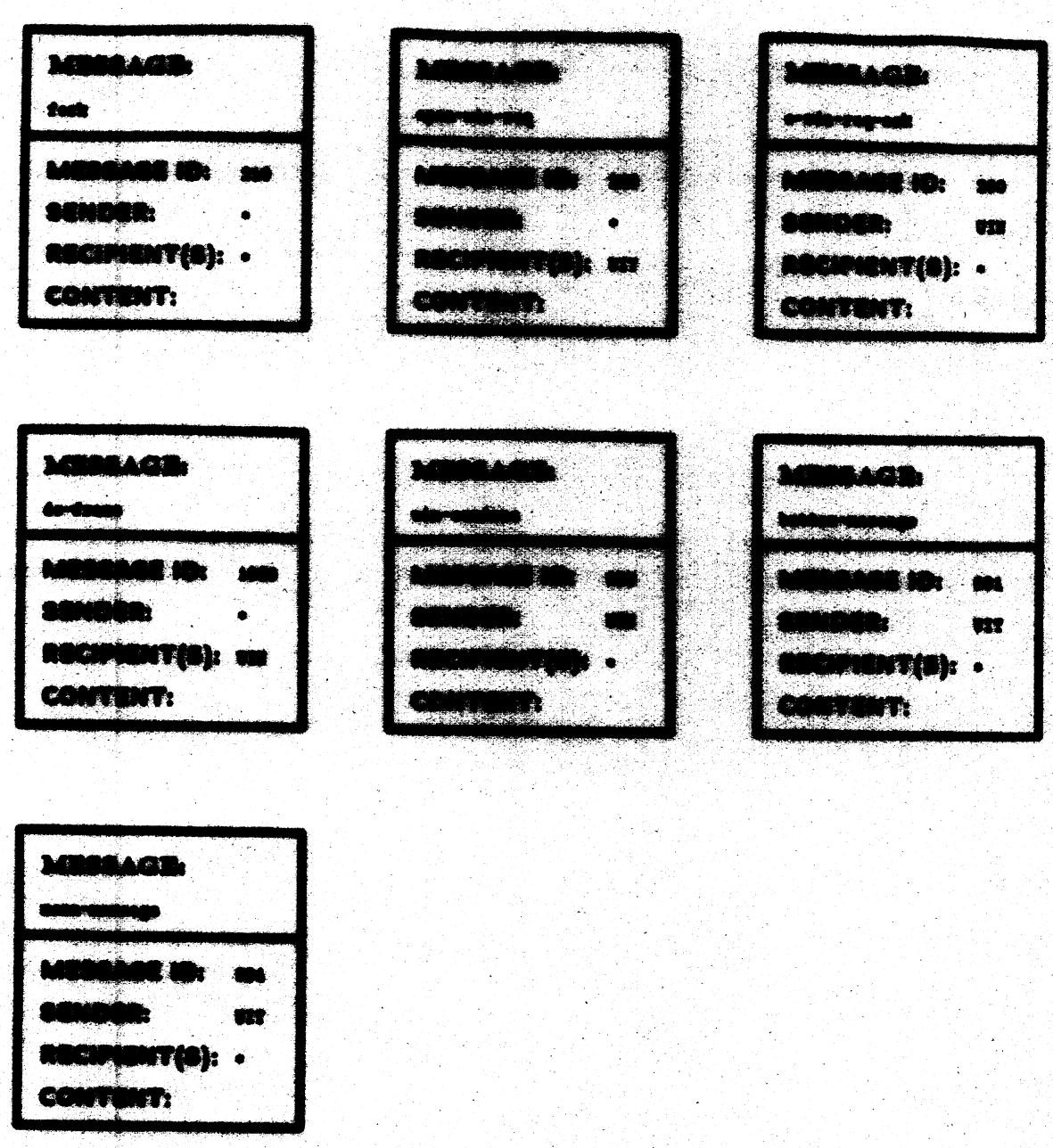
LEVEL: 2

ALIASES: button: (button-message wit client)
mous: (mous-message wit client)

PATTERN: ($or button mous)

TIMEOUT: 100

ACTORS: client: *
wit: WIT



Appendix C

A Message Journal

This list of messages represents a segment of a message journal generated by a program in the GenRad system. The frames represent the messages observed while the program was attempting a transaction of type **atg-interaction**, described in Appendix B. Two of the messages, numbers 2 and 4, are not part of the transaction, but are part of another transaction which was taking place simultaneously (note that the processes involved are distinct from those in the rest of the segment). With each message is listed its type, as described in Appendix B. This information is, of course, not included in the unprocessed journal, but is included here for readability.

