INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the original text directly from the copy submitted. Thus, some dissertation copies are in typewriter face, while others may be from a computer printer.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyrighted material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is available as one exposure on a standard 35 mm slide or as a 17" × 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6" × 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
Accessing the World's Information since 1938
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
Real-time monitoring of parallel and distributed systems

Ogle, David Mark, Ph.D.

The Ohio State University, 1988
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs or pages ___
2. Colored illustrations, paper or print ______
3. Photographs with dark background ____
4. Illustrations are poor copy ______
5. Pages with black marks, not original copy ______
6. Print shows through as there is text on both sides of page ____
7. Indistinct, broken or small print on several pages √
8. Print exceeds margin requirements ______
9. Tightly bound copy with print lost in spine ______
10. Computer printout pages with indistinct print ______
11. Page(s) _______ lacking when material received, and not available from school or author.
12. Page(s) ________ seem to be missing in numbering only as text follows.
13. Two pages numbered ______. Text follows.
14. Curling and wrinkled pages ______
15. Dissertation contains pages with print at a slant, filmed as received ______
16. Other ________________________________________________________________

______________________________________________________________

UMI
REAL-TIME MONITORING
OF PARALLEL AND
DISTRIBUTED SYSTEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

by

David Mark Ogle, B.S., M.S.

****

The Ohio State University
1988

Dissertation Committee:
Karsten Schwan
Douglas Kerr
Richard Parent
Richard Snodgrass

Approved by

Adviser
Department of Computer
and Information Science
To Christopher and Brian, my nephews
ACKNOWLEDGMENTS

First, I must thank my adviser, Karsten Schwan, who has patiently guided me through this entire process. He has provided guidance, both in the area of research and in the area of writing, while allowing me to maintain my individuality. For this, I will be forever grateful. I would also like to thank Dr. Richard Snodgrass, from the University of North Carolina for serving on my committee. His expertise and insight have contributed greatly to this thesis. I would like to thank Dr. Douglas Kerr and Dr. Richard Parent for serving on my committee and for suggesting improvements to my work.

Several other have contributed to this dissertation. Sridhar Vasudevan implemented the Adaptation Controller. Rajiv Ramnath is responsible for the database. Mayank (Micky) Choudhary implemented the monitor on the Encore multiprocessor. Yiannis Samiotakis and Kannan Varadhan worked on the RTDM project. John Gawkowski did the initial graphical interface implementation. Prabha Gopinath wrote most of Appendix B.

I wish to thank all the members of the Issos and PARTS research projects. Their work has greatly contributed to my dissertation. I am especially indebted to Rajiv Ramnath, Sridhar Vasudevan, Prabha Gopinath and Tom Bihari.

I owe a great debt to many others at Ohio State for providing both professional and moral support. Michael Kaelbling Ph.D., has provided invaluable assistance. Mike has diligently edited my dissertation and provided many insights on how to improve
my work. Ben Blake has been instrumental in motivating me to finish. Ben was free with his time, and was always willing to discuss my work and ways to improve it. I thank Prabha Gopinath and Peter Wiley for making the entire process more fun, and for keeping me going when things got tough.

There are many others who have provided encouragement, who have been there to bounce ideas off, and who have provided friendship throughout the years. I wish to thank, in no particular order: Win Bo, Ahmed Gheith, Anne Keunke, Mike Weintraub, Gregor Taulbee, Conleth Oconnell, Bill Punch, and Mike Fortin. I also need to thank two friends no longer at Ohio State, Kevin Culp and Chris Blaber, without whom I would not have been able to pursue my doctorate. I am especially indebted to Kevin for being crazy enough to go to the mountains with me.

I wish to thank the staff, especially Marty, Chris, Jane, Judy and Ernie for all their support. I also wish to thank Steve Romig, our system programmer, for providing support and assistance.

I wish to thank God for endowing me with the intellectual capacity to pursue an advanced degree, and for providing me with wonderful parents and family. Finally, I wish to thank my mother and father (LTC Charles R. Ogle US.A. (Ret), D.P.A., M.S. M.A.) without whom I could not have obtained my education. Their constant support has proved to be my most valuable asset.
VITA

14 August 1959  Born - Nurnberg, West Germany
1981           B.S. Computer and Information Science,
              The Ohio State University, Columbus, Ohio
1981-1984 Full-time graduate student: Graduate Teaching and Research
            Associate.
1984           M.S. Computer Science, The Ohio State University, Columbus,
              Ohio.
1984-present Full-time graduate student: Graduate Teaching and Research
             Associate.

Publications

Language and System for Parallel Programming", IEEE Transactions on

System for Parallel Programming", Proceedings of the 9th IEEE International

Collection and Analysis of Dynamic Information in a Distributed System", Tech
Report, OSU-CISRC-TR-85-12

Fields of Study

Major Field: Computer and Information Science

Studies in: Monitoring Distributed Systems, Operating Systems,
            with Dr. Karsten Schwan
            Database Systems, with Dr. D. Kerr,
            Heterogeneous Distributed Systems, with Dr. S. Mamrak
# TABLE OF CONTENTS

ACKNOWLEDGMENTS ................................................................. iii

VITA ......................................................................................... v

LIST OF TABLES ...................................................................... x

LIST OF FIGURES ................................................................... xi

LIST OF ABBREVIATIONS ...................................................... xiii

CHAPTER PAGE

| I. INTRODUCTION                                          | 1               |
| 1. Approaches to Monitoring                           | 2               |
| 2. Requirements of a Monitor                          | 4               |
| 1. RTDM Project                                       | 4               |
| 2. Issos Project                                      | 7               |
| 3. General Goals                                      | 11              |
| 4. Thesis Statement                                   | 13              |

| II. RELATED RESEARCH                                    | 14              |
| 1. Other Work                                          | 14              |
| 2. Our Contributions                                  | 21              |

| III. VIEWS OF DISTRIBUTED PROGRAMS                     | 23              |
| 1. Monitoring Specification                           | 23              |
| 1. Views                                              | 24              |
| 1. Attribute Definition - Conceptual Viewpoint        | 25              |
| 2. View Definitions                                   | 26              |
| 2. An Example from the Issos System                   | 27              |
| 3. Examples from the RTDM Project                     | 29              |
| 2. Conclusions                                        | 31              |

| IV. DISTRIBUTED COLLECTION MECHANISMS                  | 32              |
| 1. Overview of Monitoring System                      | 32              |
| 2. Collection Mechanisms                              | 34              |
| 3. Multiprocessor Specific Implementation             | 37              |
| 4. Compilation into Collection Mechanisms             | 39              |
| 1. Attributes versus Views                            | 40              |
| 2. Attribute Definitions                              | 41              |
3. View Definitions ................................................................. 42
   1. Correctness Values ......................................................... 43
   2. Monitor Actions ............................................................. 44
   3. Performance Values ....................................................... 45
5. Evaluation of the Method .................................................... 46
6. Conclusions ........................................................................ 50

V. ANALYSIS .............................................................................. 52
   1. Monitoring Cost Model ..................................................... 53
      1. Parallelization ............................................................. 57
      2. Perturbation versus Latency ......................................... 57
   2. Non-Quantifiable Monitoring Costs ................................. 58
   3. Sample Program ............................................................ 58
   4. Goals ................................................................................ 59
   5. Algorithm for Analysis Placement .................................... 62
   6. Analysis Using Extended Sensors ..................................... 65
   7. Information Flow ............................................................ 66
   8. Local Placement of Analysis ............................................ 68
   9. Further Optimization ...................................................... 70
10. Sampling versus Tracing .................................................... 71
11. Other Methods .................................................................... 72
12. Drawbacks to Our Approach ............................................. 74
   1. Incorrectly Reporting a View is Active ............................. 75
   2. Incorrectly Reporting a View is Not Active ..................... 75
13. Examples ............................................................................. 76
14. Validity of the Approach .................................................... 78
   1. Validity of Latency Model ............................................... 78
   2. Validity of Model Using Derivatives ................................. 81
   3. Validity of the Perturbation Model ................................. 82
15. Distributed Time ................................................................. 83
   1. Clock Synchronization .................................................... 83
   2. Time in Analysis in the Network Environment ................ 84
16. Temporal Issues .................................................................. 85
17. Predictive Capability ........................................................ 85
18. Conclusions ....................................................................... 85

VI. DATA STORAGE ..................................................................... 87
   1. Entity Relationship Model ............................................... 88
   2. Instantiations .................................................................. 90
   3. Monitor and Database Interaction .................................... 92
   4. Entity Creation ............................................................... 93
      1. Action Routines ......................................................... 97
      2. Database Operations .................................................. 98
   5. Temporal Issues ............................................................. 99
2. Measurements of Message Sends in the Network Environment .. 150
3. Conclusions ................................................................. 151
IX. CONCLUSIONS .......................................................... 152
  1. Independence ............................................................. 152
  2. Predictive Capability ................................................... 153
  3. Performance ............................................................. 154
  4. Tool Integration ........................................................ 155
  5. Extensions to Our Research ......................................... 156
A. ALOE GRAMMARS .................................................. 158
  1. Grammar for Attribute Definitions .............................. 158
  2. Grammar for View Definition ..................................... 165
B. DATABASE OPERATIONS .......................................... 175
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Results from Base Case</td>
<td>129</td>
</tr>
<tr>
<td>2. Results Using Central Analysis</td>
<td>131</td>
</tr>
<tr>
<td>3. Results Using Local Analysis</td>
<td>133</td>
</tr>
<tr>
<td>4. Results Using Sensor Analysis</td>
<td>134</td>
</tr>
<tr>
<td>5. Base Case: Two Sensors</td>
<td>138</td>
</tr>
<tr>
<td>6. Central Analysis: Two Sensors</td>
<td>139</td>
</tr>
<tr>
<td>7. Local Analysis: Two Sensors</td>
<td>140</td>
</tr>
<tr>
<td>8. Sensor and Local Analysis: Two Sensors</td>
<td>141</td>
</tr>
<tr>
<td>9. Latency Using Standard Flow</td>
<td>143</td>
</tr>
<tr>
<td>10. Latency Using Changed Flow</td>
<td>144</td>
</tr>
<tr>
<td>11. Encore: Base Case</td>
<td>147</td>
</tr>
<tr>
<td>12. Encore: Using Local Monitor</td>
<td>148</td>
</tr>
<tr>
<td>13. Encore: Using Sensor Analysis</td>
<td>149</td>
</tr>
<tr>
<td>14. Message Sending Costs</td>
<td>150</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Hardware Components of the RTDM Project</td>
<td>5</td>
</tr>
<tr>
<td>2. Components of the Prototype Adaptation Environment</td>
<td>8</td>
</tr>
<tr>
<td>3. Overview of the Distributed Monitoring System Mechanisms</td>
<td>33</td>
</tr>
<tr>
<td>4. Sample Network Sensor</td>
<td>35</td>
</tr>
<tr>
<td>5. Sample Multiprocessor Sensor</td>
<td>38</td>
</tr>
<tr>
<td>6. Attribute and View Compilation</td>
<td>39</td>
</tr>
<tr>
<td>7. Sample Probe</td>
<td>41</td>
</tr>
<tr>
<td>8. Regular Sensor</td>
<td>43</td>
</tr>
<tr>
<td>9. Extended Sensor</td>
<td>43</td>
</tr>
<tr>
<td>10. Overview of the Robotics Application</td>
<td>59</td>
</tr>
<tr>
<td>11. Overview of the Flow of Information</td>
<td>60</td>
</tr>
<tr>
<td>12. New Flow of Information</td>
<td>67</td>
</tr>
<tr>
<td>13. Analysis Placement Flowchart</td>
<td>73</td>
</tr>
<tr>
<td>14. Entity Definitions and Instantiations</td>
<td>91</td>
</tr>
<tr>
<td>15. Tool Interaction</td>
<td>92</td>
</tr>
<tr>
<td>16. Temporal Database</td>
<td>100</td>
</tr>
<tr>
<td>17. ALOE Screen</td>
<td>111</td>
</tr>
<tr>
<td>18. RTDM Display Screen1</td>
<td>119</td>
</tr>
<tr>
<td>19. RTDM Display Screen2</td>
<td>121</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS

ALOE .......... A Language Oriented Editor
ARL .......... Action Routine Language
BNF .......... Backus-Naur Formalism
COOL .......... Concurrent Object Oriented Language
DML .......... Data Manipulation Language
IDL .......... Interface Description Language
ISSOS .......... Industrial SystemS Operating Software
OS .......... Operating System
PCS .......... Program Construction System
RTDM .......... Real Time Distributed Monitoring
CHAPTER I
Introduction

In distributed and parallel environments, monitoring is performed to provide users with information about the programs they are running, and the underlying systems on which their programs run. Monitoring is performed for a variety of reasons including: debugging, replay, performance enhancement, and providing overall program information. In each case, monitoring involves the collection of information from the target (either a program or the underlying system), and the presentation of that information to the user.

Individual users may use the same monitored information in different ways. Consider, for example, a report that a machine has crashed. An operator will use this information to initiate reboot or repair procedures. A user, with a program to run, will use this same information to make sure that his program neither attempts to run on the 'down' machine nor relies on services from it.

Besides using the information differently, users often require the monitor to react differently to the same event. Consider the same example of a machine that crashes. The operator would like to be informed by the monitor as soon as the machine goes down, so he can take some immediate action. The user, on the other hand, is not interested in the information until he is ready to run his program. He can then query the monitor to find out the current status of the machine.
These simple examples demonstrate that individual users require different functionality from a monitor, even though the users may be working on the same set of machines.

I.1. Approaches to Monitoring

In this section we briefly discuss three approaches to monitoring distributed systems. Each approach has advantages and drawbacks associated with it. Consider a sample distributed sorting program that consists of a large number of processes communicating via messages. Typically, a user will be interested in determining where the bottlenecks are in such a program. One bottleneck might be the communication between specific processes. To determine if communication is a bottleneck, the user monitors the message traffic between particular processes. More specifically, the user may only wish to know when the communication between two processes, say P1 and P2, becomes unacceptably high. (The definition of what is unacceptable may vary from user to user.) There are many ways to provide users with the desired information.

One method is to monitor and display all message traffic information. Unfortunately, this provides the user with too much useless information. He will receive information concerning the communication between processes P4 and P5, when all he wants is information about the communication between processes P1 and P2. Besides displaying too much useless information, this approach also forces the monitor to collect large amounts of useless information, thus causing the monitor to perform less efficiently, and with increased latencies. The advantages of this approach are that the monitor and the user interface can both be simple, since the user is responsible for most of the analysis.
A second approach to monitoring is to collect all communication information about processes P1 and P2 and display this information to the user. The user would then be responsible for constantly checking to see if the communication level is acceptable. Problems arise using this approach. Assume that instead of just being interested in P1 and P2, the user is interested in the communication between P1 and P2, between P2 and P3 and between P4 and P5. In this approach the monitor would display all the communication information to the user, and the user would be forced to determine if any of the communication levels are unacceptable. It is possible that the user may not notice that the communication level is unacceptable until it is too late, since so much information is displayed. This problem increases as the number of processes increases. The advantages of this approach are that the user interface is simple and the monitor is still somewhat simple, though it is now responsible for doing more of the analysis.

A third approach to monitoring allows the user to shift the burden of the work to the monitor. This approach forces the user to more precisely specify what is to be monitored. In this approach, the monitor is responsible for both collecting and analyzing the information. Using the communication example, the monitor only reports to the user when the communication level reaches the specified threshold. By using this approach, the amount of information actually collected may be significantly smaller than with either of the other two approaches. The amount of information displayed to the user is also minimized. The drawback of this approach is that it limits the dynamic capabilities of the monitor. For instance, if all message traffic information is collected, as is the case in the first approach, the user has a complete history of what happened. He could, at any point in time, look at the communication history between any given processes. It is this third approach that we adopt. (Chapter V has a more detailed discussion of the problems associated with this approach, and proposed solutions to these problems.)
I.2. Requirements of a Monitor

To further demonstrate the different requirements that users place on monitors, we introduce two projects, the Real Time Distributed Monitor project (RTDM) and the Issos project, as typical distributed projects requiring a monitor. In the RTDM project, an operator is interested in monitoring a large distributed architecture consisting of hundreds of nodes, a main network, and multiple subnetworks. In the Issos project, a user wishes to improve the performance and reliability of his distributed program. The requirements that each of these projects place on a monitor differ and will be discussed in the subsequent sections. Using these two projects as examples, we will present a general approach to real-time monitoring of parallel and distributed environments.

I.2.1. RTDM Project

At The Ohio State University, the number of machines for which the Computer and Information Science department is responsible has increased significantly in recent years. As recently as four years ago the department was responsible for a single Vax 11/780 and two Sun-2 workstations. Today the department is responsible for maintaining several hundred machines, including over 120 Sun 3-workstations and multiple Pyramid 98x and 98xe machines.

The goal of the RTDM project is to provide tools that provide information to operators and system programmers. The architecture of the system is shown in Figure 1. An example of the type of information that is needed by a user in the RTDM
environment is a list of all machines that are currently 'down'. This information might be used by an operator to determine which machines need service. In smaller systems, it might be feasible for an operator to login to each machine and determine its status. When the number of machines increases, it becomes more difficult for a single person to manually collect all the information needed.

![Diagram of network layout](image)

**Figure 1: Hardware Components of the RTDM Project**

In the RTDM Project, there are three general categories of information that users need: emergency information, current information, and statistical and historical information.

*Emergency information* needs to be relayed to the user as soon as possible, so that the user may take some action on that information. An example of this would be the information that a machine has crashed.
Statistical and historical information is collected over time. For example, in the RTDM environment, each subnet is served by one or more local disk servers. The architecture is designed so that most of the requests a user in a particular subnet makes for files will be served by the local diskserver. Statistics are needed on the number of messages for file retrieval that are sent out of a particular subnet. This information can be used by the system manager to verify whether the file configuration and distribution is proper.

The final category of information is current information. An example is the current load on a particular processor or a particular subnet. An operator, getting ready to run a backup of a diskserver, needs to know the load on that diskserver. If the load is too high he may decide to postpone the backup until a later time. Current information is also useful to common users. For instance, if a programmer is looking for a lightly loaded machine on which to login, he can request the status of all the machines, or of a particular machine, to determine where best to login.

In order for the monitor to provide all the functionality required by the RTDM users, the monitor must do the following:

1. The monitor must be able to collect specified information about individual nodes in the system, about combinations of nodes, and about the networks that connect nodes. The monitor must also be able to analyze this information to see if it meets the criteria specified by the user.

2. The monitor must have a way to notify the user when an emergency condition occurs. Emergency information requires immediate action. Simply putting the information in a database or printing it to a system log is inadequate.
3. The monitor needs to be distributed throughout the network. A part of the monitor needs to reside on each node in the system in order to collect and analyze information about that node. Since analysis may involve information from multiple nodes, there must also be a central monitor that can correlate information from each of the nodes.

4. The monitor must have storage capability in order to store the statistical and historical information for later retrieval.

5. The monitor needs a convenient interface that can display the information collected to the user and that allows the user to control what information is being collected.

6. The monitor should cause little perturbation to the environment being monitored.

7. Finally, the monitor must be reconfigurable. For instance, if a particular node in a subnet goes down, information about that subnet should still be accessible, even if the node which crashes is the node responsible for collecting subnet information.

These are the requirements placed on the monitor by the RTDM project. We now introduce the Issos project, and the requirements it places on the monitor.

I.2.2. Issos Project

The goal of the Issos project is to provide users with tools for parallel programming [35]. For many applications, the exploitation of parallelism can lead to substantial improvements in runtime performance and reliability. Realizing those improvements in practice typically requires extensive prototyping, experimentation with, and evaluation of, such prototypes and redesign and reimplementation, termed
program tuning. We refer to tunings performed for the purpose of performance enhancements as adaptations [35]. Adaptations can be done statically (before the program begins executing) or dynamically (while programs are running).

The Issos project involves the construction and implementation of a large prototypical environment used to perform static and dynamic adaptation to distributed and parallel programs. The function of each part as it relates to the monitoring system is described briefly below (see [33] for a more extensive description of this environment). In Figure 1, the lines between the modules indicate interactions between them:

![Components of the Prototype Adaptation Environment](image)

- Program construction system (PCS) is used for program entry, editing, compilation, and initiation of linking and loading. It also provides language constructs for the specification of the adaptations to be performed for each program compilation and run.
• **Adaptation controller** (AC) performs the specified adaptations. It requests information from the monitor in order to perform adaptations.

• **Loader/operating system** (OS) is responsible for distributed loading, linking, and program startup and execution. It is also responsible for making available to the monitor and AC certain information regarding the distributed program, such as its process identifiers and the addresses of monitored variables.

• **Monitoring system** is responsible for collecting, analyzing, and making available the program information required by the AC.

Adaptations can be performed on distributed programs or on the underlying system on which the programs run. The underlying system consists of the operating system, the network protocols, the processors, etc. This will be referred to as the **multicomputer system** in this dissertation. For our purposes, only software adaptations will be discussed. Hardware and firmware adaptations are left for future research.

Adaptations can be performed by users or by programs. The optimal algorithms, heuristics or rules of thumb used in the adaptation process decide when to perform an adaptation and which adaptation to perform. Both of these decisions require the availability of information about the programs and the multicomputer system. Static adaptations require historical information about past runs of the program. Additionally, they may require some information about the current state of the system. In contrast, dynamic adaptations require current information about the program and the system that must be supplied while the program is still executing. Late or invalid information may result in useless, invalid or unnecessary adaptations. This means that the monitor must be able to quickly provide information about the program and the system, with minimum latency, *while* the program is still running and there is time left to perform an adaptation.
In order to provide the information necessary to perform adaptations, a monitoring system is needed that can monitor all aspects of a multicomputer system and of the programs that run on it. The monitor must provide a way to:

1. specify what information is needed,
2. specify when the information is needed and how correct and current the information needs to be,
3. provide methods of collecting the information based on this specification,
4. provide a method of storing the information collected, and
5. finally, it must provide a method of presenting the information to the performer of the adaptations.