Msg. #	Msg. ID	Sender	Recipient(s)	Time	(Type)
1	210	(MSG 99)	(ATG 2)	15	fork
2	1080	(TSD 22)	(UIN 666)	16	do-frame
3	200	(ATG 2)	(UIT 10)	17	open-win-req
4	996	(UIN 666)	(TSD 22)	18	uin-comdone
5	210	(UIT 10)	(UIN 11)	19	fork
6	200	(UIN 11)	(ATG 2)	21	open-win-req-ack
7	1080	(ATG 2)	(UIN 11)	25	do-frame
8	996	(UIN 22)	(ATG 2)	100	uin-comdone

Appendix D

The JDM Display

The following sequence of frames from the JDM display demonstrates a section of an actual debugging session, using the events defined in Appendix B, and the journal listed in Appendix C.

In the first frame (p. 114), the *abstraction level* is set to **5**, thus the only abstraction of interest is the transaction **atg-interaction**. The abstraction level selector allows choices for levels **0** (single message level), **3**, or **5**, since those are the levels of transactions discovered by the Analyzer. The level **2** transaction, **interact-choices**, was not found in the journal, and so level **2** is not one of the choices.

The *status window* indicates that the journal contains a faulty version of **atg-interaction**, starting at timestep 15, and reaching a "dead end" at timestep 21. It also notes that the transaction consists of two elements, a **fork** message (indicated by level 0) at time 15, and an **open-window** transaction at level 3, beginning at timestep 17. The map indicates that four processes are active in the **atg-interaction15** transaction: **MSG99**, **UIN11**, **ATG2**, and **UIT10**.

In the following frame (p. 115), the level has been switched to **3**, and the first level **3** transaction, **describe-window16**, is displayed. By noting the labels of the process nodes, the user can determine that this transaction does not involve the same processes as **atg-interaction15**, and is not of concern in this

debugging session.[27] In the third frame (p. 116) the first element of **atg-interaction15, open-window17** is displayed. Note that the screen still displays the representation for **describe-window16**, since that transaction has not ended yet.

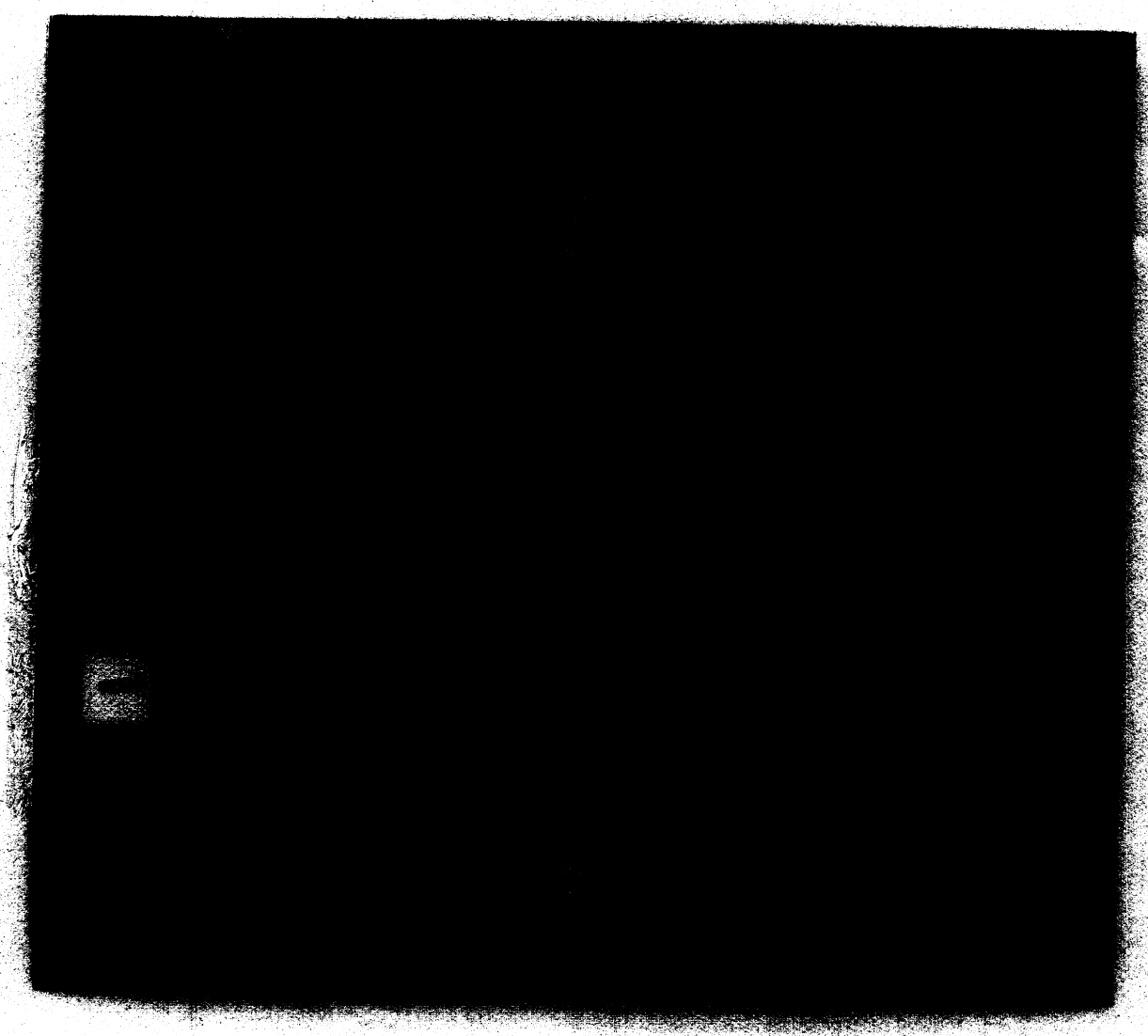
In frame four (p. 117), the cause of the "dead end" noted at level five in the first frame is revealed. The third element of **atg-interaction15, describe-window25**, is shown to be faulty. Specifically, a **do-frame** message was sent out at time 25, but no **uin-comdone** message was received in return. Thus process **ATG2** never receives a reply from process **UIN11.**