The Issos system places many of the same requirements on the monitor that the RTDM project placed on its monitor. The requirements in common are:

1. The monitor must be able to collect information about individual pieces of a program or combinations of pieces.
2. The monitor must be capable of notifying the user when emergency conditions occur.
3. The monitor must be distributed in nature, since the target applications are distributed.
4. The monitor must provide some storage capability for historical and statistical information collected.
5. Since the purpose of adaptations is to increase the performance of the monitored programs, the monitor must run with little perturbation.
6. The monitor must be reconfigurable.

In contrast to the RTDM project, the Issos project monitor exhibits the following additional requirements, due in part to the somewhat stringent performance requirements:
1. The distributed nature of the programs being monitored introduces errors into the results returned by the monitor. These errors are due to the lack of a centralized timing mechanism, as well as the latencies associated with transmitting data from one node to another. Since the user is interested in performing adaptations he must be able to specify correctness criteria to the monitor. Thus, the monitor must provide a mechanism that allows the user to specify how correct results must be.

2. The distributed nature of the monitor also introduces latencies between when something occurs and when it is reported to the user. The monitor must provide a mechanism for the specification of the maximum acceptable latency between when an event occurs and when the adaptation controller is notified the event occurred.

3. The monitor must be capable of changing what is being monitored. For instance, a user may be performing an adaptation that increases the size of a queue, in which case the queue’s size might be monitored. After the adaptation is performed, the user may no longer be interested in monitoring the queue size.

I.3. General Goals

The general goal of this research is to develop an efficient, predictive, and independent real-time monitor for distributed systems.

By *predictive*, we mean that the monitor is capable of determining whether monitoring is possible, given the correctness and performance constraints specified by the user. We will show that it is possible, under certain conditions, to make the monitor predictive, or as predictive as possible. To achieve this, we first capture the
costs of distributed monitoring by developing cost models for perturbation and latency. We then validate the cost models empirically. Furthermore, we show how to statically estimate the correctness of the information collected.

With respect to independence, we will show that monitoring can be machine, operating system\(^1\), language, and application independent. This independence is possible because monitoring specification is achieved by having users define monitoring views of their programs. These views specify what should be monitored, the correctness of the information being monitored, the performance constraints placed on the monitor, and can specify the graphical display characteristics of the information. We will show it is possible to automatically generate the application dependent portions of the monitor based on these view definition. This increases the usability of the monitoring system.

In this dissertation we will show that distributed collection and analysis mechanisms must be used to make a distributed monitor efficient. We will demonstrate the feasibility and usefulness of such mechanisms and show that the distribution of collection and analysis mechanisms minimizes perturbation and latency of monitoring. For this, the monitor is shown to require access to program information.

We offer suggestions on what features are necessary to provide a user interface for a real-time distributed monitor, though this is not a major focus of this research. We will show it is possible and desirable to separate the storage of graphical information from the implementation of graphical display routines. Finally we discuss tool integration. We show that it is feasible and practical to integrate tools using an entity-relationship database while preserving the real-time characteristics of the

---

\(^1\)We assume some minimal operating system support, e.g., a communication mechanism.
monitor. We will also show, using examples, that the entity-relationship model supports the notion of views and allows views to be shared by multiple tools.

I.4. Thesis Statement

This thesis presents an approach to monitoring in distributed and parallel environments that is efficient, predictive, and independent, yet allows the monitor to be tailored to meet specific needs of individual users. It introduces newly developed and implemented collection and analysis mechanisms that can monitor such environments. The thesis also presents a user interface, including a monitoring language, that allows users to specify monitoring views. These views are automatically compiled into the collection and analysis mechanisms.
We begin this chapter by discussing research relevant to our own work. Where necessary, we explain how our approach to the problem of monitoring differs from research conducted elsewhere. At the end of the chapter we summarize the differences between our work and other work in this field, and explain the contributions to the field our work offers.

Before beginning we remind the reader that the goals of our research can be found in Section I.3.

II.1. Other Work

Much of the early work in the field of monitoring dealt with profiling [8, 13, 31, 46, 47]. Profiling involves the timing of procedure or statement execution. Abrams and Nutt [1, 26] introduced multiprocessor monitoring systems built upon the idea of profiling but slanted toward performance evaluation. Our work, as presented in this thesis, differs from approaches that only allow profiling. We support the monitoring of more aspects of a program than just the timing procedure or statement execution, as will be shown throughout this chapter.
The basic framework for our research in the field of monitoring comes from Snodgrass [41]. In his work, Snodgrass introduces the relational model as the appropriate model for information generated in a distributed system. Since information varies over time, he introduces temporal database operators. These operators provide the ability to state queries involving time. He also introduces the concept of sensors, probes, resident monitors and the central monitor as the mechanisms necessary to monitor distributed systems. His work focuses on the database query model as the interface between the user and the monitor. Our work builds upon these initial concepts but varies in several important ways. Snodgrass' work dealt with the collection of information from a multiprocessor target machine connected to a single front end computer running a monitoring system. Collection in his system is facilitated by changes to the operating system. Analysis makes use of interpretive update networks running on the dedicated front end machine. Because of this, they are inherently slow [40]. Our work concerns multiprocessors and large distributed computer networks, where kernel changes cannot be assumed possible. In such environments, analysis cannot be assumed to occur on a separate front end machine. In addition, in our system the information collected is used to perform real-time adaptations, thus the monitor must perform efficiently. Our analysis mechanisms are compiled into the resident and central monitors, and are often compiled directly into the programs being monitored.

Snodgrass' model builds on the relational model. We require that this model be extended to an object based real-time model, in order to facilitate integration of the monitoring system with other tools in the programming system. Snodgrass, in his temporal queries, does not consider specification of performance or correctness criteria, both of which are essential to real-time distributed monitoring.
Much of the work done in distributed monitoring lies in the field of distributed debugging, where users are interested in collecting information about distributed programs while they are running, in an attempt to debug the programs. We mention some of this work now and explain how it differs from ours. Before beginning we point out that debugging is really a two step process: monitoring and debugging. Because of this, we treat a debugger as another tool that can be integrated with the monitor, much like the adaptation controller, and we concentrate our debugging discussion on information collection rather than on the debugging process itself.

Some of the most innovative work in debugging was done by Bates and Wileden [2]. In their work they define debugging events. There are two categories of events: primitive events and high-level events. Sample events are: "read file", "write file", and "begin execution". In Bates' latest work [3], done in parallel to our work and the work of others [40], filtering of information is added for purposes of reducing the amount of information sent from node to node. Like other research, information is filtered after it has been generated by a process. In addition, we perform filtering before information is generated, i.e., during the collection process. Toward this end we define program view abstractions, similar to Bates' events, that allow for more generalized specifications of what to monitor. Our monitoring specifications may involve single variables or complex combinations of variables. While Bates is now adding display routines to his debugger he does not consider allowing display specification information associated with event description. We believe display specification should be part of view specification, since it allows users to specify all pertinent view information at one time. As we will show in a later chapter, including view display information with the view definition allows for independence of view

\[2\] Most recently developed debuggers have some sort of display routines because of the availability of bit-mapped display processors.
display specification and view display implementation. As with most debuggers, speed, perturbation and latency are not considered important problems, so most systems do not allow for the specification of performance and correctness constraints when defining events (views, in our terminology).

Many of the other approaches to debugging are less sophisticated than Bates and Wileden. Designs like IDD [12] perform debugging by collecting information about message traffic. In such approaches communication is considered the only interesting aspect of distributed or parallel systems. In IDD, all communications between processes are processed by a single supervisor process, making it a bottleneck. Filtering is done as the final step. IDD supports a time logic, similar to Snodgrass’ temporal query language [42]. We do not support such a notion of time. In a subsequent chapter we offer suggestions as to how our monitor could be extended to include temporal logic. IDD has some graphical capabilities, but again it does not support the concept of allowing users to simultaneously specify what is to be monitored, what actions to take with the collected information, and if/how the collected information is to be displayed.

Another debugging system is Bugnet [7]. In this system distributed checkpoints are performed every '30' seconds. There are many differences between our approach and theirs (or any checkpoint oriented monitor). The two biggest differences are speed and granularity. Their system is not concerned with how easily the approach can be scaled up to large diskless workstation environments. For instance, it is estimated that a checkpoint can generate up to 1 megabyte of data per machine. If the network contains 15 machines, then 15 megabytes of information get generated every 30 seconds. It is easy to see that this approach would quickly generate gigabytes of information in a large distributed environment. Bugnet is also not interested in capturing what occurs during the 30 seconds of execution. For instance, a variable can
change hundreds of times during that period but the only record kept is its value at the end of the 30 second period. A specification language, telling what to monitor, is also not provided.

Another interesting research project in the area of monitoring and debugging is being performed by Miller [24, 23]. Like many other approaches to debugging, he traces message traffic and uses breakpoints. As we have mentioned, using breakpoints is infeasible in real-time monitoring. What is unique about Miller’s work is his desire to use flowback analysis and high level responses. He is interested in developing a system where users can ask questions (similar to Snodgrass’ queries) and receive high level answers. For instance, the user could ask "why is variable X equal to 7". He wants to build a system that responds to this query by saying "variable X is 7 because variable "A" is 4 and variable "B" is 3 and "X = A + B". A major difference between his approach to the problem and ours is that he wants to do "after the fact" queries. We, on the other hand, want to do "before the fact" specification in an attempt to tailor the monitor’s implementation, and thus its performance and correctness. In our approach, a question such as Miller’s "why is variable X equal to 7", would be defined as a view asking the monitor to inform the user when "the value of X reaches 7". This is necessary since real-time flowback analysis is impractical due to the amount of information that must be collected.

Since we mention views, it is important to discuss Linton’s research which attempts to use a relational database to represent program information [20]. Linton defines static views of programs based on symbol tables. Our views cannot be constrained to contain only static program information. Linton’s static view approach limits the number of interesting views that can be defined. If a parallel program consists of multiple processes, Linton’s system does not allow view definitions built
concerning the number of times two pieces of the program invoke each others’
operations. Linton, as does Snodgrass, stores views in a standard relational database.
We cannot do this because of the real-time performance constraints place on the monitor.

Linton also introduces the concept of *pictographs*. A pictograph is a static
description of how a view should be displayed. Our work regarding graphical display
also allows for graphical specification by the user when defining views. Linton’s
more current work concerns program replay [21]. In this work he is concerned with
collecting information to be used in replaying a program. Collection of information is
achieved through the use of periodic checkpointing and by tracing messages.

Other current research in the area of program replay and reverse execution is
done by Mellor-Crummey and LeBlanc [19]. Theirs is a decentralized approach to
monitoring, where each node in the system collects information. Their approach
focuses on efficiently collecting information about message traffic and using that
information in the re-execution of a program. They believe, as do we, that only a
minimum amount of information should be collected, otherwise the performance of
the monitor is unacceptable. We allow a finer grain of specification, allowing for
collection of information concerning the internal execution of a part of a distributed
program. We consider a program replay program to be another example of a tool that
needs monitoring information. Mellor-Crummey and LeBlanc also have an interesting
temporal display notion, where things are displayed over time. They display all
information, rather than displaying particular views of information. This approach
does not work when doing real-time monitoring, since too much information must
displayed to the user. We choose to use views as the unit of display, allowing the user
to choose which views are displayed and how they are displayed.
In the area of monitoring, one of the more recent works is by Joyce et al. [14] in the context of the JADE research project [43]. It focuses on message traffic as the sole interesting aspect of distributed programs. In JADE, all communication is achieved using JIPC (JADE Inter-Process Communication) constructs. Joyce’s approach is strongly tied to JIPC since the monitor must have access to all communications. Furthermore, Joyce allows no user specification of what to monitor. Joyce does provide a graphical interface, but again it does not allow the users to specify how things should be displayed.

Other work in the area of monitoring includes the work performed by Plattner [28]. His most interesting contribution is the concept of allowing a monitoring statement with a predicate and an action. The actions are limited to incrementing counters, storing values and stopping execution. Our actions allow for the monitor to directly interact with other tools in the system. His predicates are defined on the state space of a single process, with no specification of predicates involving multiple processes. Collection is achieved by having the monitor listen to a bus or “pin”. This requires the monitor to have access to special hardware facilities, which is impossible for us since we desire hardware independence.

Other researchers, such as Parkinson [27], suggest the inclusion of hardware support for monitoring. Hardware support can result in an increase in the speed of the monitor. We do not address this issue in our research since we have no specialized monitoring hardware. It would be an interesting extension of our work to attempt to implement our distributed monitor using specialized hardware, to decrease the perturbation and latency associated with monitoring.
II.2. Our Contributions

We make contributions to the area of monitoring in distributed and parallel environments. First, unlike most work in the area, we treat monitoring as an integral part of the programming process. This allows us to more precisely tailor our monitor to meet desired performance criteria. We define a specification language with which the user can specify views of his program. A view includes information about what should be monitored, performance and correctness constraints, actions to be taken when the view is active, and how it is to be displayed. Our approach is application, machine and language independent.

We work in an object oriented programming environment, where objects have attributes. An attribute, by default, is the smallest monitorable portion of a program. It is important to note that many of the monitoring systems described here concern message traffic. While this is a very important aspect of distributed programs, it is not the only aspect. It is important to allow users to define monitorable views of portions of their programs and not just of the messages passed between them.

Our system allows us to place a cost, in terms of latency and perturbation, on monitoring. This gives the monitor the ability to be somewhat predictive, i.e., the monitor, under certain assumptions, can statically determine if it is possible to monitor the views specified by the user. Our approach allows the monitor to collect information using tracing and sampling. The monitor can automatically choose which method of collection is appropriate for each view definition. The monitor also automatically generates all the collection and analysis mechanisms based on view specification.
Our monitor provides information that is used in making real-time performance adaptations to distributed programs. We integrate our monitor with tools within the programming environment, in an attempt to allow the monitor to access program information. Thus the monitor shares information with other tools in the system, and is integrated through an entity relationship database. By using an entity relationship model for tool integration, tools can be added to the system or removed from the system without effecting the monitor.

In the rest of this thesis we will be discussing our monitoring system. We begin by introducing the concept of views, and presenting constructs necessary for view definition. We next show how these views are translated into collection mechanisms. Analysis mechanisms are then introduced, and we discuss how view definitions must be extended to include information necessary to determine what analysis mechanisms should be used. We introduce cost models then for perturbation and latency. We then discuss tool integration and introduce a database model, including an implementation, used to achieve tool integration. A discussion of the user interface follows this, including a discussion of the user interface built in conjunction with this research. We conclude the dissertation by presenting an evaluation of the monitoring approach and system introduced.
CHAPTER III
Views of Distributed Programs

This chapter discusses how users interact with the monitor to direct the collection of information. The goal of this chapter is to show it is possible to specify monitoring criteria in an application independent manner. Toward this end, views and a language to specify them are introduced. An attribute specification language is also presented in this chapter. Examples are presented that demonstrate the ease with which these languages can be used to define monitoring criteria in both the RTDM and the Issos projects. We begin this chapter by introducing the information model used when specifying attributes and views.

III.1. Monitoring Specification

The user directs the monitor's actions by specifying which aspects of his program he is interested in monitoring. To properly specify what to monitor, the user must have a model of the target program and the underlying multicomputer system on which it runs. For this discussion we will refer to both the target program and the underlying multicomputer system as the target application. We have chosen an extension of the entity relationship model [4] as our model because it allows us to describe both the target application and monitored information. In this model, each
program is described as a set of entities and relationships between entities [35, 29]. The mapping from a program to a set of entities and relationships is performed by the user (see [29] for a description of how the target program is described). Entities and relationships, defined by the user, may have attributes associated with them. Entities and relationships are typed. For example, when monitoring the message traffic in the sample distributed sort program, the program is described as entities of type process, with attribute Process_Id. These entities are related by the relationship Communicates_With. This relationship describes all message traffic within the program. Subsets of such information, such as all communication between processes P1 and P2, or P4 and P5, are defined using views.

III.1.1. Views

A view can be defined on either an entity or on a relationship between entities. Each view of an application program defines (1) which entities and attributes are involved, (2) when the view is active, (3) performance and correctness criteria of the view, (4) what action to take when the view is active, and (5) how the view is to be displayed. In this chapter we discuss the first two points of view definitions. In Chapter V we discuss points (3) and (4), since they concern how the monitor performs analysis. In Chapter VII we discuss point (5). Since a view is defined on attributes of entities or relationships, and not on the specific variables of a program, views are not dependent upon implementation details. Before discussing view definition, we will first explain how attributes are defined.
III.1.1.1. Attribute Definition - Conceptual Viewpoint

During the program construction phase, mentioned in Section I.2.2, the user defines his program set of entities or relationships. Again, determining which parts of the program are considered entities, and what relationships exist is left up to the individual user. As he is defining these entities he also defines the attributes that are associated with each entity. For example, when he is defining the entity of type process, he would associate the attribute Process_Id with that entity. The user may also define attributes with relationships. For instance, when the user is defining the relationship Communicates_With, he can associated the attribute number_of_messages with that relationship. These attributes can then be used in the definition of views. Below we introduce a language for defining attributes. We introduce this language so the user understands what attributes are, and how users define them. In the following grammar, reserved words are in capitals; user specified fields are in lower case. The grammar is as follows:

ATTRIBUTE DEFINITION FOR OBJECT objname
attribute_name : exp
END ATTRIBUTE DEFINITION

The user defined fields have the following meanings:

1. "objname" is the name of the object for which the attributes are being defined.
2. "attribute_name" assigns a name to an attribute.
3. "exp" is any valid C expression involving any variables defined in the object. (It is assumed that the user checks to see that the variables are defined before using them in an attribute definition.)

The terms object and entity are used interchangeably throughout this thesis. Users program in an object oriented environment. These objects are translated into entities in the monitor's model.
Using the example above, an attribute named Process_Id for object P1 would be defined as:

```
ENTITY DEFINITION FOR OBJECT P1
    Process_Id : my_procid
END ENTITY DEFINITION
```

This example assumes that the variable my_procid exists in the source code of the object. The language is general since a single statement of this kind may actually define multiple attributes within a single object.

### III.1.1.2. View Definitions

Views define the parts of a program that the user is interested in monitoring. These views are defined in terms of attributes of entities and relationships. Views contain information about when the view is active, the required correctness and performance constraints of the monitor, and view display information. Performance and correctness constraints are used to tailor the monitor to meet the needs of individual users, as we shall see in subsequent sections. (Again, in this section we are only concerning ourselves with a portion of the view definition, namely what entities and attributes are involved, and when the view is active.) Below we show the grammar for view definition:

```
VIEW DEF view_name
    exp
    RETURN ret_exp
END VIEW DEF
```

The definition of the user specified fields is below:

- "view_name" assigns a name to this particular view definition.
- "exp" refers to any valid C expression involving attributes. It determines when the view is active.
"Ret_exp" lists the attributes the user wishes returned. (The exact semantics of the RETURN statement are discussed in a later chapter.)

As an example, assume that a user wants a view involving a queue manager object. The view is active when the value of an attribute called Queuesize1 is greater than the value 24. The following view would be defined:

```
VIEW DEF Queue_limit_exceeded
   (QueueManager[1].Queuesize1 > 24)
   RETURN (QueueManager[1].Queuesize1)
END VIEW DEF
```

This is a specification of a view called Queue_limit_exceeded, which involves an object of type QueueManager, where QueueManager[1] is an instantiation of the QueueManager object. The view is active when the value of the attribute Queuesize1 exceeds the value 24. When the view becomes active, the RETURN statement indicates that the value of the attribute Queuesize1 should be returned.

To demonstrate the ease of attribute and view definitions, and to demonstrate the versatility of views, we present examples from both the Issos system and the RTDM project.

III.1.2. An Example from the Issos System

As we discussed in Chapter 1, the goal of the Issos System is to allow a user to perform adaptations to parallel and distributed programs in order to enhance performance and reliability. Programs in the Issos system are written in the object oriented language COOL [33].

We introduce the following sample application to familiarize the user with the Issos system. The sample application is a parallel quicksort application. The purpose of the application is to use multiple instantiations of the quicksort algorithm to sort a
large array of elements. Sample objects in this application include: an array manager object, a queue manager object and a quicksort object. The values to be sorted are stored in an array, which is managed by the array object. Indices of the array elements yet to be sorted are placed in the queue manager object. The actual sorting is done by the quicksort objects. An instantiation of the quicksort object removes a pair of indices from the queue, retrieves the corresponding values from the array and sorts those elements. When it is done, it places those values back into the array, and places two sets of indices in the queue. One sample adaptation might be to increase the number of quicksort objects as the number of indices in the queue grows.

All the objects in this example have attributes associated with them. Sample attributes are: Queuesize for the queue manager object, and number_of_elements for the array object.

With our model, the user can easily define what he is interested in monitoring. Assume the user is interested in the attribute "Queuesize". Specifically assume that he is interested in knowing when the number of index pairs in the queue exceeds 10. The user would first define the attribute for the queue object. This definition would be as follows:

```
ATTRIBUTE DEFINITION FOR OBJECT Queue_object
    Queuesize : queuesize
END ATTRIBUTE DEFINITION
```

This defines one attribute for the object Queue_object. The attribute is named Queuesize and is defined for the variable queuesize. Now the user can define the view he is interested in:

```
VIEW DEF Queuesize_exceeded
    (Queue_size.Queue_size > 10)
    RETURN (Queue_size.Queue_size)
END VIEW DEF
```

This defines the view Queuesize_exceeded, which is active when the attribute Queuesize of the Queue_size object exceeds the value 10. The value of the attribute Queuesize will be returned when the view becomes active.
This example shows how our model can be used to specify monitoring information that can be used in performing real-time adaptations. It also shows that specification is a two-phase process. First the user defines attributes of objects. In this phase the user must have knowledge about the internal structure of the target application. For instance, in the example above, the user must know that the `queue_size` is a variable in the code section of the object `Queue_size`. We consider this to be a reasonable assumption since the user conceptually defines attributes of objects while he is defining the objects themselves. Recall that our model of application software is object oriented. The second phase of specification is the specification of views. During this phase the user defines views in terms of attributes and not in terms of variables.

III.1.3. Examples from the RTDM Project

In the RTDM environment, a user is interested in monitoring a large loosely coupled computer system. In this environment it is assumed that the user is either interested in monitoring the kernel, the network, or programs such as `netstat` [44], that are designed to collect information about networks and nodes. In the RTDM project, these programs have been modified so that they periodically run and collect statistics.