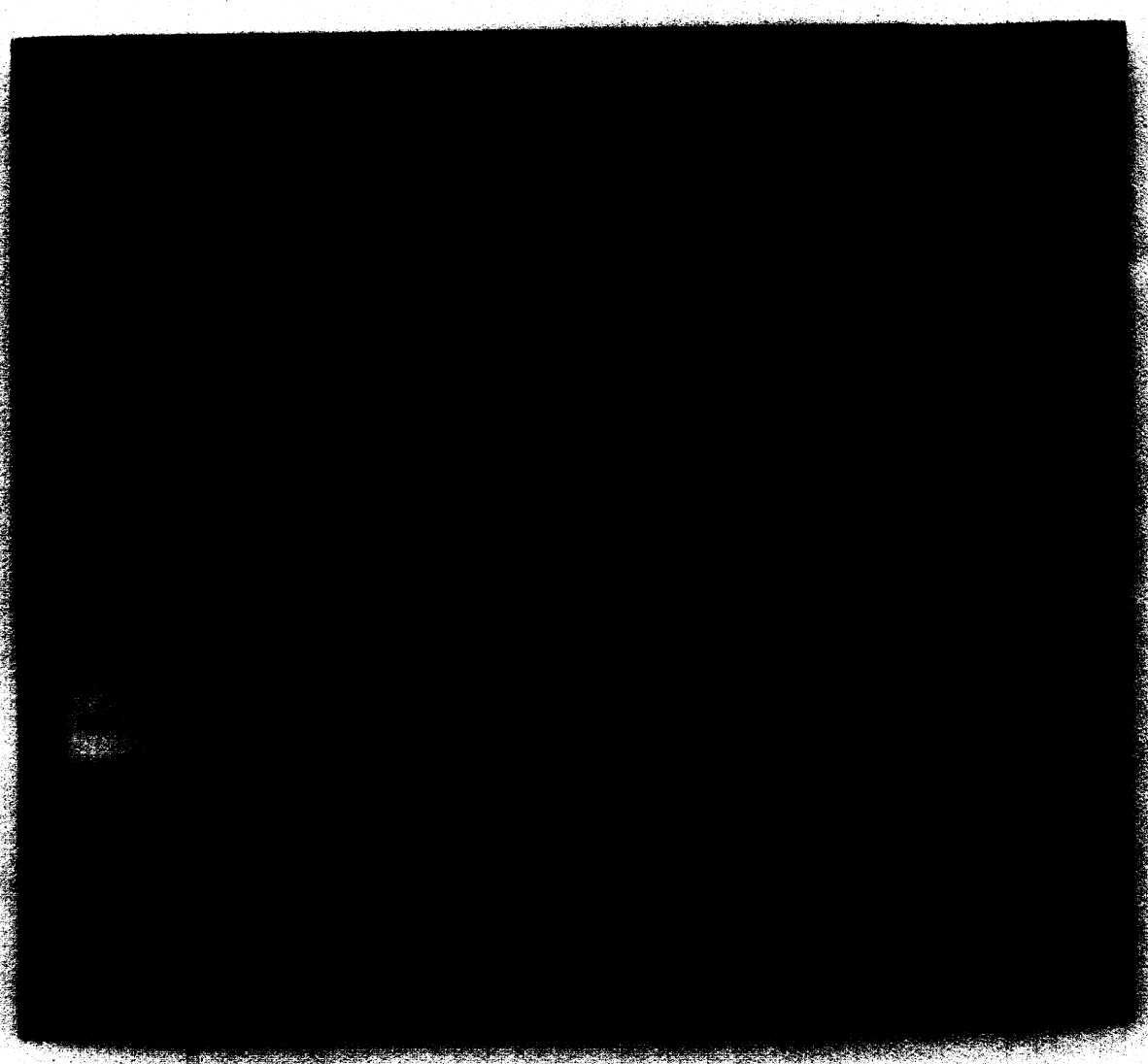
The JDM display screens up through frame four pinpoint for the user the source of the problem. In frame five (p. 118), a clue is given as to the cause. Frame five shows another faulty **describe-window** transaction, this one missing its first message, rather than its second. Significantly, the **uin-comdone** message is being sent to **ATG2**, which should have been waiting for a **uin-comdone** message from process **UIN11.** But the sender in this case is a new UIN process, **UIN22.**

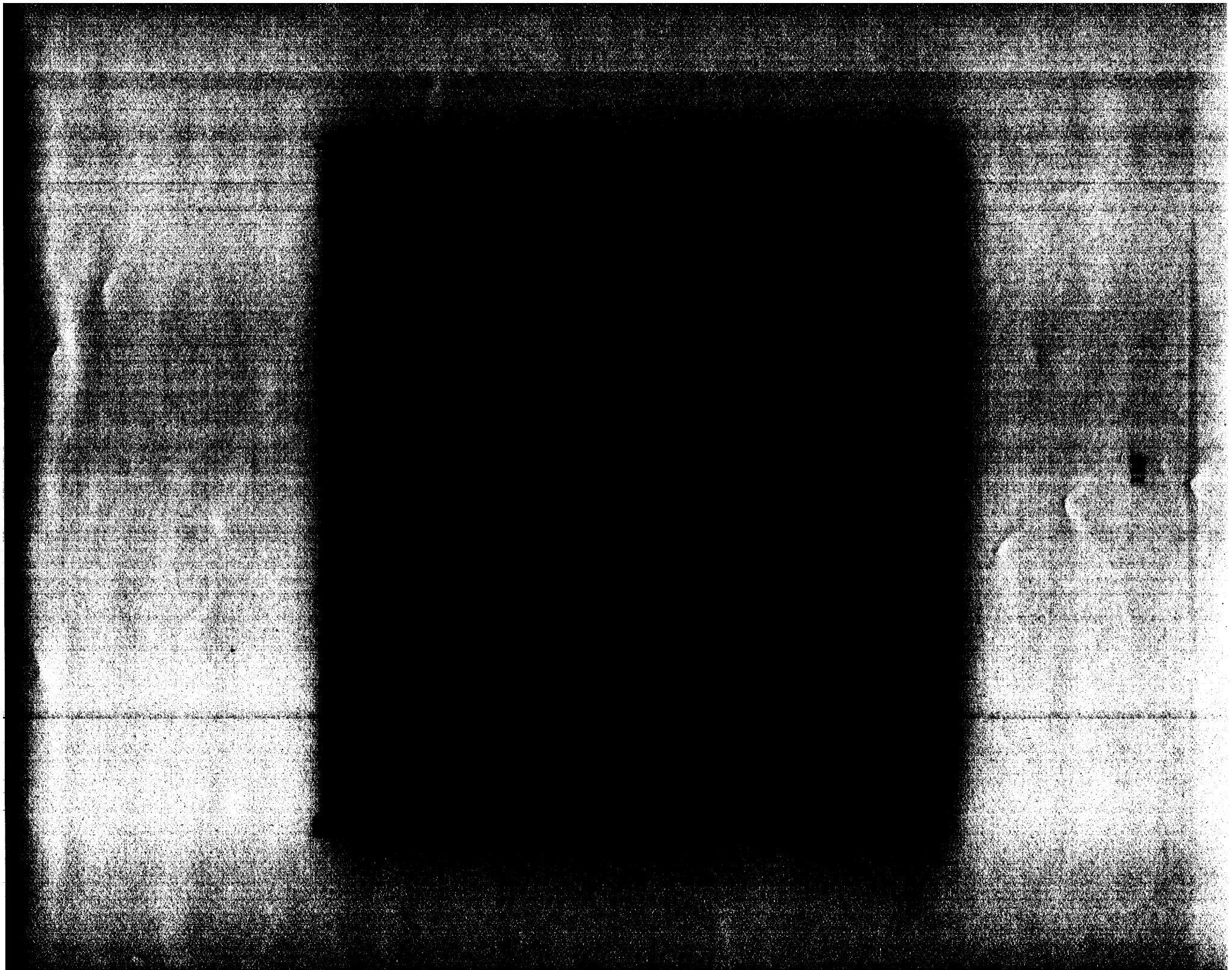
The display sequence described above eventually led to a discovery of the problem: in the process of creating a window for an application, the user interface task (UIT) created a special task, a *User Interface Node* (UIN), specifically to handle communication between the application task and the window being created. The UIN created in this case, due to a bug in the software, was exiting itself unilaterally. The UIT, seeing that the UIN task it had created had died, restarted it, *but with a new process id number.* As a result, this new UIN task continued to operate the window, but its messages to