In the RTDM project, the user sees everything as objects or relationships between objects. For example, the program `netstat` would be considered an object.

Assume the user is interested in knowing when the number of bad header checksums for a particular node, say `Seventh`, reaches an unacceptable level, e.g. 15. He would first define the attributes for the object `netstat`:

```
ATTRIBUTE DEFINITION FOR OBJECT Netstat_Seventh
    Bad_header_checksums : number_bad_header_chksums
END ATTRIBUTE DEFINITION
```
Next he would define the view he is interested in, namely a view that is active when the number of bad header checksums on the node Seventh is more than 15:

```
VIEW DEF Bad_header_chksums_Seventh
  (netstat_Seventh.Bad_header_chksums > 15)
  RETURN (Netstat_Seventh.Bad_header_chksums, "Seventh")
END VIEW DEF
```

This defines a view named `Bad_header_chksums_Seventh` which is active when the number of bad checksums on the node named Seventh exceeds the value 15. When this occurs, the attribute `Netstat_Seventh.Bad_header_chksums` and the character string `Seventh` should be returned.

**Parameterization.**

In many situations, users want to define the same attributes for multiple objects. For example, in the RTDM project the system programmers are going to be interested in the number of bad checksums on all the machines in the network. One method to do this would be to define "Bad_header_chksum" attribute for each of the machines. The problem with this approach is that since there are well over one hundred and fifty machines in the target environment, the user is forced to type in the attribute information for each of the machines. A method of parameterization is needed in the view specification grammar. We have built a macro capability into our parser for the language that allows the user to specify multiple objects in the `objname` field. A useful extension would be to allow a single view to be defined for multiple objects.

**Handling of Constants in Expressions.**

Often the user defines views involving constants. For example, in the `Bad_header_chksums_Seventh` view defined above, the user defines a view that is active when condition

```
(netstat_Seventh.Bad_header_chksums > 15)
```

is true. Frequently users change their minds and want to change the constants involved in such conditions. For instance the user may decide that instead of looking for values
greater than 15, he now wants to look for values greater than 50. For practical implementation reasons, we have designed the view compiler, which will be discussed in a subsequent chapter, to automatically change all references to constants into array references. This allows users to dynamically change the constant values being used in condition evaluation.

III.2. Conclusions

In this chapter we introduced the entity relationship model as the model used to describe both the target application and the monitored information about that application. Entities and relationships have attributes associated with them, and it is upon these attributes that views are defined. Since views are defined in terms of attributes of entities and relationships, the views themselves are not restricted by implementation details of a particular entity or relationship, thus making views application independent. In order to implement views, the target application must be described as entities and relationships, and attributes must be able to be associated with those entities and relationships. We have presented a language for defining attributes of entities and relationships. In subsequent chapters we demonstrate that it is possible to define most applications in terms of entities and relationships. This chapter also showed that attributes and views are easy to define, and are application, language, and machine independent.
CHAPTER IV
Distributed Collection Mechanisms

This chapter contains a discussion of the collection mechanisms used in our implementation. This chapter shows that distributed collection mechanisms must be used in a distributed environment to monitor efficiently. Further, this chapter demonstrates the versatility of our collection mechanisms by demonstrating the use of these mechanisms in both a network environment and in a multiprocessor environment. This chapter also demonstrates the ease of use our monitoring system by showing how attribute and view definitions are automatically compiled into the correct collection mechanisms.

IV.1. Overview of Monitoring System

The monitoring system described below is responsible for the collection and analysis of distributed program information. Its overall structure is shown in Figure 1, for a sample distributed network consisting of multiple Pyramids™, an Encore Multi-Max™ multiprocessor, and multiple SUN™ workstations all running Unix™ and connected by EtherNet™ networks and subnetworks.

Figure 3 depicts the flow of information from the user's program to a local monitor, called a resident monitor [41], running on each machine. Eventually the
Figure 3: Overview of the Distributed Monitoring System Mechanisms

information is given to the central monitor. The flow of the information and what happens to the information during that process is discussed in this chapter.
IV.2. Collection Mechanisms

Information collection can be performed in two modes [39]: by sampling or by tracing. *Tracing* consists of the reporting of all occurrences of an event within a certain time interval. Tracing is synchronous with the occurrence of an event; it is performed when all occurrences of an event must be known (e.g. when collecting complete 'history' information) or when a user wishes to keep track of all occurrences of an event. Keeping track of the number of messages in a message queue is an example of a traced event. It requires noting both when messages are inserted into the queue, and when messages are deleted from the queue. *Sampling*, on the other hand, is when some, but not all occurrences of an event must be known. Sampling is not necessarily synchronous with the occurrence of an event, and is useful when an immediate reaction to an event is not necessary. Continuing with the example of the message queue, sampling could be performed if the user wants to periodically check how many messages are currently in the queue.

*Sensors* [38, 39] are small pieces of code residing within the program being monitored. A sensor samples or traces the occurrence of a certain event at the request of a resident monitor. Sensors report information, such as current value and time, to the resident monitor. When to report the information is determined partially by the user during specification. If a sensor contains analysis code, it is called an *extended sensor*. These will be discussed in Chapter V. Sensors are generated automatically by the monitor based on a programmer's specifications of views. However, the placement or insertion of the generated sensors within the application code must be performed manually. The automation of sensor placement remains a research problem [41].
Assume the user is interested in using a sensor to trace the value of the attribute `bad_header_chksums`, defined in Section III.1.3. The sensor shown in Figure 4, shows what the network environment sensor for that attribute might look like. A multiprocessor sensor would use a shared memory call instead of a message send primitive.

```c
if (status[1] == 1)
{
    sensor_struct.command = NEW_VAL;
    sensor_struct.sensor_num = 1;
    sensor_struct.sensor_time = gettimeofday();
    sensor_struct.int_result = number_bad_header_chksums;
    sendto(monitor_socket_send, &sensor_struct,
           sizeof(sensor_struct), 0,
           &monitor_sin_send,
           sizeof(monitor_sin_send));
}
```

Figure 4: Sample Network Sensor

A traced sensor begins tracing when it is enabled by the resident monitor; it stops tracing only when it is disabled. The sensor tracing the value of the variable `number_bad_header_chksums` in Figure 4 generates a value everytime the variable is changed, provided that the sensor is enabled. The status of the sensor (i.e. enabled or disabled) is kept in the array `status`, which is checked before the sensor is fired. In this particular case, since this is the first sensor in the program, its status value is kept in the first position, `status[1]`.

The resident monitor is given the results of such a trace via event records generated by the sensor. Event records contain the following information:

1. A command identifier, telling the resident monitor that the information is from a sensor, as opposed to a command from the central monitor.
2. A sensor number, so the monitor knows which sensor is reporting.
3. The time at which this event was recorded.
4. A value to be returned. In this case it is the integer content `number_bad_header_chksums`.

Event records are communicated to the resident monitor by notification or by message. Communication by notification implies that the receipt of the record by the resident monitor is synchronous with the occurrence of the event. Communication by message implies that the event and the receipt of the event record are asynchronous, since the message may be queued for an unknown period of time. This is also possible in a shared memory environment where messages are stored in a shared queue and may remain queued for an undetermined amount of time. For example, when collecting history information regarding the values of the variable `number_bad_header_chksums`, event records can be received asynchronously (by message), since the resident monitor need not 'immediately' know about the occurrence of each change in the variable's value. However, if the resident monitor has to react immediately to the event that `number_bad_header_chksums` exceeds some threshold value, then it must be interrupted synchronously with the event (i.e. notified). In the network environment, the only method of sharing information is by message passing. To achieve notification in this environment, messages are sent to a special socket in the resident monitor, and the resident monitor is signalled via a standard Unix signal, at which time it reads from the special socket. It is worth noting that using Unix signals is not the most reliable method for doing notification since Unix signals can be lost or pre-empted.

A variable is said to be sampled if some, but not all, changes to it are monitored. A sampled sensor is a sensor used to monitor a sampled variable. For instance, assume the user is monitoring a variable which keeps track of the number of elements in a
queue, called queue_size. If the user is only interested in ensuring that the size of the queue never exceeds a threshold, he could sample the values whenever they were incremented, and ignore them when they were decremented. The sensor used to sample "queue_size" would be considered a sampled sensor. Event records from sampled sensors contain the same information as event records from traced sensors. They may also be communicated to the resident monitor by notification or message.

Probes are pieces of code residing within the resident monitor on each network node that directly access the address spaces of individual processes on that node, thereby providing a convenient mechanism for sampling. The main advantage of probes over sensors is that the application code need not be changed for probing, so events that are to be probed can be defined dynamically. However, for the Unix implementation of the monitoring mechanisms described in this thesis, a probe is implemented in two parts: (1) primitive probes inserted into the application program by the monitor, which can be activated via Unix ‘signal’ operations issued by (2) higher level probes located within a resident monitor. Primitive probes are automatically inserted into each application program based on programmer-stated specifications of the various attributes of program components. The tradeoffs in performance between probes and sensors is discussed in Section V.1.2.

IV.3. Multiprocessor Specific Implementation

Besides Suns and Pyramids, we also have an Encore Multimax multiprocessor at Ohio State. A few of the applications we run in the Issos project are built to run on the Encore. It is also important in the RTDM project for information to be made available about the Encore. A small prototype multiprocessor monitor was designed and built for the Encore. The prototype demonstrates the versatility of the general design of the
monitor and the independent nature of the monitoring approach. The prototype is also used to show how views can be translated into the implementation-specific portions of the monitor, specifically sensors and probes, regardless of whether the target hardware is a Sun workstation or an Encore multiprocessor.

The monitor on the Encore still makes use of sensors, probes, and a resident monitor. The difference is in the way communication is achieved. Probes and sensors are implemented using shared memory, instead of message passing. Figure 5 shows the multiprocessor equivalent to the network sensor shown in Figure 4.

```c
if (status[1] == 1)
{
    sensor_struct.command = NEW_VAL;
    sensor_struct.sensor_num = 1;
    sensor_struct.sensor_time = gettimeofday();
    sensor_struct.int_result = number_bad_header_chksums;
    enqueue(sensor_struct);
}
```

Figure 5: Sample Multiprocessor Sensor

A shared memory access, via the command enqueue, allows two or more processes to directly share a variable. The current implementation buffers the sensors structures between the sensor and the local monitor. The current buffer size is large enough, with 200 elements, so that the monitor has not lost values. A shared memory access on the Encore, using enqueue, takes approximately 800 μs, as opposed to the approximate 3-10 ms that are required by the message sends in the network environment. This implies that collection of information is quicker on a multiprocessor. Of key interest is whether shared memory environments are more conducive to central analysis mechanisms. We will show, in Chapter VIII, that even in shared memory multiprocessor environments, central analysis mechanisms are still the bottleneck.
IV.4. Compilation into Collection Mechanisms

This section concerns the 'compilation' of attribute and view definitions into distributed monitoring mechanisms. Again, automatic compilation is necessary to ensure the ease of use of the monitor. Figure 6 gives a general overview of how the monitor compiles attribute and view definitions into collection mechanisms.

![Diagram of Attribute and View Compilation]

The monitor is responsible for collecting and analyzing information in order to determine when a view is active. How it decides if a view is active can effect performance, both of the monitor and of the program it is monitoring. We implement
tracing with sensors, which are automatically generated by the monitor and inserted into the user's code manually. Sampling is achieved with probes, which reside partially in the user's code and partially in the resident monitor. Probes are automatically generated and inserted. Tradeoffs exist in the costs and correctness of the two modes of collection [39, 34]. Basically, probes result in less perturbation but also provide results that are less accurate than the results provided by sensors.

In the network environment, the main cost of collection is the transfer of monitoring information from its place of collection to its place of analysis. In the network environment, intranode messages require approximately 3.2 ms (Section VIII.2 contains a discussion of the actual costs of messages sends, and how those numbers were acquired.), whereas internode messages require anywhere from approximately 4.5-10 ms. The analysis of a message takes an average of a few hundred microseconds. These costs imply that the cost of analysis of monitoring information (determined by processing speed and amounts of memory available for storage of information to be monitored) is small compared to the cost of collection in a network environment (determined by message transfer times). The use of shared memory on individual nodes for information transfer (now implemented on an Encore multiprocessor) reduces the cost of transfer to a few hundred microseconds per item transferred.

IV.4.1. Attributes versus Views

Attributes and views define different monitorable aspects of the target application. Attribute definitions list all monitorable attributes of an object. For each attribute, sampling code is generated. Views, on the other hand, define those aspects of the system the user is most interested in seeing. Views are translated into a
combination of tracing and sampling code. This choice is due to the high costs of tracing.

IV.4.2. Attribute Definitions

Attribute definitions are translated into sampling code that allows the user to obtain the value of the attribute upon request. Consider the following attribute definition for the object QueueManager:

```
ATTRIBUTE DEFINITION FOR OBJECT QueueManager
  Queuesize : que_size
END ATTRIBUTE DEFINITION
```

This view definition translates into a probe which returns the current value of the attribute Queuesize, whenever the user requests it. The portion of the probe that would be generated and placed in the user's code, in a network application, is shown in Figure 7.

```
switch(probe_number) {
  case 2:
    result = queuesize;
    break;
}
```

```
sendto(monitor_socket_probe,&result,
  sizeof(Result),0,&monitor_sin_probe,
  sizeof(monitor_sin_probe));
```

Figure 7: Sample Probe

The probe shown in Figure 7 returns the value of the variable queuesize, when probe number 2 is fired. In the worst case, the cost of probing this view is four messages: (1) from the user's view manager to the central monitor, requesting that the view be updated, (2) from the central monitor to the appropriate resident monitor, requesting the value, (3) from the resident monitor to the appropriate process
requesting the value for the attribute Queue size, and (4) from the process being monitored to the view manager, returning the value. Thus, worst case probing costs are 3 internode messages and 1 intranode message. The total time for probing is approximately 33.5 milliseconds.

IV.4.3. View Definitions

Views are translated into tracing code. A problem with tracing is that everytime a value changes, a message is generated. Consider again the view defined above:

VIEW DEF Queue limit exceeded
   (QueueManager[1].Queuesize1 > 24)
   RETURN (QueueManager[1].Queuesize1 )
END VIEW DEF

If this view is traced then a sensor similar to the one shown in Figure 8 is generated. If this sensor is used, then a message is generated every time the value of the attribute Queuesize1 changes. If this value changes frequently, then using this sensor would adversely effect the performance of the monitor and of the program being monitored by generating numerous messages. One solution to this problem is to allow the desired analysis (i.e. is "Queuesize1 > 24") to occur in the sensor itself. The resulting extended sensor, shown in Figure 9, generates a message only when the value of Queuesize1 exceeds the value 24. Extended sensors, like the one shown here, can be used to attempt to make the monitor more efficient.

One of the basic goals of this research is to allow a user to tailor the performance of the monitor. Toward this end, view definitions can be extended to include information concerning the correctness of the information being monitored and the performance constraints required by the user. In the next sections we introduce constructs that can be added to view definitions that allow the user to specify performance and correctness constraints. This information will be used by the monitor to attempt to optimize the monitor's performance.
if(status[3] == 1)
{
    sensor_struct.command = NEW_VAL;
    sensor_struct.sensor_num = 3;
    sensor_struct.sensor_time = gettimeofday();
    sensor_struct.int_result = queuesize1;
    sendto(monitor_socket_send,&sensor_struct,
           sizeof(sensor_struct),0,
           &monitor_sin_send,
           sizeof(monitor_sin_send));
}

Figure 8: Regular Sensor

if(status[3] == 1)
if(queuesize1 > 24)
{
    sensor_struct.command = NEW_VAL;
    sensor_struct.sensor_num = 3;
    sensor_struct.sensor_time = gettimeofday();
    sensor_struct.int_result = queuesize1;
    sendto(monitor_socket_send,&sensor_struct,
           sizeof(sensor_struct),0,
           &monitor_sin_send,
           sizeof(monitor_sin_send));
}

Figure 9: Extended Sensor

IV.4.3.1. Correctness Values

Correctness values allow the user to specify how correct, with respect to time, the results must be. Chapter V explains why the correctness constraints are useful. The monitor uses this specification in an attempt to determine whether a certain view requires tracing or sampling. Consider another sample view:

VIEW DEF Both Queue limits exceeded
        (QueueManager[1].QueueSize1 > 24) AND
        (QueueManager[2].QueueSize2 > 24)
        CORRECT TO WITHIN (10 UNITS)
        RETURN (QueueManager[1].QueueSize1)
END VIEW DEF

This view is defined to be active when the value of QueueSize is greater than 24 in
both instantiations of QueueManager. We make use of a view’s correctness value to
determine whether to trace or sample the attributes involved. Here, the options are to
trace both queuesizes, or to trace one and probe the other. In the network
environment, deciding between these options is straightforward. (The time units in the
network environment are milliseconds.) One option for this view is to trace the value
in QueueManager[1] and probe for the value in QueueManager[2]. In this
case, once the value of QueueSize1 in QueueManager[1] exceeds 24, the
sensor sends a message to the monitor, which takes approximately 3.5 ms. The
monitor then probes the value of QueueSize2 in QueueManager[2]. If
QueueManager[2] resides on a different node, then probing takes approximately 3
messages, 1 intranode and 2 internode messages, or about 17 ms. Since 17 ms is larger
than the correctness value of 10 ms, we must trace both values. If on the other hand,
QueueManager[2] resides on the same node (assuming no shared memory), then
the cost of probing is 2 intranode messages, or approximately 7 ms. Since this is less
than the 10 ms specified by the user, we could probe one and trace the other. If shared
memory exists on a node, then the amount of time in this case needed to probe is 2
shared memory accesses. Again, tradeoffs exist between the correctness of the result
and the performance of the monitor. (See Section V.10.)

IV.4.3.2. Monitor Actions

The monitor described in this work is integrated with other tools. One such tool is
a view storage facility. As we will see in Chapter VI, the view storage facility
supports an entity relationship database. It is assumed that tools will be the users of
monitored information. For instance, in the Issos project, the adaptation controller will
be one user of monitored information. In the RTDM project, the graphical display tool
(to be discussed in Chapter VII) is a user of monitored information.
When a view becomes active, the monitor must decide what to do with the information that is in the RETURN statement. This is the purpose of the command portion of the view definition. The two commands supported are: RETRIEVE and NOTIFY.

RETRIEVE is the simpler of the two commands. Each view has a database entry associated with it. When a view becomes active, the RETRIEVE command tells the monitor to update the appropriate attributes of the view. This translates into an update of the database entry associated with this view (see Chapter VI for examples).

NOTIFY is similar in semantics to RETRIEVE. When the view becomes active, the database entry associated with the view definition is updated. NOTIFY also implies that the process specified by the user in the view definition be notified. In our implementation, notification is achieved through Unix signals. When the view becomes active, the database entry is first updated (see Chapter VI), then a signal is sent to the appropriate process, and finally a message is sent to the appropriate process. The message contains the values specified in the "ret_exp" portion of the view definition.

IV.4.3.3. Performance Values

The performance value in view specification is used by the monitor to determine how quickly it needs to report the result. This field allows the user to help the monitor generate different configurations, dependent upon the constraint specified. Consider the example of the view Both Queue Limits exceeded shown below.
VIEW DEF Both_Queue_limits_exceeded
  (QueueManager[1] Queuesize1 > 24) AND
  (QueueManager[2] Queuesize2 > 24)
Correct to within (10 UNITS)
NOTIFY 1199 Seventh 1500
within (876 UNITS)
RETURN (Queuesize1)
END VIEW DEF

Reporting the result, in the network environment, translates into a single message send operation. In order to reduce message traffic due to monitoring, the monitor can queue values that it is sending to the central monitor, and send multiple values in one message, rather than send multiple smaller messages. In the Both_Queue_Limits_exceeded example, since the performance value is so large, the monitor can wait until it has other messages to send before sending the result to the view manager. Resident monitors also have a "flush" action. If a particular resident monitor has not sent a value for a time period of "n" seconds, then all the values the monitor has queued are immediately sent to the central monitor. This is known as "flushing" the queues.

IV.5. Evaluation of the Method

In this section we discuss whether the mechanisms meet the goals of language, application and machine independence, whether they aid in the reconfigurability of the monitor and whether or not they simplify the use of the monitor.

The collection mechanisms presented here were used in four applications and on two different hardware architectures. The applications were: a game, a robotics application, a parallel sorting application, and the RTDM project. These four applications were chosen because of the variety of software which they model. The game is a communication intensive application. The robotics application is a real-time
application with computation intensive as well as communication intensive portions. The parallel sorting application is representative of replicated software, where each portion functions independently of the other portions. The RTDM project was chosen because of the size of the application. Recall that the RTDM project involves several hundred machines.

The game, the robotics application, and the parallel sorting application were all built using tools provided by the Issos project. Because of this, each program is defined as a set of objects. The translation from the objects the user defines, to the entities seen by the monitor is done by the user. The user is also responsible for defining the attributes associated with each entity. Conceptually, all the translation from objects to entities and the definition of attributes occurs before the user defines monitoring views of the program. Attribute definitions are given to the monitor, and they are automatically translated into sampling code and inserted into the target application. Automatic compilation promotes ease of use, since the user does not have to be involved in the process of probe generation or insertion.

Once the entities, relationships, and attributes are defined, the user then defines the monitoring views needed. These views are given to the monitor, which automatically generates the necessary collection mechanisms. These mechanisms are then hand inserted into the target application. In each of the three Issos applications collection mechanisms were generated that were used in the performance of dynamic program adaptations. In the robotics application, for example, the size of a queue was monitored and processes were added based on the queue's size (see [45] for a more detailed explanation of the adaptations performed).

In the RTDM project, programs are written as processes. No object model was used. In this case, the translation from processes to entities was done by the user. Each process is considered an entity. Attributes are defined by the user, using the attribute
grammar presented in the previous section. Attributes are once again automatically compiled into sampling mechanisms. These mechanisms are then included with the processes defined. Users then define views based on these attributes. In the RTDM project we generate hundreds of sensors based on view definitions and several hundred machines are monitored.

The target applications chosen represent a variety of distributed and parallel software. In each case, view definitions are used to generate the collection mechanisms needed to collect information about the target application. Our implementation is only a prototype implementation, but in each of the applications mentioned here the mechanisms performed well. The prototypes help demonstrate that the concept of views is application independent, and that the distributed collection mechanisms used are application independent. What is required, in order for our approach to work, is that the user must be able to describe the application as entities and relationships between those entities. The user must also be able to define attributes for those entities and relationships. We have shown that it is possible to describe most distributed or parallel software as entities and relationships, and that it is possible, with the user's help, to define attributes on those entities and relationships.

The performance characteristics of the mechanisms described in this chapter will be discussed in Chapter VIII.

We define two types of reconfigurability. The first type of reconfigurability is the ability of the monitor to dynamically change the flow of information. This type of reconfigurability increases the performance of the monitor as we will show in Chapter VIII. The second type of reconfigurability is the ability of the monitor to dynamically change what is being monitored, including changing the type of collection being performed. This type of reconfigurability increases the usability of the monitor, by providing additional functionality. As we will see in a later section, since tracing and
sampling have different costs associated with them, the ability to dynamically change the collection mechanism may also increase the performance of the monitor.

In our prototype implementation it is possible for sensors to send information directly to the central monitor, thus bypassing the local monitor. This bypassing is possible because collection mechanisms are generated based on static view specification. The monitor can determine the flow of information based on the view specification it receives. Our monitoring design allows for such reconfigurability.

The other type of reconfigurability is the ability to dynamically change what is being monitored, including changing what type of collection mechanisms are being used. Because of the real-time constraints placed on the monitor, this type of reconfigurability is limited in our design. Two types of dynamic changes to the information being monitored are allowed. The first simply allows the user to stop/start monitoring an already existing view, which results in the disabling/enabling of the collection mechanisms associated with that view. The second dynamic change involves the changing of constants used in the condition section of a view definition. In Section III.1.3 we explained how users might define a view to be active when the condition

\[(\text{netstat\_Seventh\_Bad\_header\_chksums} > 15)\]

is true, and then later change their minds and want the view to be active when the condition

\[(\text{netstat\_Seventh\_Bad\_header\_chksums} > 50)\]

is true. As we mentioned before, our implementation allows for this type of reconfigurability by changing all references to constants to be references to an array. This allows the monitor to treat constants as a special form of variables.

The monitor does not allow for dynamically changing conditions involving variables. For instance, if the user originally defined a view to be active when:
and later wanted to change this view to be active when:
\[
(n\text{etstat\_Seventh.Bad\_header\_chksums} > n\text{etstat\_Seventh.average\_value1})
\]

the monitor cannot do this. The reason the monitor cannot do this is that collection mechanisms are compiled into the target application. This type of change requires either a change to the collection mechanisms used, or it requires that extended sensors not be used. (As we shall show in the next chapter, since analysis code is also compiled, even the elimination of extended sensors would not allow for this type of reconfigurability.) Neither of these is possible given the real-time constraints of the monitor.

IV.6. Conclusions

In this chapter we discussed the distributed collection methods available for collecting information: tracing and sampling. We also discussed the two collection mechanisms, sensors and probes, that our monitor uses to trace and sample programs. We introduced extended sensors, which can be used to increase the performance of the monitor. We showed the independent nature of these mechanisms by showing how they are implemented in a network environment and in a multiprocessor environment. It is worth noting here that the specification of views is machine independent, but the view compiler is machine and language dependent. This allows the monitor to generate the correct version of the sensors and probes.

This chapter also showed that attribute and view definitions, introduced in the previous chapter, can be automatically compiled into collection mechanisms. Automatic compilation of attributes and views increases the ease of use of the
monitor, since users only need to be concerned with attribute and view definitions and not with sensor and probe creation. We introduced extensions to the view definition language to allow for the specification of correctness and performance constraints, which are used in tailoring the performance of the monitor. Finally, we introduced constructs for specifying the actions to be performed by the monitor, namely to retrieve information or to notify users. These constructs will be discussed in later chapters. In the next chapter we will discuss the analysis phase of monitoring.
CHAPTER V
Analysis

Analysis is the phase of monitoring where information is used to determine if a view is active. A view is active when the condition specified by the user is true, and the correctness constraints specified by the user have been met. Legal conditions are expressions involving currently known values of attributes. The monitor does not accept conditions involving the prediction of future attribute values.

In the following sections we discuss issues regarding the analysis of monitoring information including:

1. A cost model for perturbation and latency.
2. Location of analysis.
3. The cost/benefit tradeoffs in analysis location.
4. The algorithm for location selection.
5. The effects of correctness constraints on location selection.
6. How the monitor handles the distributed-time problem.
V.1. Monitoring Cost Model

In order for the monitor to be predictive, we need to capture the costs associated with monitoring. A cost model is also useful when attempting to make the monitor more efficient, i.e. if the bottlenecks are detected then it is possible to attempt optimizations on those bottlenecks.

Monitoring has two quantifiable "costs" associated with it: latency and perturbation. **Latency** is defined as the amount of time between the occurrence of an event and when the user becomes aware that the event has occurred. **Perturbation** is defined as the percentage of overhead an executable program incurs due to monitoring.

Intuitively, latency includes the time it takes to collect information and the time it takes to analyze the information. Given a typical flow of information, latency includes:

1. The time spent by a sensor or probe to do any analysis to determine if the information should be transferred to the resident monitor.
2. The time it takes for a sensor or probe to transfer information to the resident monitor.
3. The time it takes the resident monitor to analyze the information to determine if it should be sent to the central monitor.
4. The time it takes to transfer information from the resident monitor to the central monitor.
5. The time it takes for the central monitor to determine if the information should be sent to the user.
6. The time it takes for the central monitor to send the information to the user.

Below we introduce an equation that captures the cost associated with perturbation. We make the following simplifying assumptions in our cost model.

1. The cost of storing information is negligible. We make this assumption for two reasons. First, our monitor is designed as a real-time monitor, so that the quantities of information stored are not likely to be large. Storing large quantities of information takes too much time to allow the information to be used in real-time. The other reason we make this assumption deals with the target hardware, discussed in Section I.2.1. Our target hardware has a large main memory, so we assume all storage will be in-core storage.

2. A second assumption we make is that there is no intermediate buffering of the information at the sensors, the resident monitor or the central monitor. As we will discuss in Section V.11, intermediate buffering can be added.

Our equation for latency is shown below. Latency is measured in units of time:

$$L = D \times (A_s + K_s \times (R_{st} + A_t + K_t \times (R_{tc} + A_c + K_c \times R_{cu})))$$ (V.1)

where:

- \(D\) = amount of data associated with each event. This is measured in bytes.
- \(R_{st}\) = the time it takes to send a message from the sensor to the resident monitor. This is measured in seconds/byte.
- \(R_{tc}\) = the time it takes to send a message from the resident monitor to the central monitor. This is measured in seconds/byte.
\( R_{cu} \) = the time it takes to send a message from the central monitor to the user. This is measured in seconds/byte.

\( K_s \) = ratio between the amount of data received by a sensor and the amount of data output from a sensor. We assume this to always be \( \leq 1 \).

\( K_i \) = ratio between the amount of data received by the resident monitor and the amount of data output from the resident monitor. We assume this to always be \( \leq 1 \).

\( K_c \) = ratio between the amount of data received by the central monitor and the amount of data output from the central monitor. We assume this to always be \( \leq 1 \). This assumption holds because sophisticated post processing is performed outside the monitor, e.g., post processing may occur in the graphical display routines.

\( A_s \) = average time spent doing analysis in a sensor. This is measured in seconds/byte.

\( A_i \) = average time spent doing analysis at the resident monitor. This is measured in seconds/byte.

\( A_c \) = average time spent doing analysis at the central monitor. This is measured in seconds/byte.

Perturbation is the second quantifiable cost associated with monitoring. Intuitively, perturbation is the percentage of reduction of speed that a process realizes because it is being monitored. It consists of the amount of time a processor spends in "monitoring" related activities, divided by the total amount of time a process would take to run if it was not being monitored.

The following formula captures the cost of perturbation as a percentage:

\[
P = \frac{N \times (L - (A_c + K_c \times R_{cu})) + M_{init} + T_{process}}{T_{process}}
\]

Where:

\( L = \) latency
$N$ = the number of events.

$M_{\text{init}}$ = amount of time spent initializing the monitor.

$T_{\text{process}}$ = amount of time the process takes to run when not monitored.

In this chapter we will show that the quantifiable costs associated with monitoring, perturbation and latency, can be affected by:

1. changing the location of analysis,
2. changing information flow, or
3. by changing the method of information collection.

Note that the cost model for perturbation involves $N$, the number of events that occur. To be totally predictive, the monitor would need to know the value for $N$, before the program is executed. Since it is unrealistic to assume the monitor knows the number of events, it is impossible for the monitor to be totally predictive. Given that we assume the number of events will be low, however, we can make the monitor somewhat predictive in nature.

At this point we mention costs that we do not directly take into account in our models.

1. The time spent by the operating system doing a process switch between the target application and the resident monitor. This is seen only on machines like the Suns, where the resident monitor and the target application share a CPU. On a multiprocessor, process switching may not occur, since the resident monitor may run on a separate board. Because of this we have chosen to ignore this term in our equations.
2. We also ignore message interference caused by the monitor. If the monitor generates a large number of messages that must be transmitted across a network, then these messages may interfere with messages generated by the target application. For instance, if the target application
is extremely communication intensive and the monitor generates large numbers of messages, then the target application may notice a decrease in performance due to message collisions, message queueing, etc. We have chosen to ignore this term in our equations, since this is an application-dependent term, and to an extent a network-specific term (i.e., it does not usually occur in a multiprocessor where all communication is through shared memory).

V.1.1. Parallelization

We briefly want to discuss the difference between the wall time and processor time, when talking about latency. Assume that a program consists of three processes: P1, P2 and P3. One way to calculate the total latency for the program would be to sum the latencies of each of the three processes, P1, P2 and P3. Our equation for latency does this. In actuality, since much of the processing can be done in parallel (specifically A\textsubscript{2} and A\textsubscript{3}) the perceived latency (wall time) may be much less than what we calculate. This is important, because it shows the distribution of analysis decreases the perceived latency for the entire program compared with the perturbation predicted using the cost models.

V.1.2. Perturbation versus Latency

Tradeoffs exists between perturbation and latency. Placing analysis close to where information is collected decreases the cost of collection, and latency is decreased. However, the closer the analysis is to the process being monitored, the more the monitor may perturb that process. We mention these tradeoffs, but propose
no solution. In Chapter IX we discuss some ways of modifying our analysis-placement algorithm, given in Section V.5, to account for the latency/perturbation tradeoffs.

V.2. Non-Quantifiable Monitoring Costs

There are 'costs' associated with monitoring that are not captured in either of the two equations. We list these costs here and will discuss them throughout this chapter.

1. The flexibility of the monitor.
2. The message traffic interference introduced by performing monitoring.
3. The ability to share information between tools.

V.3. Sample Program

A parallel robotics application (see Figure 10) is used as a sample application program. The application has the following hardware: a vision system (including a camera), a turntable (conveyor) and a robot-arm. The locations of blocks, placed on the turntable, are determined by the vision system. The robot arm can track and move the blocks.

The parallel program implementing block trace and manipulation consists of multiple software objects including: multiple controller objects, a vision object, a conveyor object, a robot-arm object, and multiple inverse-kinematics objects. The controller objects receive commands from a user such as "track a block", "pick up a block", or "find location of a block". The vision object receives and stores block location information from the self-contained vision system. The conveyor object controls the speed of the conveyor. The inverse-kinematics objects calculate the joint angles necessary to move the robot-arm. The robot-arm objects issue commands to the robot-arm.
V.4. Goals

Figure 11 shows the default flow of information in the implementation of the monitoring system described in this dissertation. Proposed changes to this flow are given in Section V.7. Sensors and probes collect information from the user's program and send it to a resident monitor; resident monitors send information to a central monitor; the central monitor sends information about active views to the user.
In many cases, much of the information collected is "disposable" information, i.e. it is information that does not change whether the view is active or not.

As we stated in Section 1.3, one of the goals of this research is to create a monitor that can be tailored to meet specific performance criteria. This objective translates into changing the costs of latency and perturbation. These changes are achieved by reducing the number of disposable messages generated by the monitor. Given the formulas in Equations (V.1) and (V.2), there are three ways to decrease the cost of monitoring:

1. Change the total amount of data in the system, decrease D;

2. Change the rate at which data is transferred from one location to another, decrease $R_{it}$, $R_{ic}$ and/or $R_{cu}$; or
3. Change the amount of time spent doing analysis at each step, resulting in a decrease in $K_s$, $K_t$, and/or $K_c$.

In order to decrease the cost of monitoring, one or more of the above three changes must be realized. Each of these three changes has drawbacks associated with it. For most systems, the data transfer rates, $R_{st}$, $R_{tc}$, and $R_{cu}$ are fixed. In order to change these rates hardware support or operating system support is needed. In order to transfer information using shared memory, for instance, the operating system must support shared memory constructs. If the operating system does not support shared memory, then the operating system must be changed [41]. Changing the operating system is not always a viable option. A drawback associated with decreasing the amount of data, $D$, is a loss of functionality of the monitor. The only information the monitor can provide to the user is information to which the monitor has access. Decreasing $D$ decreases the amount of information the monitor has access to, and thus decreases the amount of information the monitor can provide to the user. Increasing the amount of monitoring performed at the sensor level and at the resident monitor, changing $K_s$, $K_t$, $A_s$ and $A_t$, has the drawback that it may increase perturbation.

In this dissertation we concentrate on changes to the placement of analysis, corresponding to decreases in $K_s$ and $K_t$ and increases to $A_s$ and $A_t$. We will also suggest a method to decrease $D$. Methods of changing $R_{st}$ and $R_{tc}$ are discussed in [41] and in Section VIII.1.4.

An Example.

The following simple example shows how the location of analysis effects the total cost of monitoring. Consider the sample robotics application. Assume the application has been run before and it has been decided that if there are more than 10 elements in the queue then another inverse-kinematics object needs to be added. (Again, how the adaptation is performed is discussed elsewhere (see [45])). In order to perform the adaptation it is necessary to efficiently monitor the size of the queue.
In order to monitor the queue size, the following definitions are needed:

ATTRIBUTE DEFINITION FOR OBJECT Inverse_Kinematics

QueueSize[1] : que_size

END ATTRIBUTE DEFINITION

VIEW DEF QueueSize_Exceeded

(Inverse_Kinematics[1].QueueSize[1] > 10)
Correct to within (25 UNITS)
NOTIFY 1199 seventh 1500
within (876 UNITS)
RETURN (Inverse_Kinematics[1].QueueSize[1])

END VIEW DEF

The analysis necessary to determine when the condition:

(Inverse_Kinematics[1].QueueSize[1] > 10)

is true can occur at three locations: the sensor used to collect the information (thus making it an extended sensor), the resident monitor, or a logically centralized monitor. Changing the location of analysis to either the sensor or the resident monitor will decrease the amount of data transferred, reducing the number of disposable messages and thereby decreasing the total cost of monitoring\(^4\). Analysis location placement is performed automatically, based on view definitions. In the next sections we will discuss analysis placement.

V.5. Algorithm for Analysis Placement

In this section we introduce the algorithm used for analysis placement. In this algorithm we make the following simplifying assumptions:

1. All sub-expressions consist of simple boolean conditions.

---

\(^4\)Disposable messages present two problems. First they increase latency and perturbation. Second, in some environments, like the network environment, increased message traffic by the monitor interferes with the message traffic of the user. This is a cost associated with monitoring that we have chosen not to quantify.
2. There are at most two sub-expressions in any one expression.

3. We assume for our algorithms that only AND and OR conditions exist.

We assume NOT is a unary operator, and thus it is handled as the negative of a regular expression.

These assumptions make the implementation of the algorithm easier; they do not effect the generality of the approach.

The algorithm looks at view definitions and generates analysis code. Recall that view definitions include information about which attributes are involved, which objects those attributes reside in, and information about correctness and performance. The monitor also has information concerning where each object will execute. This information is obtained either from the program construction system, in the Issos project, or from the user, in the RTDM project.

The algorithm appears below.
```
EXP : FOR(i=1 to the number of expressions) DO
lookup the location of all the attributes involved in
expression[i]
IF all the attributes are in the same object
THEN generate one extended sensor for the entire
expression, doing all the analysis in the sensor
ELSE
   SUB: FOR (j=1 to the number of subexpressions) DO
      IF all the attributes in subexpression[j] are in same object
         THEN generate one extended sensor for the subexpression
            generate local and central analysis for this part
            of the expression, including binary operator
      ELSE IF all the attributes in subexpression[j] are on the
            same machine
         THEN generate sensors for each attribute
            generate local analysis for this subexpression
            generate central analysis for this part of the
            expression, including binary operator
      ELSE IF all the attributes in subexpression[j] are on
            different machines
         THEN generate sensors for each attribute
            generate central analysis for this subexpression
            generate local and central analysis for this
            part of the expression, including the
            binary operator
   END SUB
END EXP
```

The algorithm shown above is responsible for generating collection and analysis code. It assumes that all collection will be achieved through the use of sensors. We will relax this assumption in a later chapter. The algorithm parses each subexpression and attempts to determine the types of sensors that should be generated: regular sensors or extended sensors. It also generates all necessary resident and central monitor analysis code. It decides what to generate based on the location of the different attributes involved in the expression, trying to place the analysis as close to the location of collection as possible.
V.6. Analysis Using Extended Sensors

A sensor performing analysis is called an extended sensor. Intuitively, analysis done in a sensor reduces the amount of information that is transferred from the application program to the site of analysis, either the resident or central monitor. Using variables from Equation (V.1), increasing the amount of analysis done in a sensor decreases the terms:

\[ D \times K_s \]
\[ D \times K_s \times K_t. \]

Using extended sensors decreases the amount of data transferred between the sensors and the resident monitor, and between the resident monitor and the central monitor.

Consider the previous view definition for QueueSize_Exceeded. If a regular sensor is used, then every time QueueSize1 changes, the new value must be transferred to the resident monitor for analysis. (This transfer is denoted by \( D \times R_{s1} \) in Equation (V.1)). If the QueueSize is frequently changed and most of its values fall within an acceptable range, i.e., they are less than 10, then the sensor is generating numerous disposable messages.

Using extended sensors decreases the number of disposable messages generated. Analysis performed in extended sensors requires no data transfer since analysis is done in the address space of the process being monitored. For instance, if the inverse_kinematics object has an extended sensor for the view defined, then a message is only generated when the view is active. The extended sensor necessary is shown below:
if (status[1] == 1)
if (que_size > 10)
{
    sensor_struct.command = NEW_VAL;
    sensor_struct.sensor_num = 1;
    sensor_struct.int_result = que_size;
    sendto(monitor_socket_send, &sensor_struct,
           sizeof(sensor_struct), 0,
           &monitor_sin_send,
           sizeof(monitor_sin_send));
}

If the value of the attribute QueueSize (which is defined to be the variable que_size) never exceeds 10, then the extended sensor never sends a message to the resident monitor. The reduction in the number of messages by use of extended sensors is proportional to the number of changes that occur to the attribute being monitored.

The increase in the cost of monitoring due to using extended sensors is, in this example, the addition of the statement:

    if (que_size > 10).

We mentioned perturbation in Section V.1.2. Extended sensors add statements to sensors to evaluate the view condition. This increases perturbation to the monitored process. Part of the perturbation is due to the collection of the information and part is due to the analysis of the information. If the condition involves some complicated calculation, it may be the case that the perturbation due to analysis outweighs the perturbation due to collection, in which case extended sensors should not be used.

V.7. Information Flow

A second method of increasing the performance of monitor, by reducing the number of disposable messages, is to change the flow of information. In the naive flow of information, shown in Figure 11, information flows from the application program to the resident monitor, then to the central monitor and finally to the user.
Some extended sensors can report directly to the user. For instance, in the example of a user interested in knowing when the condition:

\[ \text{Queuesize} > 10 \]

is true, it is possible for the sensor to send the information directly to the user. This is possible because only one attribute is involved. If multiple attributes are involved, then all of them must be defined in the same object in order to realize this optimization. If multiple objects are involved, then the analysis cannot occur in the sensors, and this new flow of information cannot be used. Figure 12 shows this new flow of information.

![Diagram of new flow of information]

Figure 12: New Flow of Information

Other tradeoffs exist in deciding whether to change the flow of information. A non-quantifiable cost is associated with changing the flow of information. If the value
of an attribute is sent directly to a display tool, then it becomes inaccessible to other tools. Instead of being a "common" value, situated in a common database, the value becomes "private", usable only by a specific tool.

V.8. Local Placement of Analysis

While extended sensors may reduce the number of disposable messages, their use is limited to views, or conditions, involving attributes defined on a single object\(^5\). If the user defines fairly simple views, involving attributes local to one object, then the monitor can generate extended sensors.

Views involving attributes from many objects may not be able to use extended sensors. Consider the following view definition:

```plaintext
VIEW DEF Both Queue Limits Exceeded
  (Inverse Kinematics[1].QueueSize1) >
  (Inverse Kinematics[2].QueueSize2)
CORRECT TO WITHIN (25 UNITS)
NOTIFY 1199 seventh 1500
WITHIN (876 UNITS)
RETURN (Inverse Kinematics[1].QueueSize1)
END VIEW DEF
```

In this case the user defines a view that is active when the condition

```
QueueSize1 > QueueSize2
```

holds. Using the algorithm given in Section V.5 the monitor first checks the location of QueueSize1 and QueueSize2. If they are both in the same object, it is possible to use an extended sensor. In this case they are not in the same object. Since we cannot use extended sensors the analysis must occur at another location. According to the algorithm, if the two objects involved reside on the same machine, then local

\(^5\)In a multiprocessor environment this may not be true since a sensor may have access to variables in other objects.
analysis can occur. If they reside on different machines then central analysis must be used.