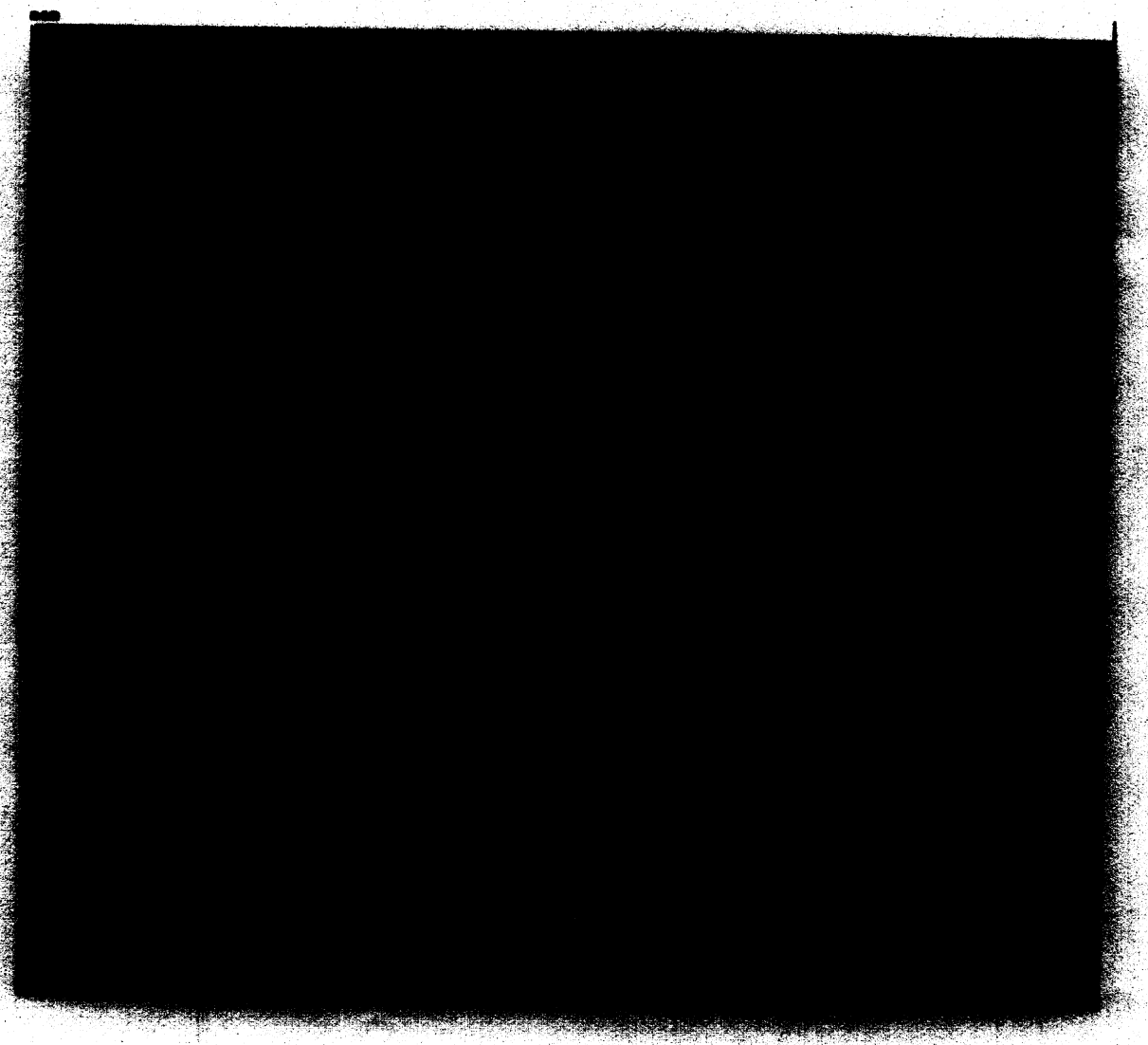
[27] This could also be determined by noting the elements listed under **atg-interaction15** in the status area, and noting that **describe-window16** is not among them.

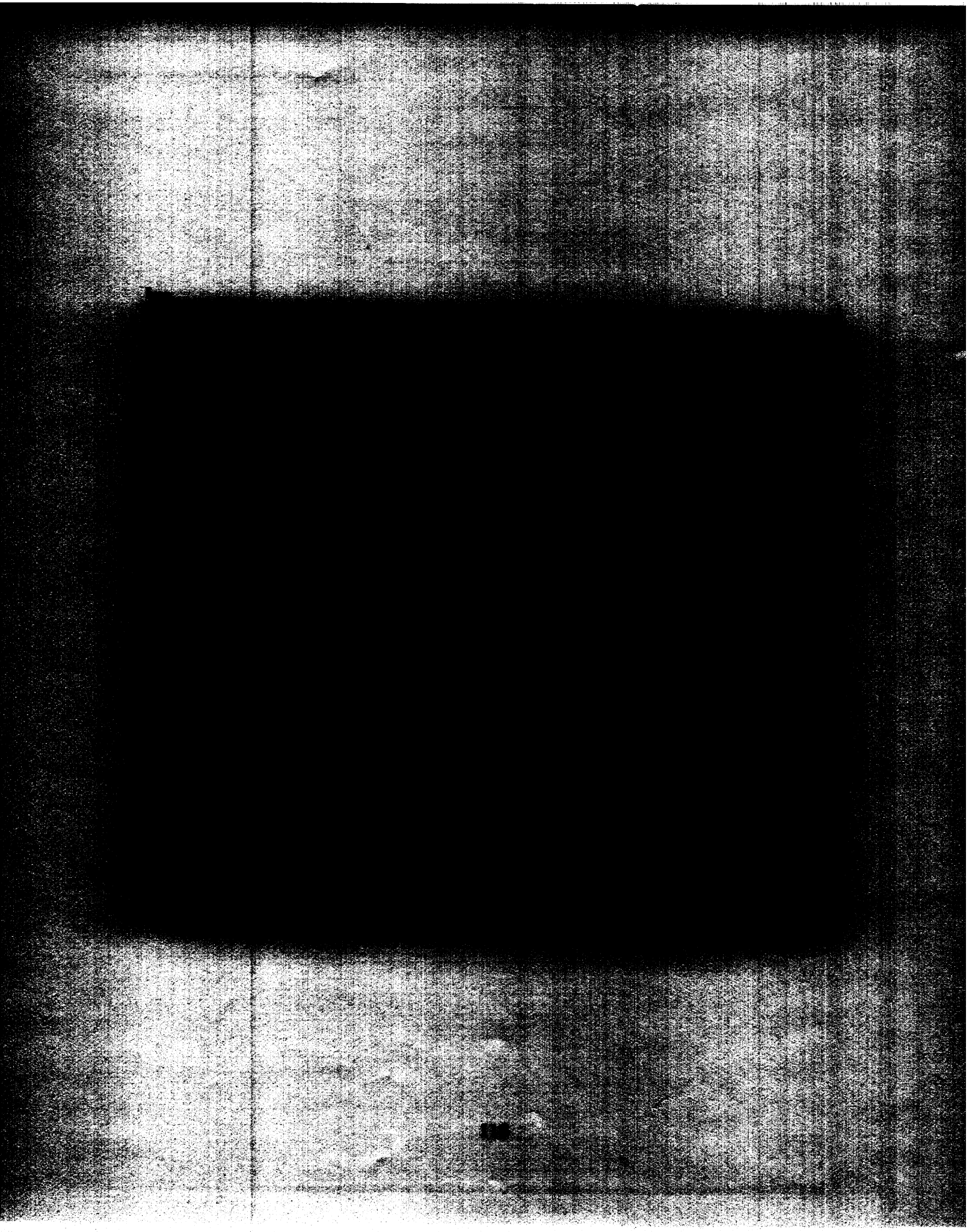
the application task were ignored, because it had the wrong process id number. From the user's point of view, this made it seem as though the ATG task was ignoring user input.