Local analysis is performed to further reduce the number of disposable messages between the resident monitor and the central monitor. Consider the composite view \texttt{Both\_Queue\_Limits\_Exceeded}, defined in the last section. As we stated, extended sensors cannot be used for this view since the attributes are not in the same object. Local analysis, decreases the message traffic between the resident monitor and the central monitor. In this example, each time one of the sensors reports a new value, the resident monitor checks the value to see if the condition has been met. If it has, then the resident monitor generates a message to the central monitor. Again, this optimization reduces the message traffic between the resident monitor and the central monitor. Local analysis has the same latency/perturbation tradeoffs as sensors do (see section V.1.2).

If local analysis cannot be performed, meaning the attributes are on different nodes, then central analysis must be performed. Central analysis is the most costly form of analysis. Since all values generated must be sent to the central monitor for evaluation. If the central monitor is only logically centralized, it is possible that it resides on a node with a resident monitor. If this is the case, then the cost of monitoring is reduced, assuming the cost of message sends is $\gg$ the cost of analysis done by the central monitor.
V.9. Further Optimization

In some instances, even though all the attributes in the view condition are not on the same node, not all the analysis has to occur at a central site. Optimization can be performed at the sub-expression level. Consider the following sample composite view specification.

```
VIEW DEF Both_Queue_over_24
  (Inverse_Kinematics[1] Queuesize1 > 24) &
  (Inverse_Kinematics[2] Queuesize2 > 24)
Correct_to_within (25 UNITS)
NOTIFY 1199 seventh 1500
within (876 UNITS)
RETURN (Queuesize1)
END VIEW DEF
```

In this example, even though Queuesize1 and Queuesize2 do not both reside in the same object, extended sensors can be used to provide intermediate analysis. The conditional expression consists of two sub-expressions:

```
(Inverse_Kinematics[1] Queuesize1 > 24)

(Inverse_Kinematics[2] Queuesize2 > 24)
```

The algorithm used in determining whether to use extended sensors evaluates each sub-expression to determine if extended sensors or local analysis can be used in the sub-expression. In this case, extended sensors can be used for both sub-expressions, since they involve a single attribute. Extended sensors in each object would only send a message to the central monitor when their condition is true. This would result in a decrease in the number of disposable messages.
V.10. Sampling versus Tracing

Another way to increase the performance of the monitor is to change the method used to collect information. In many cases, the monitor can use both tracing and sampling when doing analysis, and still meet the correctness criteria specified by the user. Again, tracing is when all changes to an attribute are monitored; sampling is when only some changes to an attribute are monitored. A switch from tracing to sampling may result in a decrease in the total amount of data collected by the monitor. (This is a decrease in $D$, using Equation (V.1)). Sampling may result in a decrease in perturbation. This depends on the frequency with which you sample versus the frequency that the attribute’s value actually changes. This section explains how the correctness constraint is used to determine whether to do sampling or tracing.

Consider the following view again:

```
VIEW DEF Both_Queue_limits_exceeded
    (QueueManager[1] Queuesize1 > 24) &
    (QueueManager[2] Queuesize2 > 24)
Correct to within (25 UNITS)
NOTIFY 1199 seventh 1500
within (876 UNITS)
RETURN (Queuesize1)
END VIEW DEF
```

This view is defined active only when the value of Queuesize[i] is greater than 24 in both instantiations of QueueManager. For situations involving multiple attributes, we make use of a view’s correctness value. Here, the monitor can trace both queuesizes, or it can trace one and sample the other. Deciding between these options is straightforward due to the known times associated with sampling and tracing. The algorithm given in Section V.5 must be changed to produce both sensors and probes, rather than just sensors.
This method of combining sampling with tracing is useful only when sub-expressions are joined by binary operators which require both sub-expressions to be true (this is the AND operator). Once again, this method is only possible because the monitor is provided with information about where the objects run and which attributes are defined on which objects. Below we show the flow chart for the algorithm that decides whether to use tracing or sampling.

This algorithm generates both sensors and probes if possible, optimizing the performance of the monitor. The algorithm is similar to the previous algorithm where only sensors are generated. In this algorithm, however, the monitor checks each subexpression to see if probes can be used and the correctness constraints can still be met. This assumes the monitor has access to the costs associated with tracing and sampling. This is a valid assumption, since these costs are well known costs. The algorithm assumes that the first subexpression will be traced, and that each subsequent subexpression will either be traced or sampled, depending upon whether or not the correctness constraints can be met. If all the binary conditions are ORs, or if none of the correctness constraints can be satisfied using probes, then this algorithm is identical to the previous algorithm that makes no attempt to use probes.

V.11. Other Methods

Other methods exist which may improve the performance of the monitor. One of these methods is to buffer messages at sensors or at the resident monitor. As was shown in [41], buffering of information decreases the total number of messages sent between two sites. If there is a significant overhead associated with message setup, then a scheme which buffers information and sends one large message, as opposed to multiple smaller messages, will decrease the total monitoring time. Using Equation
Figure 13: Analysis Placement Flowchart
(V.1) and (V.2), if $R_{st}$ and $R_{tc}$ are dominated by setup costs, and are not dependent on the size of the message being transferred, then buffering will decrease perturbation. Of course any intermediate buffering will result in an increase in latency.

Buffering at the sensor level may result in a decrease of perturbation related to timestamping of events. Each sensor timestamps the information it sends out. The current method is to use the time returned by the Unix `gettimeofday()` routine. The cost associated with using this routine is currently a few hundred microseconds per call. There are implementations that allow direct access to the clock and thus greatly reduce the time required to perform a timestamp [18]. It might be possible to buffer the values at the sensor level before calling the timestamp routine, and timestamping multiple values at once. The benefit of this method is a decrease in perturbation. The drawback of this method is a loss of accuracy, since multiple values will now have the same timestamp.

V.12. Drawbacks to Our Approach

In this section we introduce two possible errors that may result using our methods of monitoring. The first is that the monitor may report that a view is active, when in reality the view is not active. The second is that the monitor may not report a view is active, when in fact the view is active. In both cases, the errors are a result of the implementation and not a result of the method. While neither error is common, they are possible and worth mentioning.
V.12.1. Incorrectly Reporting a View is Active

Incorrectly reporting a view is active may occur when a result has not reached the site doing the analysis, and the analysis mechanism must make a decision based on the information it currently has. One example of when this might happen is when the user wishes to know when X and Y are both true. (Also assume, for simplicity, that a central analysis mechanism is being used.) When the central analysis mechanism receives a message that X is true, it checks to see what value it has for Y. If the last value it has for Y says that Y is true, then the monitor reports the view is active. It is possible that Y is not true and the message saying Y is false has not reached the central monitor yet. The message could be buffered by the system or it could be buffered by the monitor. It would be possible to force the analysis to wait until all buffered messages are delivered before making a decision. We chose not to do this because of the real time application of the monitor. Assuming a reliable message protocol, the window in which this may is occur is equal to the time it takes information to get to the central monitor. This is approximately 12.5 milliseconds in our system.

V.12.2. Incorrectly Reporting a View is Not Active

Incorrectly Reporting a View is not active occurs when a result has not yet reached the monitor. Again assume the user is interested in knowing when X and Y are both true during the same time. This time assumes the user is tracing X and sampling Y (it only samples Y when X is true; also assume a central analysis mechanism). Assume the central monitor gets a message that X is true. It immediately
samples the value of $Y$. Between the time the sample request is sent and the time the request is serviced $Y$'s value changes from true to false. The monitor receives a message saying $Y$ is false and reports that the view is not active, when in reality it was active. Again, this time is bounded by the amount of time it takes information to go from a sensor to the central monitor, which is approximately 12.5 milliseconds in our system.

Both of these errors could be alleviated if we traced all values, if we had a global clock, and if we were not interested in using the monitor for real-time applications. Since this is a real-time monitor, we have chosen to live with the possibility of these kinds of errors.

V.13. Examples

In this section, we present an example from the network environment. This simple example is used to demonstrate the usefulness of our approach in the network environment.

The following example, done in the network environment, demonstrates the usefulness of our approach. The time unit for the network environment is a millisecond. The first example shows how the placement of analysis in sensors decreases the number of disposable messages. The second example demonstrates the tradeoffs in performance between sampling and tracing.

In the RTDM project, users are interested in knowing when "emergency" conditions occur. One emergency condition is when too many error packets occur per time period. (We define it to be an emergency when there are more than 5 error packets in a ten second time period.)
The routine collecting information on a subnet is named "getpacks". This routine periodically collects information about the subnet. One of its attributes is Error_packets, which keeps track of how many error packets were detected during any one snapshot period. For this example assume that getpacks collects information every ten seconds. (How frequently it collects information is specified by the user.)

Given this scenario, and without extended sensors, a message is generated and sent from getpacks to the resident monitor every time the value of Error_packets changes. Due to the nature of the project, the value of Error_packets rarely exceeds 5. (This is only natural since a value greater than 5 suggests some sort of error with the network.) Without extended sensors, getpacks generates, on the average, 6 disposable messages per minute for each of its attributes, and there are approximately 40 attributes⁶. With extended sensors, a message is only generated when a threshold is exceeded. There are also other routines, similar to getpacks, which are also being monitored. Without the use of extended sensors, several hundred disposable messages are generated every minute. The obvious conclusion is that using extended sensors are essential to performance of the monitor. Without extended (or hardware/OS support) the monitor would not perform in real-time.

A second example, demonstrating the tradeoffs between sampling and tracing, was discussed in Section V.10. That example shows how the monitor decides whether to trace or sample a value based on the correctness constraint. Recall that sampling is cheaper than tracing, so whenever possible, the monitor should sample values.

⁶Appendix C has a list of the attributes.
V.14. Validity of the Approach

In this section we will show that placing analysis in sensors and in the resident monitors minimizes the latency of the information. We will also show that in the network environment the placement of analysis does not significantly effect perturbation.

V.14.1. Validity of Latency Model

In this section we will be using the cost models introduced in Section V.1. We begin with some assumptions and definitions:

1. We assume that \( N \) and \( D \) are fixed constants and that they are large. The product of the two will be represented by the greek letter \( \kappa \) in the equation below.

2. The size of a message generated by a sensor or the resident monitor is fixed in length.

3. \( A = A_s + A_f + A_c \) where \( A \) is the total amount of time spent doing analysis.

4. \( K_s \times K_f \leq 1 \). This means \( K_s \) and \( K_f \) are related.

5. \( K_s \propto f(A_s) \) and \( f(A_s) \) is a non-decreasing function of \( A_s \). This states that \( K_s \) is inversely proportional to a function of the amount of analysis performed at the sensor, \( A_s \). Or that as the amount of analysis done at the sensor level increases, \( K_s \) either stays the same or decreases. An intuitive explanation of this is that a sensor can do two things with the data, it can
generate a message for each data item or it can generate one message for all the data items combined. In the worst case, analysis done in the sensor can decide that each data value needs to be sent on (either to the resident monitor or to the central monitor), in which case \( K_s = 1 \). (\( K_s = 1 \) means that sensor analysis does not cut down the number of messages generated.) This is no worse than doing no analysis in the sensor. In the best case, sensor analysis determines that only one message be sent, in which case \( K_s \) is a minimum. (\( K_s = D_{in} \), meaning only 1 message is sent for all the data.)

6. \( K_t \propto f(A_t,A_f) \). The same argument above holds for this. Also note that \( A_t \) is a function of \( A_f \) because of point 3.

7. We define a new variable \( X \) where: \( X = (R_{sl} + A_f) \times K_s \). \( X \) is non-increasing as \( A_f \) increases. Intuitively, this states that as \( A_f \) increases (which either does not change \( K_s \) or decreases \( K_s \)) then \( X \) either stays the same or decreases. \( X \) has a minimum where \( K_s \) has a minimum. In order for this to be true, we assume that \( R_{sl} \gg A_f \), so much so that \( R_{sl} + A_f \) is indistinguishable from \( R_{sf} \).

8. We define a new variable \( Y \), where \( Y = (R_{lc} + A_c) \times (K_s \times K_f) \). \( Y \) is non-increasing as \( A_c \) and/or \( A_f \) increase. Again, as \( A_c \) and/or \( A_f \) increase, the corresponding \( K \) values either do not change or they decrease. \( Y \) has a minimum where \( K_s \times K_f \) has a minimum. Due to definition, this occurs when \( K_s \) has a minimum value. Again, we assume that \( R_{lc} \gg A_c \), so much so that \( R_{lc} + A_c \) is indistinguishable from \( R_{sl} \).

Given these assumptions, we can rewrite Equation (V.1) as follows:

\[
L = (A_x + X + Y) \times \kappa
\]
We know that X and Y both either stay the same or decrease when $A_z$ increases. This is due to the fact that as $A_z$ increases, $K_z$ either stays the same or it decreases. If $K_z$ decreases then the value of $X$ decreases much more rapidly than the value of $A_z$ increases. The best possible result would be to minimize the value of $K_z$. Since the minimum value of $K_z$ is data dependent, it is impossible to statically decide the best value for $A_z$. Since we make the assumption that $A_z << R_{sl}$, and since we know that $K_z$ is proportional to $A_z$, we attempt to minimize latency by maximizing $A_z$. This occurs when $A_z = A$. This means that doing all the analysis in a sensor would result in the lowest possible latency, since any decrease in $A_z$ would result in either no change to $K_z$ or in a decrease in $K_z$. In actuality, this may not result in minimizing latency, since maximizing $A_z$ may not decrease $K_z$. We accept this result for two reasons. First, because as we stated, it is impossible to statically determine the optimum value for $A_z$ that results in a minimum for $K_z$. Second, the difference between the actual minimum and the result obtained by maximizing $A_z$ is small, since the dominating term in the formula is $R_{sl}$.

Given these assumptions, and since getting $A_z = A$ is impossible in a distributed application, the next best thing is to increase $A_z$ as much as possible, and then increase $A_i$ as much as possible. (This is because $Y$ is a function of both $A_z$ and $A_i$). This proves that latency is smallest when all the analysis is performed in the sensors. It also proves that the closer the analysis is to the collection, the less data transfer that occurs, and the smaller the latency. Thus, our algorithm that attempts to place analysis as close to collection as possible minimizes latency.
V.14.2. Validity of Model Using Derivatives

In this section we show, by derivation, that our algorithm for analysis placement is correct.

Re-written:

\[ L = D \times [A_s + K_s \times (R_{sl} + A \times \delta)] \]

where:

\[ \delta = A_I + K_I \times (R_{ic} + A_c + K_c \times R_{cm}) \]

Since \( \delta \) does not involve \( A_s \), we will consider it a constant.

We know that \( K_s \) is related to \( A_s \), basically that \( K_s = \frac{1}{A_s} \).

If we postulate that \( K_s = \frac{\beta}{A_s} \), we obtain:

\[ L = D \times [A_s + (\frac{\beta}{A_s}) \times \delta] \]

which can be written as:

\[ L = D \times [A_s + \frac{(\beta \times \delta)}{A_s}] \]

\[ \frac{dL}{dA_s} = D \times [1 - \frac{(\beta \times \delta)}{(A_s \times A_s)}] \]

The values of \( dL \) are as follows:

1. \( dL \) at \( A_s \) very small is very large negative
2. \( dL \) at \( A_s = \sqrt{(\beta \times \delta)} \) is 0
3. \( dL \) at \( A_s \) large is \( D \)

\( L \) is minimized when \( A_s = \sqrt{(\beta \times \delta)} \). Unfortunately, we do not know the value of \( \beta \), since its value depends on the data. This implies that it is not possible to statically determine the placement of analysis that will result in the lowest latency. Our analysis placement algorithm chooses to maximize \( A_s \), which may not result in a minimum
value for latency. Given the assumption that transfer rates $\gg$ analysis time, however, the difference between our solution and the optimum solution is small.

Analogous arguments can be made for the value of $A_r$. What the results in this section show are that if we assume the value of $A$ (the total amount of analysis to be done for each event) is small, then placing analysis close to collection results in a near optimum value for latency.

V.14.3. Validity of the Perturbation Model

We now mention something about what happens to perturbation due to minimizing latency. Below we re-write Equation (V.2), substituting terms in for $L$. First, we need one further assumption:

1. $P_{time}$ and $T_{init}$ are constants.

$$P = \frac{(A_s + X + Y) \times \kappa + T_{init}}{P_{time}}$$

Since we have already assumed that $R_{sl} \ll A$ and $R_{lc} \ll A$ the dominant terms in this equation become $R_{sl}$ and $R_{lc}$. Changes to the $A$ terms have little to no effect on the cost of perturbation.

In a multiprocessor environment, however, where $R$'s approach 0, the $A$ terms become dominant, and then $L = A_r$. This implies that the more analysis performed centrally (or at least on a different processor) the lower the perturbation becomes, approaching $\frac{T_{init}}{P_{time}}$ from above. Restated, if no analysis is done in sensors, and transferring information to another processor is free (or very inexpensive), there will be no perturbation to the monitored process.
V.15. Distributed Time

In this section, we will briefly discuss the problem of time [17] in distributed systems, and explain how we deal with it. The major problem in distributed systems with regard to time, is that there is no global clock, and thus no consistent global time. Our approach is to synchronize the clocks as closely as possible, and enlarge the granularity of the clock "tick".

V.15.1. Clock Synchronization

There are certain properties about a network environment that are measurable. Below are the properties needed to attempt to synchronize the clocks on different machines in our system.

1. Each node in the system has a local clock, signified by $T_n$.
2. The minimum time to transfer a message between any 2 processors in our distributed system is measurable and is noted as $T_c$.
3. The minimum time to receive a message (process and recognize it) is measurable and noted by $T_r$.
4. The maximum time to transfer a message between any 2 processors in the distributed system is measurable and noted as $T_{max}$.

To synchronize the clocks, we need to do the following\(^7\):

---

\(^7\)For this entire synchronization process we assume that the communication medium is totally reliable, and that the synchronization routine is the only routine accesses the network.
1. A logically centralized node $N$ is selected by the user and sends (broadcasts) its local time to all other nodes.

2. All other nodes set their local clocks to be equal to $T_n + T_c + T_r$.

The clock granularity now becomes $T_g = T_{\text{max}} - (T_c + T_r)$. We now say that events occurring on different processors at times $T_1$ and $T_2$ occurred simultaneously iff $|T_2 - T_1| < T_g$.

V.15.2. Time in Analysis in the Network Environment

In our distributed system, on any subnet, the following times have been measured: $T_r + T_c$ is approximately 5 milliseconds, and $T_{\text{max}}$ is approximately 7 milliseconds (we assume reliable data transfer while doing synchronization). The value for $T_g$ is then 2 milliseconds. We now have a definition for the granularity of the clock, 2 milliseconds. If two values are timestamped by different machines at times $T_1$ and $T_4$, then from the monitor's standpoint they occurred at the same time if $|T_1 - T_4| < 2$ milliseconds. If the user specifies that the analysis has to be correct to under 4 milliseconds, then the values collected must be timestamped from the same clock. This implies the analysis must be of information collected from the same machine. Since different target machines will have different clock units, it makes sense for the user to specify correctness to within 'X' time UNITS. For instance, the granularity of the clock on a Sun is 16 microseconds, whereas the granularity of the clock on the Encore Multiprocessor is 1 microsecond. The user should be shielded from this type of implementation information whenever possible.
V.16. Temporal Issues

Our analysis mechanisms work on information that is currently available. We classify this as a valid time approach [42]. In the next chapter we will briefly discuss how our analysis mechanisms would change if we had temporal storage mechanisms.

V.17. Predictive Capability

In the introduction we stated that one of the goals of this research is to create a monitor that is predictive. We said we would achieve this predictive characteristic by first developing cost models for perturbation and latency, and by then empirically demonstrating each of the cost models is correct. In this chapter we have introduced cost models for perturbation and latency. We have shown that given certain assumptions, it is possible for the monitor to be made predictive. We have also shown it is impossible for the monitor to be made totally predictive. In Chapter VIII we will validate our model using examples and measurements. We will also show that these models can be used to predict the performance of the monitor in certain situations.

V.18. Conclusions

In this chapter we introduced a monitoring cost model. Based on this model, we showed that the total cost of monitoring could be decreased in many situations by doing the analysis as close to the collection of the information as possible. This is due to the high cost of information transfer in the network environment. We discussed
ways of decreasing the cost of monitoring by changing the flow of information, and by changing the methods used to collect information.

Inherent to this work is the knowledge that tradeoffs exists when determining how to monitor. The most apparent tradeoffs are between latency and perturbation and between functionality and speed. We have suggested a few methods of increasing the speed of the monitor and decreasing the latency while maintaining full functionality. These methods have been tested in a network environment and proved to be valid. Their validity is based on the fact that data transfer rates, $R_{sf}$ and $R_{lc}$, are significantly larger than the perturbation due to analysis. Experiments shown in Chapter VIII demonstrate the usefulness of this approach in a multiprocessor environment, where data transfer rates are lower. Based on Equations (V.1) and (V.2), we expect that in any architecture where $R_{sf}$ and $R_{lc}$ are small (meaning low transfer costs), the placement of analysis close to the collection is less necessary than it is in the network environment. We will show, however, that our cost models hold even in environments the transfer of information is inexpensive.
CHAPTER VI
Data Storage

In our approach to monitoring, users define views of their programs in terms of the monitorable attributes of their programs. The monitor translates these view definitions into collection and analysis mechanisms and begins collecting and analyzing information about those views. The monitor is also responsible for taking action based on the views defined. For instance, the monitor might have to notify a user when a view becomes active, or store the view so the user can see it later. Some of the information collected by the monitor is used by the monitor to determine when a view is active. Once a view becomes active, the information must be shared with other tools. For example, views can be displayed to users via a graphical interface tool (to be discussed in Chapter VII), or views can be given to other tools, such as the adaptation controller.

The monitor needs a mechanism that can store intermediate values used by the monitor in determining a view's status and that can also store and share views with other tools in the system. If a tool is to provide the necessary functionality to the monitor, it should support the monitor's notion of objects and relationships, as well as provide a mechanism that allows for actions such as notification and storage. The tool should also allow the monitor to retain its real-time performance characteristics. We have chosen a database as the tool to perform these functions.
Our database model is an extension of the entity relationship model [4]. The main extension to the model is the inclusion of action routines, which will be discussed below. We chose this model specifically because it meets the criteria specified in the previous paragraph.

In the rest of this chapter, we will discuss our entity relationship (E/R) database model, explain how objects are mapped to entities, explain how views are mapped to relationships, and explain how the database and the monitor interact. It is not the intent of this chapter to give an indepth explanation of the data model chosen, the database design or the data manipulation language. For a detailed description of these, the user should read [29].

VI.1. Entity Relationship Model

In our entity relationship model, the database consists of: entity sets, relationships and sets.

An entity set is comprised of a set of entities. An entity is similar to a record. Each entity has attributes associated with it, and these attributes have values. The values of the attributes may change over time. Rather than define entity sets, relationships and sets using the standard diagram approach, we have chosen to define each using a textual approach. In our approach a template is associated with each definition. In the following examples, entity sets are defined using an E-TEMPLATE definition, relationships are defined using an R-TEMPLATE definition, and sets are defined using an S-TEMPLATE definition. The following example shows the definition for the entity set processor.
In this example, we have defined entity set processor. Every entity set, relationship, and set has a TYPE field, which is a user specified field. The TYPE field defines the type of the elements in the entity set, relationship or set. The entity set processor has two attributes: Host_Name, which is a string, and QueueSize, which is an integer.

Relationships are used to store information about related entity sets. The following example shows the information necessary to define the relationship runs_on.

This example defines the relationship runs_on. The relationship relates the entity set process to entity set processor. This relationship has two attributes, process_number, which is an INTEGER, and processor_name which is a STRING.

While our definitions of entity sets and relationships are identical to the standard model's definitions [4], our definition of sets differs slightly. In the standard approach, sets are derived by taking the union of two or more already existing entity sets. In our model, we allow the user to define sets that are the union of entity sets, relationship, or sets. We mention this here as a means of explaining the kinds of sets allowed in our model. From a monitoring perspective, most sets are of limited interest.
For a further explanation of why sets are useful, the user is once again referred to [29].

The following example shows the definition for a set of type Processor_set.

```
S-TEMPLATE
    TYPE : Processor_set
    CONTAINS processor_pyramid, processor_sun
    ATTRIBUTES
        NONE
END
```

This example defines a set of type Processor_set, which contains the entity sets processor_pyramid and processor_sun. There are no attributes associated with this set. This set would be the union of the entity sets containing entities of type processor_pyramid and entities of type processor_sun. In the standard relational approach, this would be defined by saying processor_pyramid "is a" (ISA) Processor_set and processor_sun "is a" (ISA) Processor_set.

VI.2. Instantiations

In this section we will briefly make the distinction between definitions and instances. We will use entities as examples, but the same discussion is true for relationships and sets. Definitions of entity sets are shown above. In a definition, the template for the entity set must be given to the database, so that the database can register it. In the example above, an entity set whose elements are of type processor is defined.

After an entity set has been defined, the user may create an entity. If, for example, he wants to store information about the processors Wayward, Seventh and Favorite, he can create three entities of type processor. All three entities belong to the entity set processor. Figure 14 shows this.
Figure 14: Entity Definitions and Instantiations
VI.3. Monitor and Database Interaction

This section discusses how the database and the monitor interact. To the monitor, the database is the tool it uses to store views. Other tools then retrieve these views from the database. This interaction is shown in Figure 15.

Figure 15: Tool Interaction

The interaction between the monitor and the database is as follows:

1. The monitor issues commands to the database to create entity sets, entities, relationships, and sets.
2. The database creates these entity sets, entities, relationships and sets.
3. The monitor associates *action routines* with some entities, relationships and sets (to be discussed below).
4. The monitor stores, manipulates and updates view information in the database.
5. Other tools (such as the adaptation controller) retrieve view information from the database.

VI.4. Entity Creation

Entity sets and entities are created during the program construction phase. At that time the user defines all the attributes associated with an entity set, defines all the entities, and enters the entities into the database. Views are defined on existing entities (or entity sets). Consider the following view definition:

```
VIEW DEF Queue1_larger
    (QueueManager[1].Queuesizel) >
    (QueueManager[2].Queuesizel)
    Correct to within (5 UNITS)
    NOTIFY 1199 Favorite 1500
    within (876 UNITS)
    RETURN (QueueManager[1].Queuesizel)
END VIEW DEF
```

This defines the view *Queue1_larger*, which is active when the condition

```
(QueueManager[1].Queuesizel) > (QueueManager[2].Queuesizel)
```

holds. This view relates the entity *QueueManager[1]* to the entity *Queuemanager[2]*. (It is assumed that during the program construction phase the

---

8In implementation, the definition of attributes is done as part of the monitoring phase of programming. This was done to simplify the implementation process of the Issos project, and because there is no real program construction phase in the RTDM project.
user defines an entity set of type QueueManager, and that QueueManager[1] and QueueManager[2] are entities in that set.) In our implementation, we have chosen to store views as relationships. We do this so the attributes specified in the RETURNS portion of the view definition are stored and are easily accessible.

The relationship used to store this view is shown below.

```
R-TEMPLATE
  TYPE : Queue1_larger
  RELATES
    QueueManager : QueueManager
  ATTRIBUTES
    Queuesize1 : INTEGER
    Active : BOOLEAN
END
```

Again, this example defines the view Queue1_larger. Our implementation allows us to store this view as a relationship. The relationship defined for this view has two attributes, Queuesize1 and Active, of type INTEGER and BOOLEAN respectively. Before continuing it is worth mentioning that some views involve only one user defined entity or entity set. For instance, consider the following view:

```
VIEW DEF Queue1_limit_exceeded
  (QueueManager[1].Queuesize1) > 10
  Correct to within (5 UNITS)
  NOTIFY 1199 Favorite 1500
  within (876 UNITS)
  RETURN (QueueManager[1].Queuesize1)
END VIEW DEF
```

Although this view only involves one user defined entity, QueueManager[1], we store this as a relationship. The relationship is between the entity QueueManager[1] and the entity 10. We assume items such as integers, reals, characters, etc. all belong to their own respective entity sets, and that these entity sets exist by default.

We now continue our discussion on how the monitor handles the view Queue1_larger. The relationship Queue1_larger must be registered with the database. Registration is achieved using the generic database command
The general format for this command is:

```
_cdb_template_reg("type_def", "selector", "contains_list", "relates_list", "attribute_list", ActionRoutineName).
```

The parameters have the following meanings:

1. **type_def** - this is the user defined field TYPE, shown in all the definitions.
2. **selector** - this is either an E-TEMPLATE, an R-TEMPLATE or an S-TEMPLATE. This tells the database whether the entry is an entity set, a relationship or a set.
3. **contains_list** - this parameter is only valid when defining a set. It defines the members involved in the set.
4. **relates_list** - this is only valid when defining relationships. It defines the entity sets being related.
5. **attribute_list** - this lists the attributes being defined along with their types.
6. **ActionRoutineName** - this names the action routine associated with this entity set, relationship or set as will be discussed in Section VI.4.1.

To register the view Queue1_larger with the database, the following command must be issued.

```
_cdb_template_reg("Queue1_larger","R_TEMPLATE", 
                  "", "QueueManager:QueueManager", 
                  "QueueSize1:INTEGER, 
                  Active:BOOLEAN", 
                  NULL)
```

---

9For a complete list of database commands see Appendix B
This command registers a relationship of type Queue1_larger with the database. The relationship has the attributes QueueSize1 which is an INTEGER and Active which is a BOOLEAN. There is no action routine associated with this relationship.

To create an instance of this relationship, the monitor must issue a _cdb_ent_create command. The format for this command is as follows:

```
_cdb_ent_create("type_def","Extra_attributes")
```

The parameters in the statement above, have the following meanings:

1. **type_def** - the user defined type of a previously registered entity set, relationship or set.

It is possible to add extra attributes for specific instances. This means that two entities in the same entity set, for example processor, can have different numbers of attributes.

To create an instance of the relationship Queue1_larger, the following command is issued.

```
_cdb_ent_create("Queue1_larger",""")
```

To create a set, the monitor first registers the set with the database, then creates an instance of the set, and finally enters members into the set. As was previously mentioned, to register anything with the database the command _cdb_template_reg is used. To create an instance, the command _cdb_ent_create is used. To create a set of type processor and to create an instance of this set, the following commands would be used.
A set must be initialized before it can be used. This is done with the 
_cdb_set_init command, since the _cdb_ent_create command does not initialize a set. Once the set has been initialized elements can be added by doing a set union between the set and the new element.

VI.4.1. Action Routines

An action routine [15], which is similar in nature to a trigger, can be associated with an entity, a relationship or a set. Usually action routines are associated with relationships. From the monitor’s point of view, action routines allow for interactions between tools. For instance, the monitor is interested in notifying other tools like the adaptation controller, the display routine, etc., when a view becomes active. One way to implement notification is to force the monitor to write a separate notification routine for each tool it is integrated with, and to use mechanisms such as Unix signals to achieve notification. A more consistent method is to use the database to achieve notification. Conceptually notification is achieved by having the database modify an attribute in another database entry. This modification may then trigger user specific notification routines. This method allows the monitor to treat all notifications identically. Each tool provides an action routine to the database that can perform its version of notification. This routine is then invoked when the appropriate attribute is updated.
Consider again the view definition Queue_larger, which we have been using throughout this chapter. In this view, the user wants to be notified when the condition
\[(\text{QueueManager}[1].\text{Queuesize}_1) > (\text{QueueManager}[2].\text{Queuesize}_1)\]
holds. An action routine could be associated with the relationship Queue_larger that performs the notification. The code body for the action routine can be provided by the monitor or by the tool defining the view. In implementation, the action routine would check the values of the attributes, and if the condition has been met, then a Unix signal is sent to process 11999 on the machine Favorite. A message would then be sent to the user (in the network environment) containing the value of the attribute Queuesize_1, or the user could retrieve the value from the database. A benefit of using this approach is that if the user wants to redefine the semantics of notify, for instance instead of being signalled he wants mail sent to an operator, the monitor does not need to be changed. The action routine associated with that particular view must be changed, but from the monitors point of view nothing has changed.

VI.4.2. Database Operations

In Appendix B we provide a list of the common database operations. This list, including the examples, is taken verbatim from [11].

The operations listed in Appendix B allow the monitor to manipulate views. The manipulations performed by the monitor are: creating entity sets, entities, relationships, and sets and updating attribute values. In a later chapter it is discussed how the user can dynamically create views using these operations.
VI.5. Temporal Issues

Time is an important issue in real-time monitoring. In this chapter we will discuss the main issues associated with time (from a monitoring point of view), and we will discuss how we have handled these issues. We will also discuss extensions to our work to make include temporal database [42] operators.

VI.5.1. Time in Databases

Snodgrass and Ahn have identified four categories of database models(snotime), based on how each handles time. Their four categories are: snapshot databases, rollback databases, historical databases and temporal databases. Below is a synopsis of his categories.

A snapshot database models the real world by snapshots taken at a particular point in time. Updates are performed using standard data-manipulation operations such as insertion and deletion. Changes made to the relations take effect as soon as they are committed, past states are lost once a change occurs.

A rollback database is similar to a snapshot database with the exception that past states are stored. This approach requires that the past states be indexed by transaction time, which is the time at which the information is entered into the database. Selecting a snapshot state is referred to as rollback. Rollback databases allow users to access information as of a particular previous time.

Historical databases differ from rollback databases since historical databases store a single historical state per relation, thus storing the history as it is best known.
Errors are corrected as they are discovered and no record is kept of the errors corrected. Previous states are not kept, so it is also impossible to view a relation as it was some time in the past.

\[ \text{Figure 16: Temporal Database} \]

A temporal database supports both transaction time and valid time. A temporal database makes it possible to view information relative to some other moment, thus capturing the history of retroactive and proactive changes. Figure 16 (this figure was taken from [42]) shows a temporal relation. The relation starts with no elements. At time $T_1$ three tuples are added. At $T_2$ one more tuple tuple is added. At time $T_3$ a tuple is added and an existing tuple is deleted. Finally, at time $T_4$ a previously existing tuple is deleted (this is the result of a tuple being added that should not have been added). Every update has two phases, first the historical relation is copied, then the update is applied to this new historical relation.
VI.5.2. Our Approach

We use an historical database approach. Our view language is also historical in nature. In our database, past states are not stored. When information is entered into the database, the database is considered to be valid at that time. In Section VI.5.3 we discuss some of the drawbacks associated with using this approach. The following example is used to explain why this approach presents problems.

Consider the following view:

```plaintext
VIEW DEF Queue1_larger
   (QueueManager[1].Queue1size > 24) &
   (QueueManager[2].Queue2size > 30)
Correct to within (20 UNITS)
NOTIFY 1199 Seventh 1500
within (876 UNITS)
RETURN (QueueManager[1].Queue1size )
END VIEW DEF
```

Assuming the objects reside on different nodes and both attributes are being traced, the following scenario is possible. Queue1size exceeds 24 at some time \( T_1 \). This information is sent from the sensor directly to the central monitor, which enters it into the database. At some time \( T_2 \) Queue1size decreases to under 24. This is noted and sent to the central monitor. That message is delayed in arriving at the central monitor. At time \( T_3 \), before the message that the value of Queue1size is less than 24 arrives at the central monitor, the value of Queue2size exceeds 30. This information is sent to the central monitor and arrives before the message with timestamp \( T_2 \). The monitor evaluates the condition:

\[
(QueueManager[1].Queue1size > 24) &
(QueueManager[2].Queue2size > 30)
\]

\(^{10}\)All times are considered to be global times for this example.
It appears the condition is true, so the user is notified. When the message arrives that says at time $T_2$ $\text{QueueSize}_1$ was less than 24, the view is marked as invalid. We do not consider this an error, since the condition was valid given the information available at the time of evaluation. Since what to do with an invalid view is tool dependent, we choose to ignore the problem. A graphical tool may choose to redisplay the information. An adaptation controller, that has already implemented an adaptation, may choose to ignore the new information or try to undo the adaptation. The solution to the problem would be to allow each tool to provide an action routine that was fired when a view was retroactively changed. The problem occurs because of the real-time nature of the monitor. If the monitor did flushes, or if it waited until all messages arrived and were serviced, then this problem would not occur. Since we want to build a real-time monitor, we have chosen to accept these errors.

VI.5.3. Limitations of Our Approach

In this section we briefly discuss some of the limitations of our approach. We point readers to [42] for a more complete list of problems of historical databases.

The major limitation with our approach is the inability to easily specify issues concerning time. It is difficult, using our approach, to specify relationships such as: before, after, while, etc. Say, for example, that a user is interested in knowing when the attribute $\text{QueueSize}_1$ exceeds its threshold before the attribute $\text{QueueSize}_2$ exceeds its threshold. Using our current view definition language, this view would be difficult to specify. In order to allow for this specification, the view language would need to be extended to include a temporal constructs such as those offered in TQUEL [41].

From a database standpoint two major changes would be necessary to implement a temporal approach. Both were mentioned when we introduced temporal databases.
First, past states need to be stored. Second, time must become an implicit field associated with each entity. (More accurately, time would be implicitly associated with each attribute/value pair of an entity.) Again, the database would need to support a temporal query language, such as TQUEL.

Using a temporal database changes neither the mechanisms used for analysis nor the location of analysis, i.e., we would still use sensors, probes, resident monitors and a central monitor. Since views could be defined in terms of time, however, the analysis mechanisms would need direct access to the temporal database. In the network environment, the database would have to be shared among all the processes being monitored. This is a non-trivial problem [29].

VI.6. The Database Implementation

In this section we briefly describe the database implementation and discuss possible improvements to the implementation. Again, for a more detailed discussion of the database implementation see [29].

The database is implemented in the programming language C and all entities, relationships and sets are stored as character strings. The interface between the database and the rest of the world is a message based interfaced and uses standard Unix message calls of send and receive.

There are two major drawbacks to our implementation: it is centralized and it is slow. The implementation is centralized for ease of implementation. In our prototype implementation, we did not want to worry about problems such as: how to distribute the database, concurrency control, consistency, etc. Instead, we wanted to test the concept of integrating tools using an entity relationship database. There are two obvious ways to improve the database: distribute it and increase its performance.
Below we offer suggestions on how to achieve these improvements. In the following suggestions we ignore the problems associated with concurrency control and consistency.

Distribution can be achieved at many levels. For instance, the database management routines can be replicated so that a copy resides on each machine in the system. The data in the database can be partitioned so that each machine has access to the information it needs. Doing so increases the performance of the database by allowing the monitor on each machine to have local access to the database. Recall that local messages are faster than messages that must go across the network. A further optimization would be to allow for shared memory access (on the same node) between the monitor and the database. This would reduce the access time from a local message send, costing approximately 3.8ms, to a local memory access, costing a few hundred microseconds.

A second method of distributing the database would be to compile a portion of the database management routines directly into the resident monitor. This would allow the resident monitor direct access to data in the database, and it would give the monitor quick access to all the database operations. This would reduce the cost of interaction between the database and the monitor to the cost of a procedure call.

A third method of distribution is to allow the database management routines to be compiled with each resident monitor and each process in the target application. This would allow not only the resident monitor access to the database, but it would allow sensors access to the database. This would allow sensors efficient access to all the database operations, thus increasing the power of sensors.

This last distribution method has already been prototyped. The monitor, including sensors, have their own data structures. While the monitor maintains these structures itself, they are similar to the structures stored by the database. The main
difference is that these data structures are not shared with other tools. Keeping private copies of information is reasonable, since there are structures that will be used exclusively by one tool in the system, and the database should be able to optimize access to these structures. Performance improvements, such as having a shared memory interface between the resident monitor and the database, to allow the monitor to store views to be shared with other tools, would increase the performance and usability of the monitor.

VI.7. Evaluation of Tool Integration via a Database

One of the goals of this research is to provide a real-time monitoring tool that is integrated with other tools in the programming environment. In this chapter we introduced entity relationship database that allows for such integration. In the next chapter we present evidence of the usefulness of the entity relationship model, by showing how a graphical tool and the monitor interact using an entity relationship model.

Our general goal is to allow tool integration. Most databases allow for multiple tools to share information, whether the database uses records, trees, entities, etc. We found the entity relationship model to be useful in describing both program information and monitoring information [29]. Interactions, such as notification, can be achieved by extending the standard entity relationship model to include action routines.

Integration through the database allows the monitor to present a consistent interface to all tools, thus increasing the ease of use of the monitor. The monitor creates entities, relationships and sets and stores them in the database. The monitor associates action routines with some of these entries. This interface means that the
monitor does not have to create a new interface everytime a new tool is added to the system. For example, if a debugging tool is added to the programming system, the interaction between the monitor and the debugger is well defined. The debugger will define views or the information it needs. The monitor will store these views in the database. Any specialized action routines the debugger needs it must supply to the database. Thus, from a monitoring perspective, tools can be added and removed without changing the monitor. Without action routines the monitor would be forced to create special interfaces for each tool in the system.

VI.8. Conclusions

In this chapter we introduced an extended entity relationship model as a model usable to achieve real-time tool integration. We also introduced a database tool built in conjunction with the Issos project, that implements an extension of the entity relationship model, including action routines. The entity relationship database model provides routines needed by the monitor to store, manipulate, and share views with other tools in the system. We have suggested changes to the implementation of the database that would allow it to be a more useful tool to the monitor, i.e. to allow the monitor to perform tool integration in real-time. In the next chapter we give an example of how the entity relationship model is used in tool integration.
CHAPTER VII
User Interface

Throughout this dissertation we have discussed the concept of views. Users define views of their programs, the monitor collects information concerning views, and the database stores and shares views with other tools. In this chapter we discuss the user interface portion of the monitor. The chapter is divided into two portions. In the first portion we will motivate the need for a user interface and list the requirements of such an interface. The second section introduces a prototype implementation of a user interface built in conjunction with research into real-time monitoring of distributed systems.

VII.1. Motivation for an Interface

Mechanisms are needed to allow users to see the information that has been collected by the monitor. There are many possible ways to provide such a mechanism. Consider the example of a distributed sorting application that is communicating via messages that was introduced in Chapter I.1. Assume the user is interested in monitoring all the message traffic in the system, but is currently interested in just seeing the information concerning the communication between process P1 and P2. One possible approach to displaying the information in this case would be to display
all the message communication information. Clearly this is an ineffective method since it provides the user with too much information. The proper method is to display subsets of information. We define views as the appropriate subsets to display.

Mechanisms are needed that allow for view definition, view manipulation and view display. We have already stated that users will define the views used to determine what is monitored. In this section we introduce criteria, both subjective and objective, for the user interface.

The first criterion of the user interface is that it should be easy to use. We define an interface to be easy to use if it builds upon tools the user already uses. By building upon existing tools, the learning period for the user will be reduced and the user should feel more comfortable with the system.

The interface should guide the user and prompt the user for input. This is especially true when views are being defined. A user, with a minimal amount of proficiency, should be able to define fairly complex views. Again, guiding and prompting the user increases the usability of the monitor.

The interface should be view oriented. By this we mean that the interface should allow for the definition of views and that views should be the unit of display. The interface should also work at a high level, e.g. view definition should be at the object/attribute level. This shields the user from changes to the underlying levels. Say, for example, the programming language was changed from C to Pascal. Since the definition of views is language independent, changes to the underlying system do not effect the user's view definitions. Thus the implementation of the user interface is not tied to a particular system or language. At some level the monitor must change views into collection and analysis code, and at that level the monitor is system dependent.

Both the input and output sections of the interface should be designed upon the same principles. Views defined during the definition phase should be displayed during
the output phase. Changes to views should be made using the same interface with which the views were originally defined.

The interface should be integrated with the rest of the system. This is important to ensure efficiency. If the interface is directly linked with the database, for instance, the speed with which the interface works can be increased.

Since the environment is always changing, the interface should be modifiable. This implies that the interface should be modular. Input and output phases should be separate, thus allowing both to change separately. More precisely, we want the specification of how a view is to be displayed kept separate from the implementation of the display. This allows implementations to change, without effecting specification.

The interface should be somewhat timely. It must work in "good" time\(^{11}\) in order to display the information accurately. A display tool that performs too slowly tends to disinterest the user and actually lags behind the real world. In a later section we give examples of this.

With these features in mind, we will now discuss one implementation.