Appendix E

A BNF Grammar for MAL

What follows is a BNF grammar for the MAL expressions used in the MAM implementation. The expressions described are similar in syntax to lists in Scheme. They are designed as such in order to maximize ease of parsing by the Scheme-based Analyzer. They are not, however, easy to read, and the examples of MAL forms shown in Appendices A and B use a more readable representation.

The exact format of MAL expressions is relatively inconsequential, relative to understanding the language. What is important, and what is captured in this grammar, is the structure of the elements which make up MAL expressions.

```
Reserved symbols: ( ) $OR ? *

event_description          ::=  transaction_description
                                | message_description

transaction_description    ::=  (name level (actors)
                                (aliases) (pattern) timeout)

message_description        ::=  (name msgid (sender)
                                (recips) contents)

name                       ::=  string

level                      ::=  integer

actors                     ::=  (actor) | (actor) actors
```

```
actor                ::=  actor_name (actor_constraint)

actor_name           ::=  string

actor_constraint     ::=  * | process_type_list

process_type_list    ::=  process_type
                          | process_type process_type_list

process_type         ::=  string

aliases              ::=  (alias) | (alias) aliases

alias                ::=  alias_name (event_name arglist)

alias_name           ::=  string

event_name           ::=  string

arglist              ::=  actor_name | actor_name arglist

pattern              ::=  expression
                          | expression pattern

expression           ::=  alias_name
                          |(* pattern)
                          |(? pattern)
                          |(NOT pattern)

timeout              ::=  integer

msgid                ::=  integer
```

```
sender          ::=  process_type_list

recips          ::=  (process_type_list)
                     | (process_type_list) recips

contents        ::=  <undefined>

string          ::=  character | character name

integer         ::=  digit | digit integer

character       ::=  A | B | C | D | E | F | G
                     | H | I | J | K | L | M | N
                     | O | P | Q | R | S | T | U
                     | V | W | X | Y | Z | digit

digit           ::=  0 | 1 | 2 | 3 | 4 | 5 | 6
                     | 7 | 8 | 9
```

References

[Abelson85] Abelson, Harold and Gerald Sussman.
Structure and Interpretation of Computer Programs.
MIT Press, 1985.

[Arvind80] Arvind, Vinod Kathail, and Keshav Pingali.
A Dataflow Architecture with Tagged Tokens.
Technical Report MIT/LCS/TM-174, MIT Laboratory for Computer Science, Sept., 1980.