VII.2. An Implementation

In this section we discuss an implementation of the user interface. Some of the requirements set forth in the previous section have been met, others have not. In the following sections we discuss and evaluate our interface and explain what changes are necessary in order to meet all the requirements. We divide our interface into two portions: definition and display. We discuss view definition first. Included in the view definition section is the attribute definition section.

\(^{11}\)We define good time to be somewhere around 2-3 seconds. This is an arbitrary choice based on our experience.
We first introduce the tool used to create and define attributes, and then explain how the same tool is used for view definition. We include attribute definition here, since it was implemented as part of this research.

Users define attributes of entities using an attribute definition language. In order to enter definitions in this language, and to free the user from learning and memorizing its syntax, a mechanism is needed that guides the user through attribute definitions. We have chosen to implement a language oriented editor (ALOE) \cite{22, 25} as the tool for entering attribute definitions. ALOEs are similar in nature to syntax-directed editors and provide a framework in which the user can be guided through attribute definitions. ALOEs also provide routines to parse statements entered by the user. Parsing routines are necessary when the monitor translates view definitions into collection and analysis code.

Before an ALOE can be used, all the rules for the language must be explicitly defined, and any parsing schemes needed must also be defined. Once this is done the ALOE may be used. We built a very simple ALOE for attribute definition. The complete grammar for the ALOE is given in Appendix A.

To explain how an ALOE works, we will show how a user defines the attributes process_name and process_time for an object called process1. Periodically, we will show what the ALOE window looks like. One need not know ALOE’s internals in order to use it.

In Figure 17, we show the entire screen as seen by the user. In subsequent examples we will only show a portion of the screen, leaving out the lower half.

The fields in Figure 17 have the following meanings:

1. $entity_name : The name the user wishes to give the entity.
2. $entity_type : The type of the entity, it is used as the TYPE field in the database entity definition command.
shelltool = /bin/csh
ATTRIBUTE DEFINITION entity_name
TYPE entity_type
FOR OBJECT object_name
attrdefn
END ATTRIBUTE DEFINITION

Figure 17: ALOE Screen
3. \$object\_name: The name of the object for which these attributes are being declared.

4. \$attrdefn: A list of attribute definitions

Other information contained in the figure includes the type of the node currently being highlighted. In this case, \$entity\_name is of type \texttt{string()}\). The user first chooses a type, and then enters the value for that field. The ALOE also lists all the commands the user can enter. These include: \texttt{CATEGORY-HELP}, \texttt{COMMANDS}, \texttt{CONSTRUCTIONS}, etc. These commands are useful in changing and editing information being entered.

Using our example, the user first selects a type for the field being highlighted, \$entity\_name. In this case, the only option for the type field is \texttt{string()}\), so the user enters the word \texttt{string()}\). The ALOE then prompts the user for a value. Once he enters a value, \texttt{process\_def} in this case, the cursor is automatically moved to the next user supplied field and that field is highlighted. This is shown below.

\begin{verbatim}
ATTRIBUTE DEFINITION process\_def
TYPE \$entity\_type
FOR OBJECT \$object\_name
\$attrdefn
END ATTRIBUTE DEFINITION
\end{verbatim}

The user selects a type for this field, from the list provided. Again, the only valid type is \texttt{string()}\). He then enters a value for \texttt{entity\_type}. When the cursor moves to the \$object\_name field, the type is listed as either \texttt{single} or \texttt{multiple}. This is because of the macro capabilities mentioned when we first discussed attribute definitions. Attributes can be defined either for a single object or for multiple objects. The user chooses a type, \texttt{single} in this case. The ALOE then prompts the user telling him that the valid type of \texttt{single} is \texttt{string()}\). The user then selects \texttt{string()}\) as the type and is prompted for a value. After he enters the value \texttt{process\_1}, the cursor highlights the \$attrdefn field. \$attrdefn is
defined to be of type $attrdefn$. Since this is the only acceptable type, the user enters it. The statement then expands to the following:

```plaintext
ATTRIBUTE DEFINITION process1_def
  TYPE process_type
  FOR OBJECT process1
  ($type) $attrname : $exp
  $attrdefn
END ATTRIBUTE DEFINITION
```

This tells the user to now enter the type of attribute. Our implementation could change to allow the ALOE to verify the validity of the type (i.e. it could allow the user to only enter integer, string, real, etc.). The same is true for the $exp statement. Similar changes could be used to check the validity of the $exp statement. We chose to implement both the $type and the $exp statements as type string() merely for ease of implementation. When the user is finished inserting attributes for this object, he hits the return key when the cursor is highlighting the $attrdefn statement. The result, after the user has entered the two attributes, is shown below.

```plaintext
ATTRIBUTE DEFINITION process1_def
  TYPE process_type
  FOR OBJECT process1
  string process_name : my_process_name
  integer process_time : my_proc_time
END ATTRIBUTE DEFINITION
```

Note that the attribute process_name is defined to be the user variable my_proc_time and the attribute process_name is defined to be the user variable my_process_name. It would be possible at this time to have an ALOE check to see if those variables actually exist in the object. This would be possible if the attributes were being defined in the program construction phase and the program were being constructed within the same ALOE (an expanded version of this ALOE). We did not implement this feature, however.

Attributes for other entities can be inserted by requesting the editor to supply more templates. This is done by doing the `extend-after` command, which is a standard ALOE command.
View definitions are handled by an ALOE very similar to the ALOE used for attribute creation. In theory, both ALOEs could be combined into one large ALOE. The ALOE used for view definitions is shown below.

```
VIEW DEF $view_name $exp $correctness_value $command WITHIN ($perf_value UNITS) RETURN $ret_exp
END VIEW DEF
```

The fields have the following meanings:

1. `$view_name`: The name to be associated with this view. This translates into the TYPE field in the database command to create and entity.
2. `$exp`: The condition to be evaluated to determine when the view is active.
3. `$correctness_value`: How correct the view needs to be.
4. `$command`: Either a notify or retrieve command, telling the monitor what to do with the view when it becomes active.
5. `$perf_value`: The performance value for this view.
6. `$ret_exp`: The attributes to be associated with this entity. It is assumed that all attributes named in this statement are previously defined attributes (i.e. they are defined in another entity).

We will not go through the expansion of this ALOE. It works in the same manner as the previous ALOE. For a description of the expansion of each statement, the user is directed to appendix A. Again, it is possible to do more syntax checking in the ALOE. For instance, we could define the `$exp` to be any valid C expression, and enforce those rules. For ease of implementation we chose not to do this.
VII.3. Graphical Approach

Once the monitor starts collecting and storing view information in the database, a mechanism is needed to display the information to the user. In this section we will discuss one possible approach to displaying view information. We also discuss an implementation done in conjunction with the monitor. It is not the intent of this chapter to give an indepth discussion of all the issues involved in view display, for that see [10, 37].

VII.3.1. The Approach

Users define views, the monitor collects information about views and the database stores views (as entities). It follows that the view is the appropriate unit of display. Throughout this thesis we have mentioned that graphical information should be associated with view definition. One approach to the problem of how to display monitored information is to provide the user with a display language with which to enter display information. Consider the following view definition.

```
VIEW DEF QueueSize_Exceeded
    (Inverse_Kinematics[1].QueueSize1 > 10)
    Correct To within (25 UNITS)
    RETRIEVE
    within (1000 UNITS)
    RETURN (Inverse_Kinematics[1].QueueSize1 )
END VIEW DEF
```

The problem with this view definition is that it does not provide any information about how this view is to be displayed to the user. For instance, should this view be displayed as a bar chart, with one axis being time or should the view be displayed as a flashing icon to get the user's attention. There are other problems associated with display [36], e.g. where on the screen the information should be displayed.
As we said, one solution to this problem is to provide the user with a language for expressing graphical information. We introduce some of the general concepts about what a graphical language must capture, but present no formal graphical language. From a monitoring standpoint, a graphical language should provide constructs that specify:

1. the type of display figure to be used,
2. the location on the screen where the display figure is to be placed,
3. how relationships should be displayed, and
4. animation characteristics.

While this list is not a complete list of all the constructs necessary to properly display all views, hopefully it gives the reader a flavor for the types of constructs necessary. We will now briefly give an example explaining how display information would be specified.

Specification of the type of display figure to be used is critical. The user should have the option of choosing how to display his view. He could choose to display it as some previously defined type, or he could create his own type. This is analogous to a programmer having the option of declaring variables to be of a built-in type (integer, real, etc.) or to declare a user defined type. Standard graphical types might be: squares, circles, triangles, etc. More complicated types might be: bar charts, pie charts, etc. We envision the user specifying this criterion when the view is defined. Below is a sample using the previously defined view QueueSize_Exceeded.

```
VIEW DEF QueueSize_Exceeded
  (Inverse_Kinematics[1].Queuesizel > 10)
Correct to within (25 UNITS)
RETRIEVE within (1000 UNITS)
RETURN (Inverse_Kinematics[1].Queuesizel)
DISPLAY AS (Bar_Chart(TIME,
  Inverse_Kinematics[1].Queuesizel)
```

END VIEW DEF
This view definition defines a view that is to be displayed as a bar chart, where one of the axes is time, and the other is the value of the attribute `Inverse_Kinematics[1].QueueSize1`. This display portion of this view is stored with the view in the database and used by the display routine to display the view. Bar Chart would be another database entry. The interaction between the monitor and this entry would be via action routines, thus keeping tool integration consistent. Using this design it would possible to change the way a bar chart is displayed without having to change the view definition.

Other graphical information could likewise be specified in the view definition statement. This approach has two benefits. First, it allows the user to specify everything he needs to about a view at one time. The second benefit of this approach is that the display information is stored with the view in the database, making the view device independent. For example, if the user specifies that the view is to be displayed as a histogram, then the display routine might display it one way on a Sun 3/50 and another way on a Zenith terminal. This implies that views are associated with a display template and not a display implementation. Again, the implementation of how a bar chart is displayed could change without having to change the view definition.

As stated in a previous chapter, tool integration should be through the database. Integration through the database adds power to the graphical display tool [9]. It allows the graphical display tool access to all the database operators, giving it the power to dynamically create new views. While the real-time performance criterion of the monitor do not allow the user to dynamically request new views be monitored, it is possible for the user to do two things. First, the user can define new views based on information currently in the database, since this requires no changes to the monitor. These new views can be manipulated and displayed to the user, via the graphical display tool. The second way to dynamically create views is to create a "one-time"
view, based on previously defined attributes. Since the monitor associates probes with every attribute in the system, the user could dynamically define a new view based on already existing attributes. The monitor would obtain the current values of the required attributes, and the view would be displayed. Performance of such views would be significantly less than the performance of compiled views.

VII.3.2. Our Implementation

A graphical display system was built in conjunction with the RTDM project. This implementation makes use of IDL [16] structures to simulate an entity relationship database. In this system, an IDL node (similar to a Pascal record) exists for each processor, each subnet, and the main network. An IDL node stores attribute information about that processor, subnet or network. Currently only two kinds of relationships exist: is_on_a_subnet and is_on_the_main_net. Most processors belong to an instance of the relationship is_on_a_subnet, though a few machines are members of the relationship is_on_the_main_net. We now explain and show some of the capabilities of this display routine. Figure 18 shows what the routine shows when it is first started. The icons labeled FISH, TREE, FRUIT, etc. represent subnets.

To see attributes defined on any of these subnets, the user presses a mouse button when positioned at the icon, and a window appears showing the attributes and their last known values. To see a list of the processors on the subnet FISH, the user presses a different mouse button over the FISH subnet icon. All the processors in that set are then displayed, as is shown in Figure 19. To see attributes of the processor BASS, the user presses a mouse button over the icon for that processor. A short list of all the attributes, and their current values is then listed. If there are more attributes than can
Figure 18: RTDM Display Screen1
be displayed in one screen, the user presses another mouse button, and a scrollbar window full of attributes appears. Figure 19 also shows a short window full of attributes for the processor BASS, and a scrollbar window of attributes for the FISH subnet.

A mechanism was built that allows users to change what is being monitored. Two changes are allowed: views can be turned on/off, and constant comparators can be changed. In the former, users can tell the monitor to stop (or start) monitoring a view. This results in the turning off (on) of sensors and probes. Changing a constant comparator results in the changing of the value a sensor is looking for. If, for example, the user declared the following condition in a view definition:

\[(\text{Inverse}_\text{Kinematics}[1] \ \text{Queuesize} > 10)\],

it is possible to dynamically change the value 10 to some other constant.

We now present a sample interaction between the monitor and the graphical tool via the database (IDL structures). For this example, we will use the RTDM project. The goals of this example are:

1. To show why views must be used as the unit of display.
2. To show how the monitor and graphical tool interact using the entity relationship model.
3. To show the feasibility of our graphical tool

The current implementation of the RTDM monitor runs on 10 subnets, and 150 machines. For each subnet, approximately 35 attributes are monitored. For each machine 3 attributes are monitored. This results in a total of approximately 800 attributes being monitored and displayed.

The monitor and the graphical display routines run as separate processes. The current implementation of interaction between the two uses files. The proper interaction is via messages, to keep it consistent with the database implementation. We
Figure 19: RTDM Display Screen2
will assume, for the rest of this discussion, that the monitor and the graphical tool interact through messages. When the monitor receives the value of an attribute that must be updated (i.e. a view's state is changing from inactive to active or vice versa), the monitor sends a message to the graphical display routine. If the two processes reside on the same machine, the messages take approximately 3.8ms. If they reside on different machines then the message takes from 4.8 to 10ms. The graphical display routine, upon receipt of the attribute value, updates the corresponding view. The update involves finding the correct relationship and entity. The graphical display routine then checks to see if the view is currently being displayed. If it is, then the view is automatically updated on the screen. This update corresponds to an action routine, since an action is associated with the view (i.e. an action routine is associated with the view that is responsible for updating the screen when the view's status changes). The process of looking up a view takes a few hundred microseconds. A rough estimate of the amount of time it takes to update the screen is 70-100ms.

Assume, for the sake of argument, that all 800 attributes were being monitored for values greater than 0. (This would not actually happen, since the monitor is only looking for error conditions.) If the display routine attempted to display all 800 attributes simultaneously the graphical display routine could not keep up with the updates. In fact what happens is that the display routine starts losing messages sent to it by the monitor, and thus loses information.

This demonstrates two points. First, the display routine cannot display all attributes on the screen and perform in a timely fashion. Thus some subset of the information should be displayed. We have chosen views as the subset. The second point is that a monitor that attempts to monitor all aspects of a large distributed system is going to suffer performance problems. Inevitably, some portion of the system will become the bottleneck. In this case it is the graphical display routine. It could have
easily been the database. The point is that monitoring specific views of the target application allows the monitor to more perform efficiently.

VII.3.3. Evaluation of the User Interface

In this section we present an evaluation of the user interface. More specifically we attempt to show that the interface presented is fairly easy to use and helps the monitor to achieve its goal of real-time performance.

The most important aspect of our interface was that it allow users to define views and that these view definitions could easily be translated into monitoring code. We built hundreds of views and ran numerous tests using the output from the view definitions. Using the ALOE was fairly straight-forward.

Displaying entities and relationships was fairly easy. For example, each node in the RTDM project is defined by an icon. Relationships are depicted as lines, for instance, if the machine Bass is a on the subnet Fish there is a line between the two, as is shown in Figure 19. Our implementation of the user interface is somewhat application dependent. A solution to this has already been suggested, i.e. graphical information should be stored with the views. That way, the graphical display routine can retrieve a view and determine dynamically how to display that view based on information stored with the view. One drawback to the implementation of the display routine is the speed with which it refreshes the screen. Using windows to display information seems to overburden the Sun on which things are displayed. Suncore [44] has drawbacks associated with it, like the inability to allow user's direct access to the buffer storing the information displayed on the screen. Suncore also keeps too many data structures to allow for a more efficient display of information. One improvement to the performance problem of the display routine would be to bypass SunCore and Suntool software and write routines that directly access the pixels on the screen.
VII.3.4. Conclusions

In this chapter we introduced the requirements of the user interface portion of the monitor. Using examples from the user interface constructed in conjunction with this research, we explained how and why views are the correct unit of display. We also used the user interface implementation to explain how the monitor and the user interface interact using the extended entity relation model presented in the previous chapter. We also presented an overview of some of the costs associated with this interaction. Finally, we offered suggestions on how to improve the performance of the graphical display routine to allow for a more timely display of information.
CHAPTER VIII
Evaluation

In this chapter we evaluate whether the cost models presented in Chapter V actually capture the costs associated with perturbation and latency. If the cost models are correct, it will show that the monitor can be predictive in determining whether views specified by the user can be monitored. We also evaluate whether the placement of analysis close to the collection of the information has the expected effect on perturbation and latency. If the analysis placement algorithm is correct, it shows the monitor’s performance can be tailored to meet the user’s specifications. In conjunction with this, we evaluate whether the results of analysis placement and the cost models have predictable effects in different architectures. If the effects are similar to those observed in the network environment it will show that our approach is machine independent.

VIII.1. Quantitative Analysis

In Chapter V we introduced models capturing the costs associated with latency and perturbation using our monitoring model. We list these models below and refer the user to Chapter V for the complete definitions of the equations.
Note that we have modified the cost model for perturbation so that the number given is the total amount of extra processing time incurred because of monitoring, as opposed to being a percentage. We changed the equation so we could present the raw numbers obtained from the experiments. In order to evaluate the criteria set forth in the previous section (i.e. that the cost models are valid, that the analysis placement algorithm is correct, and that they both hold for multiple machines), we will run a series of experiments. Each experiment is designed to test the validity of specific aspects of the monitor. Since the cost models are closely related to the algorithm for analysis placement, many of the results overlap. In the first set of experiments we test the validity of the cost model for perturbation in the network environment. The second set of experiments is designed to test the validity of the algorithm for analysis placement in the network environment. These experiments also confirm the validity of the cost model for perturbation. The third set of experiments tests the validity of the cost model for latency. Since latency and perturbation are related, the results of the first two experiments also confirm the validity of the cost model for latency. The fourth set of experiments is used to test the validity of the models for perturbation and latency in a multiprocessor shared memory environment. This set of experiments also shows that analysis placement has the expected result in the multiprocessor environments. The final set of experiments are run to obtain useful information concerning communication costs in the network environment.
VIII.1.1. Evaluation of the Cost Model in the Network Environment

In this first set of experiments we test the validity of the cost model for perturbation, by using experiments run in the network environment. The sample program used throughout the evaluation section is an event generator program. The program executes for a period of time, generating events during that time period. The monitored attribute is the variable \texttt{fired\_value}, whose value is determined by statistical input parameters\textsuperscript{12}. Its value is either a 1 or a 0. The user is interested in knowing when the condition:

\begin{equation*}
\texttt{fired\_value} > 0
\end{equation*}

is true. The test program was the only user program on the machine being tested a Sun-3/75. Since the machine is connected to a subnet, we ran the experiment at different times during the day to obtain results when there were both light and heavy loads on the subnet. As we mentioned previously, communication is achieved using TCP/IP UDP messages, each approximately 100 bytes in length. No buffering is performed by the monitor, only normal system buffering is used.

To validate the cost model for perturbation, we present the results of a series of experiments using the event generator program. Each experiment uses a different placement of analysis. For each experiment we present the observed experimental results and discuss the results expected based on the cost model for perturbation.

\textsuperscript{12}We chose to use statistical event generation on the premise that it would make a difference in the observed results. As we will show, the event generation pattern has little effect on the perturbation and latency.
VIII.1.1.1. Simple Perturbation Test Using No Monitoring

We did this experiment to get the unmonitored times for the event generation program. The program loops 5000 times. In each iteration of the loop, the value of the monitored variable can change. Thus, there are 5000 possible events that can be generated by each run of the program. Whether or not to generate an event is determined by input parameters to the program. The input parameters are: the probability that a condition that is true will turn false, and the probability that a condition that is false will turn true. These parameters are used by the program to determine the value of the monitored variable.

To get averages, each experiment was executed 10 times and the numbers shown in the tables below are the averages for each of the experiments. For instance, in Table 1, the results in the row corresponding to 1256 events are results averaged over 10 runs. It is important to note that regardless of the number of events (i.e. the number of times fired_value = 1), the program executes for the same amount of time, an average of 92.3 seconds. This result is expected and shows that the process is not being effected by system routines running on the machine used for the test.

We measured three kinds of time for each experiment: cpu time, system time, and wall clock time. Cpu time and system time were obtained by using the Unix command times() [44]. Wall clock time was measured by sending messages to a timestamp program running on the Encore. At the beginning of each test run, the program sends a Unix datagram message to a timestamp routine running on the Encore. When the program finishes, it sends a message to the same timestamp routine on the Encore. The process on the Encore, upon receipt of a timestamp request, logs the value of its microsecond clock register to a file. We subtract the ending timestamp
value from the beginning time stamp value and label that time as wall clock time.\textsuperscript{13}

Table I: Results from Base Case

<table>
<thead>
<tr>
<th>Number of Events</th>
<th>User Time</th>
<th>System Time</th>
<th>Total Time</th>
<th>Wall Time</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1256</td>
<td>91.867</td>
<td>0.352</td>
<td>92.219</td>
<td>92.543</td>
<td>0.324</td>
</tr>
<tr>
<td>2472</td>
<td>91.883</td>
<td>0.309</td>
<td>92.192</td>
<td>92.381</td>
<td>0.189</td>
</tr>
<tr>
<td>3745</td>
<td>91.893</td>
<td>0.276</td>
<td>92.169</td>
<td>92.726</td>
<td>0.557</td>
</tr>
<tr>
<td>2517</td>
<td>91.872</td>
<td>0.316</td>
<td>92.188</td>
<td>92.358</td>
<td>0.170</td>
</tr>
<tr>
<td>2490</td>
<td>91.897</td>
<td>0.316</td>
<td>92.213</td>
<td>92.454</td>
<td>0.241</td>
</tr>
</tbody>
</table>

| Averages         | 91.8824   | 0.3138      | 92.1962    | 92.492    | 0.296      |

Table 1 shows the data collected for this experiment. Notice that there is no significant difference between the total process time (user + system) and the observed wall clock time. This shows that our method of using a wall clock timer is at least as accurate as using the timing calls provided by Sun. We use the results of this test as the base case for the rest of the experiments in this section.