[Baiardi83] Baiardi, F., et. al.
Development of a Debugger For a Concurrent Language.
In *Proceedings of the ACM SIGSOFT/SIGPLAN Notices Software Engineering Symposium on High-Level Debugging.* ACM, March, 1983.

[Balzer69] Balzer, R. M.
EXDAMS - EXtendable Debugging and Monitoring System.
In *Proceedings of AFIPS Spring Joint Computer Conference.* AFIPS, 1969.

[Bates81] Bates, Peter, Victor Lesser, and Jack Wileden.
A Language to Support Debugging in Distributed Systems.
Technical Report TR-81-07, Dept. of Computer and Information Science, U. of Mass. Amherst., 1981.

[Bates82] Bates, Peter and Jack Wileden.
An Approach to High-Level Debugging of Distributed Systems.
Technical Report TR-82-35, Dept. of Computer and Information Science, U. of Mass. Amherst., December, 1982.

[Bates82a] Bates, Peter and Jack Wileden.
EDL: A Basis for Distributed System Debugging Tools.
In *Proceedings, 15th Hawaii Intl. Conf. on Systems Sciences.* 1982.

[Bates83] Bates, Peter, Victor Lesser, and Jack Wileden.
A Debugging Tool For Distributed Systems.
In *Proceedings, 1982 Phoenix Conf. on Computers and Communication.* 1983.

[Bates86] Bates, Peter C.
Debugging Programs in a Distributed System Environment.
Technical Report, Computer and Information Science Dept., University of Massachusetts, Amherst, January, 1986.

[CCA80] Computer Corp. of America.
A Distributed Database Management System for Command and Control Applications: Final Technical Report - Part 1.
Technical Report CCA-80-03, Computer Corp. of America, 1980.

[Cerf83] Cerf, V. and E. Cain.
The DOD Internet Architecture.
Computer Networks 7:307-318, October, 1983.

[DARPA81] Defense Advanced Research Projects Agency.
A History of the ARPAnet: The First Decade.
Technical Report AD A1 15440, Defense Tech. Info Center, April, 1981.

[Garcia-Molina84] Garcia-Molina, H., Frank Germano, Jr., and Walter H. Kohler.
Debugging a Distributed Computing System.
IEEE Trans. of Software Engineering SE-10(2), March, 1984.

[Gertner80] Gertner, Ilya.
Performance Evaluation of Communicating Processes.
Technical Report TR-76, Dept. of Computer Science, University of Rochester, May, 1980.

[Hillis81] Hillis, W. Daniel.
The Connection Machine.
Technical Report AIM-646, MIT, Sept., 1981.

[Lamport78] Lamport, Leslie.
Time, Clocks, and the Ordering of Events in a Distributed System.
CACM 21(7), July, 1978.

[Lewis81] Lewis, Harry and Christos Papadimitriou.
Elements of the Theory of Computation.
Prentice-Hall, 1981.

[Model79] Model, Mitchell.
Monitoring System Behavior in a Complex Computational Environment.
Technical Report CSL-79-1, Xerox PARC, January, 1979.

[Schiffenbauer81] Schiffenbauer, Robert.
Interactive Debugging in a Distributed Computational Environment.
Technical Report TR-264, Massachusetts Institute Technology LCS, September, 1981.

[Smith81] Smith, Edward S.
Debugging Techniques for Communicating, Loosely-Coupled Processes.
Technical Report TR100, Department of Computer Science, University of Rochester, December, 1981.

[Teitelbaum79] Teitelbaum, Tim.
The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS.
Technical Report TR 79-370, Department of Computer Science, Cornell University, June, 1979.

CS-TR Scanning Project
Document Control Form

Date : 4/26/96

Report # LCS-TR-392

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 124 (130 - IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): 52, 114-118

Other (note description/page number):

Description :	Page Number:
IMAGE MAP (1-124)	1 - 124
(125-130) SCANCONTROL, COVER, SPINE, TRGT'S (3)	

Scanning Agent Signoff:

Date Received: 4/26/96 Date Scanned: 4/26/96 Date Returned: 5/2/96

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned
Date: 04/26/1996
M.I.T. Libraries
Document Services