\textsuperscript{13}This process was necessary for two reasons. First, because the granularity of the clock on the Suns is in jiffies (1/60 of a second) and often appeared to off by more than that when using timing calls like gettimeofday. Second, we wanted a way to determine the perturbation caused by monitoring, not just the time the process spends executing, but also the time the monitor spends executing. We consider the cost of sending the two messages, somewhere around a 8-12 milliseconds, to have a negligible effect on the total process time. It is interesting to point out that the results using this method proved to be effective, i.e. the clocks were close in most cases.
VIII.1.1.2. Perturbation Caused Using Central Analysis

In this experiment central analysis is used in evaluating the condition. A simple flow of information is used. The flow of information is from the sensor to the local monitor and from the local monitor to the central monitor. This means that with every pass through the loop, a sensor embedded in the loop sends the current value of the monitored variable to the local monitor, which in turn forwards the value to the central monitor. The results from this experiment are shown in Table 2. There are two interesting things to note about the results. First, the program takes approximately 102.77 seconds of total execution time. This is approximately 10.57 seconds longer than the time it takes to execute using no monitoring. This says that sending a message costs the sender approximately 2.1 milliseconds per message (recall that the process generates 5000 messages each of which must be sent to the central monitor). The observed wall clock time difference, between doing central analysis and doing no analysis is approximately 24.071 seconds or 4.8 milliseconds per message. Table 2 shows the information per true event. As can be seen the overhead per true event (when the condition was actually true) is quite high, ranging from 9.1 milliseconds to 27.29 milliseconds per event. We define an event to be true when the condition has actually been met, i.e. when:

\[ \text{fired\_value} = 1. \]

The cost model for perturbation states that perturbation is dominated, in the network environment, by the cost of message sends. As we show in Section VIII.2, it takes approximately 3.2 milliseconds to do a send/receive on a local machine and

\[ ^{14} \text{In reality, since we are using the UDP protocol, messages get lost when buffers fill.} \]
approximately 4.3 milliseconds to do a send/receive between machines on the same subnet. We expect the perturbation, based on the cost model, to be approximately 37.5 seconds for 5000 events. The observed result was approximately 34.295 seconds. The difference between the observed and expected results is 3.205 seconds or approximately 9% difference. At first glance, it appears our cost model is insufficient. Upon further inspection it is shown that the cost model is correct. The differences are attributable to two things: system overheads not taken into account in the model, and lost messages. The first category, system overheads, includes things like the amount of time it takes to do a process swap\textsuperscript{15}, unix accounting procedures, messages received from the outside world, etc. The more important factor, for understanding the differences between the observed and expected results, is the number of messages that are lost. Using the UDP protocol, messages can be lost at a relatively high rate. Upon

\textsuperscript{15}Processes in Unix become eligible for swapping whenever a system call is performed. Messages sends/receives are considered system calls, thus a process can be swapped everytime it sends a message.
further inspection of the experiment it was found that on the average approximately 4600 of a possible 5000 events were logged by the resident monitor. Messages are lost because the resident monitor cannot keep up with the event generator program. Given that 4600 events are received, the cost model states that the total perturbation should be between 34.7 and 35.78 seconds. The reason we give a range of expected results is that it is impossible to accurately measure the total time spent sending a message that does not get received because of a system buffer overflow. We estimate the time to be between 2.1 milliseconds and 3.2 milliseconds for messages sent to the same machine. The difference between the observed and expected time is, using 35.78 seconds as the expected time, between 1.49 and seconds. This is a difference of approximately 4%. Using the expected result of 34.7 seconds the difference is .41 seconds or approximately 1%.

Recapping the results of this experiment, it shows that the cost model prediction and the observed result were within 4%. The other result shown is that the resident monitor cannot keep up with programs generating large amounts of sensor output.

VIII.1.1.3. Perturbation Caused by Local Analysis

In this experiment, we move analysis to the resident monitor. The cost model states that the perturbation should decrease when local analysis is performed, since fewer messages are generated by the local monitor. Table 3 shows the results of the new placement of analysis. As expected, perturbation decreases. The expected results are that the process should generate 5000 messages, at a cost of approximately 3.2 milliseconds per message, and the local monitor should generate "n" messages at a cost of 4.3 milliseconds per messages. For "n = 2505" the expected results is 26.7 seconds. The observed result is 27.21. The difference is .51 seconds or approximately
2%. Again, most of the differences are attributable to the causes mentioned in Section VIII.1.1.2. Fewer messages are lost, since the local monitor generates fewer messages. This results in a speed up to the local monitor, allowing it to better keep up with the event generating program.

Table 3: Results Using Local Analysis

<table>
<thead>
<tr>
<th>Events</th>
<th>User Time</th>
<th>Sys Time</th>
<th>Total Time</th>
<th>Wall Time</th>
<th>Difference diff/event</th>
<th>Standard Deviation</th>
<th>Diff Wall time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1256</td>
<td>92.440</td>
<td>10.441</td>
<td>102.881</td>
<td>118.046</td>
<td>13.168</td>
<td>0.000110</td>
<td>25.567</td>
</tr>
<tr>
<td>2466</td>
<td>92.239</td>
<td>10.588</td>
<td>102.856</td>
<td>119.255</td>
<td>16.438</td>
<td>0.000112</td>
<td>28.263</td>
</tr>
<tr>
<td>3795</td>
<td>92.202</td>
<td>10.425</td>
<td>102.628</td>
<td>123.000</td>
<td>20.372</td>
<td>0.000155</td>
<td>30.506</td>
</tr>
<tr>
<td>2523</td>
<td>92.263</td>
<td>10.665</td>
<td>102.928</td>
<td>119.492</td>
<td>16.544</td>
<td>0.000152</td>
<td>27.000</td>
</tr>
<tr>
<td>2505</td>
<td>92.455</td>
<td>10.418</td>
<td>102.874</td>
<td>119.711</td>
<td>16.837</td>
<td>0.000148</td>
<td>27.219</td>
</tr>
<tr>
<td>Average</td>
<td>92.340</td>
<td>10.491</td>
<td>102.831</td>
<td>119.601</td>
<td>16.670</td>
<td>0.000120</td>
<td>27.009</td>
</tr>
</tbody>
</table>

VIII.1.1.4. Perturbation Caused by Sensor Analysis

Our cost model states that perturbation is a minimum when analysis is performed in the sensor. In this experiment we move the analysis to the sensor. The expected result is that the process should incur a 2 message overhead (perturbation) for each event generated. Table 4 shows the values observed. For 1256 events, the expected perturbation is approximately 9.41 seconds. The observed result is 9.3776 seconds, a difference of .0324 seconds or about a .3% difference. The result here is more accurate as is expected. Since fewer messages are generated, fewer messages get lost and Unix has less chance to interfere with the experiment. It is worth noting here, though, that
the times used for message send/receive costs are average times, as is shown in Section VIII.2 and may vary in individual experiments.

Table 4: Results Using Sensor Analysis

<table>
<thead>
<tr>
<th>Events</th>
<th>User Time</th>
<th>Sys Time</th>
<th>Total Time</th>
<th>Wall Time</th>
<th>Difference</th>
<th>diff/event</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1248</td>
<td>91.780</td>
<td>2.917</td>
<td>94.697</td>
<td>101.870</td>
<td>7.173</td>
<td>0.005748</td>
<td>0.000479</td>
</tr>
<tr>
<td>2507</td>
<td>92.075</td>
<td>5.352</td>
<td>97.427</td>
<td>109.641</td>
<td>12.214</td>
<td>0.004972</td>
<td>0.000207</td>
</tr>
<tr>
<td>3742</td>
<td>92.087</td>
<td>7.958</td>
<td>100.045</td>
<td>118.234</td>
<td>18.186</td>
<td>0.004861</td>
<td>0.000173</td>
</tr>
<tr>
<td>2486</td>
<td>91.905</td>
<td>5.296</td>
<td>97.201</td>
<td>109.441</td>
<td>12.240</td>
<td>0.004924</td>
<td>0.000208</td>
</tr>
<tr>
<td>2498</td>
<td>92.239</td>
<td>5.382</td>
<td>97.621</td>
<td>109.744</td>
<td>12.133</td>
<td>0.004853</td>
<td>0.000282</td>
</tr>
<tr>
<td>Averages</td>
<td>92.017</td>
<td>5.381</td>
<td>97.3882</td>
<td>109.786</td>
<td>12.388</td>
<td>0.005051</td>
<td>0.000270</td>
</tr>
</tbody>
</table>

VIII.1.1.5. Conclusions Regarding the Prediction of Perturbation

The major conclusion from this set of experiments is that the cost model accurately captures the costs associated with perturbation. Costs, such as process swap time, do not significantly affect the total cost of perturbation in the network environment. If the cost model for perturbation is correct, we can state that the monitor can be somewhat predictive with regards to how much perturbation a program will incur if it is monitored. In order to be totally predictive, the monitor must know how many events are going to occur. It is not realistic to assume that the monitor will have access to the number of events that will be generated, since the number of events may be data dependent. Our model allows us to be somewhat predictive if we assume a certain number of events to be the average number of events generated. A second
conclusion is that as the number of events increases, the costs associated with all three placements of analysis converge. This is an expected result, since as more events occur, more messages are generated, even when using sensor analysis. Restated, if a condition is always true, then doing sensor analysis is no better than doing central analysis, since the sensor will always generate a message. In fact, if the condition is always true, then doing sensor analysis will be slightly worse than doing central analysis, since doing sensor analysis requires that the condition be checked in the sensor. This is shown in Figure 20.

Figure 20: Perturbation per Event
Figure 21 plots the total processing times for all four experiments. Note that the times when using central analysis is identical to the time using local analysis. This is to be expected, since in each case the process generates 5000 messages.
VIII.1.2. Effect of Analysis Placement on Perturbation

In this set of experiments we validate our analysis placement algorithm with respect to perturbation, i.e. the placement of analysis close to collection decreases the perturbation to the monitored program. As we mentioned above, analysis placement is an important part of the cost model for perturbation (and latency), so this set of experiments also demonstrates the validity of the cost model. We concentrate here on showing how perturbation decreases as analysis is placed closer to the location of collection.

For these experiments we use two identical processes, each running the sample event generator program. The condition being evaluated when the central analysis and local analysis is used is:

\[ \text{attribute}_1 > \text{attribute}_2 \]

where attribute1 is the variable \text{fire}_\text{d} \_\text{value} in process 1 and attribute2 is the same variable in process 2. In the experiment using a combination of sensor and local analysis, the condition being checked is:

\[ \text{attribute}_1 = 1 \mid \text{attribute}_2 = 1 \]

where the attributes are the same as in the central and local analysis cases.

VIII.1.2.1. Simple Analysis Placement Test Using No Monitoring

This is the base case used by this set of experiments for comparison. As we mentioned, there are now two processes, each of which is executing on the same machine. Table 5 shows the results from this set of experiments. In all the experiments in this section the probability of generating an event is fixed, and the number of events generated is approximately 2500 per process.
The two processes each take approximately 92.43 seconds to execute. The wall time observed by each is approximately 185.37 seconds. To calculate the total process time we sum the individual process times for each process. This results in a total processing time of 184.862 seconds. The difference between the wall clock time and the processing time, .0508 seconds is much less than 1% and thus negligible. This reaffirms the fact that our wall clock results in times that are as accurate as the times returned by the Sun.

Table 5: Base Case: Two Sensors

<table>
<thead>
<tr>
<th>system time</th>
<th>user time</th>
<th>total time</th>
<th>wall time</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.376</td>
<td>92.024</td>
<td>92.400</td>
<td>185.336</td>
<td>92.936</td>
</tr>
<tr>
<td>0.362</td>
<td>92.100</td>
<td>92.462</td>
<td>185.402</td>
<td>92.940</td>
</tr>
</tbody>
</table>

VIII.1.2.2. The Effect of Central Analysis Placement on Perturbation

In this experiment, central analysis is used. Table 6 shows the results from this run. Using the numbers in Table 6 we see that each process takes approximately 109.459 seconds to execute. This is an increase of approximately 17.40 seconds. The difference between the average wall clock time of the processes, when using central monitoring and the base case (i.e. using no monitoring), is 61.08 seconds. This
represents the total combined perturbation time for both processes. If our analysis placement algorithm is correct, this is the worst perturbation we should see. We use this result to show that a decrease in perturbation can be achieved by moving analysis closer to the source of collection.

Table 6: Central Analysis: Two Sensors

<table>
<thead>
<tr>
<th></th>
<th>Sys Time</th>
<th>User Time</th>
<th>Total Time</th>
<th>Wall Time</th>
<th>Wall-total</th>
<th>wall-base</th>
<th>wall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>28.299</td>
<td>81.056</td>
<td>109.355</td>
<td>246.373</td>
<td>137.018</td>
<td>61.037</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28.267</td>
<td>81.296</td>
<td>109.563</td>
<td>246.522</td>
<td>136.959</td>
<td>61.120</td>
<td></td>
</tr>
</tbody>
</table>

VIII.1.2.3. The Effect of Local Analysis Placement on Perturbation

In this experiment analysis is performed by the local monitor. The expected result, assuming our algorithm for analysis placement is correct, is that the perturbation in this experiment should be less than the perturbation in the previous experiment. The observed results are shown in Table 7. Note that the average wall clock difference decreases when local analysis is used. The decrease in perturbation is approximately 14 seconds. This result is expected since local placement of analysis should decrease perturbation.
Table 7: Local Analysis: Two Sensors

<table>
<thead>
<tr>
<th></th>
<th>Sys Time</th>
<th>User Time</th>
<th>Total Time</th>
<th>Wall Time</th>
<th>Wall - Total</th>
<th>Wall-Base</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>27.827</td>
<td>78.113</td>
<td>105.940</td>
<td>232.951</td>
<td>127.011</td>
<td>47.615</td>
</tr>
<tr>
<td></td>
<td>27.947</td>
<td>77.934</td>
<td>105.881</td>
<td>232.736</td>
<td>126.855</td>
<td>47.334</td>
</tr>
<tr>
<td>Averages</td>
<td>27.887</td>
<td>78.024</td>
<td>105.911</td>
<td>232.844</td>
<td>126.933</td>
<td>47.475</td>
</tr>
</tbody>
</table>

VIII.1.2.4. Analysis Placement Using Sensor and Local Analysis

For this experiment, a combination of sensor and local monitor analysis is used. The condition being checked is:

\[
\text{attribute1} = 1 \mid \text{attribute2} = 1.
\]

Sensor analysis is used to decrease the number of messages between the process and the local monitor. Local analysis is used to further reduce message traffic. Both these should result in a decrease in perturbation. Table 8 shows the numbers observed. Each process generates approximately 2500 events (messages), as opposed to the 5000 events (messages) each generated when central analysis was used. The local monitor generated approximately 5000 messages to the central monitor. The observed perturbation is approximately 40 seconds. This result is also expected, since sensor placement of analysis should decrease perturbation.
Table 8: Sensor and Local Analysis: Two Sensors

<table>
<thead>
<tr>
<th></th>
<th>sys time</th>
<th>User Time</th>
<th>Total Time</th>
<th>Wall Time</th>
<th>Wall-total</th>
<th>Wall-Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>14.115</td>
<td>84.213</td>
<td>98.328</td>
<td>226.32</td>
<td>127.992</td>
<td>40.984</td>
</tr>
<tr>
<td>2nd</td>
<td>14.168</td>
<td>83.625</td>
<td>97.793</td>
<td>225.69</td>
<td>127.897</td>
<td>40.288</td>
</tr>
<tr>
<td>Averages</td>
<td>14.1415</td>
<td>83.019</td>
<td>98.005</td>
<td>226.005</td>
<td>127.9445</td>
<td>40.635</td>
</tr>
</tbody>
</table>

VIII.1.2.5. Conclusions on the Effect Analysis Placement Has on Perturbation

The major conclusion from this set of experiments is that as analysis is moved closer to the place of collection, perturbation decreases. This implies that our analysis placement algorithm is correct with respect to perturbation, since the algorithm attempts to place analysis in sensors and local monitors as much as possible.

An unexpected result of these experiments, and the experiments on perturbation shown in the previous section, is that the user program does not have to generate too many messages before the underlying communication protocol begins to lose messages. Again, loss rates of 10% are not uncommon. Switching to a more reliable protocol decreases the unpredictability, but it also increases the amount of time a message send/receive takes.
VIII.1.3. Evaluation of the Latency Cost Model

In this section we evaluate the cost model for latency, showing that the cost model captures all the costs associated with latency. It is important to note that latency and perturbation are related. We mention this here because all the results obtained in the previous two sections also show that the cost model for latency is correct. To show that changes to the flow of information have the expected effect on latency we run experiments that change the flow of information.

VIII.1.3.1. Latency Achieved Using the Standard Flow of Information

In this experiment, we capture the latency associated with using a standard flow of information. For this experiment, analysis is performed at the sensor. We define latency as the amount of time between when an event occurs and when the central monitor becomes aware that it has occur (i.e. when the central monitor stores the event in its data structures).

One problem we had was in deciding how to measure latency, since the central monitor and the target program reside on a different nodes, and no central clock exists. To accurately measure latency, we decided to use a handshaking approach. When the sensor sends a message saying the event has occurred, it blocks and waits for a message saying that the first message has been received. In this experiment we send the message from the sensor to the local monitor and then to the central monitor. The central monitor then sends a message back to the local monitor which in turn sends a message back to the sensor. We take that round trip time and divide it in half. The result is an estimation of the latency. Using the handshaking method for measurements also has the benefit that messages will not be lost.
Table 9 shows the results from this experiment. The program executes in the same manner as described in previous experiments. The average latency of information using this flow is approximately 8.3 milliseconds. The expected result, given by the cost model, is approximately 7.9 milliseconds. The difference between the observed result and the expected result is .5 milliseconds or approximately 6%.

The reason for this difference is based on reasons mentioned above. Another reason for this difference is that this set of experiments was run on a loaded network, i.e. there were other users on the network, thus the communication costs increase due to collisions and contention.
VIII.1.3.2. Latency Achieved by Changing the Flow of Information

Here we show that by changing the flow of information, i.e. allowing sensors to send events directly to the central monitor, latency is decreased. We also show that this decrease in latency is consistent with the latency cost model.

In this experiment, the sensor sends information directly to the central monitor. The observed latency using this flow of information, shown in Table 10, is 5.2 milliseconds. The expected result, the cost of a message send, is 4.9 milliseconds. The difference between the observed and expected results is .3 milliseconds or approximately 6%. Again, this difference is attributable to the causes mentioned above.

Table 10: Latency Using Changed Flow

<table>
<thead>
<tr>
<th>Average Latency</th>
<th>Average Hits</th>
<th>Test Latencies</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.005154</td>
<td>2490</td>
<td>0.0051536</td>
<td>0.000057</td>
</tr>
</tbody>
</table>
VIII.1.3.3. Conclusions About the Latency Model

Most of the results given in the perturbation section can be applied to the latency of information, since latency and perturbation are closely related. The major conclusion is that the cost model for latency is correct. This section also provides shows that changing the flow of information has predictable results in terms of latency. This is important for two reasons. First, it shows the cost model can statically predict the latency of information. Second, it shows that optimizations are possible that decrease the latency of information. This is important since it allows the monitor to be tailorable to meet the needs of specific users.

VIII.1.4. Multiprocessor Environment

In this section we show that the cost models for perturbation holds in a multiprocessor environment, where shared memory is used instead of message passing. If the cost model for perturbation holds, then the cost model for latency will also hold, since the two are related. In this section we present the results of a series of experiments in which two different placements of analysis are used. In the first set of experiments, we use local monitor analysis. In the second set of experiments we use sensor analysis. Central analysis experiments are not used, since, as we shall see, their results are identical to those of the local analysis experiments.

It is worth mentioning here how the multiprocessor experiments are set up. The same event generating program introduced above is used for these experiments. The loop now executes 30000 times rather than 5000 times. This was done in an attempt to stabilize the results given. Both the local monitor and the event generator program are
part of the same task. Information is passed between sensors and the local monitor through the use of sensor records, as is the case in the network environment. In the multiprocessor environment, however, the sensors place the sensor records into a large circular queue of sensor records. The queue is large enough that records are not lost during the experiments. Synchronization is achieved using P & V operations. Access to the queue is through the commands enqueue and dequeue [6]. Dequeue requests are made by the monitor, and if the queue is empty, the monitor blocks and waits until the queue has an entry. Enqueue requests are made by the sensors. Accesses to the queue, using these mechanisms takes between 1.7 and 1.9 milliseconds.

VIII.1.4.1. Unmonitored Times Using the Multiprocessor

This section presents the results of the unmonitored version of the event generator program running on the Encore multiprocessor. Table 11 shows the results for the time taken to execute this base case. It is interesting to note that the base case program run on the Encore, which goes through approximately five times as many iterations as the same program on the Suns, takes 1201.797 seconds to execute. If the Sun version of the event generator is changed so it loops 30000 times, it would take approximately 554.8 seconds to complete. This shows that the processors used on the Encore are much slower (as much as twice as slow) as the processors used by the Suns.
Table 11: Encore: Base Case

<table>
<thead>
<tr>
<th></th>
<th>number of hits</th>
<th>clocktime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14951</td>
<td>1204.702649</td>
</tr>
<tr>
<td></td>
<td>14947</td>
<td>1208.078878</td>
</tr>
<tr>
<td></td>
<td>14989</td>
<td>1199.746770</td>
</tr>
<tr>
<td></td>
<td>15004</td>
<td>1208.756280</td>
</tr>
<tr>
<td></td>
<td>15081</td>
<td>1192.606381</td>
</tr>
<tr>
<td></td>
<td>15130</td>
<td>1202.126275</td>
</tr>
<tr>
<td></td>
<td>14982</td>
<td>1192.608040</td>
</tr>
<tr>
<td></td>
<td>15012</td>
<td>1201.818700</td>
</tr>
<tr>
<td></td>
<td>15085</td>
<td>1200.350468</td>
</tr>
<tr>
<td></td>
<td>14935</td>
<td>1207.173535</td>
</tr>
</tbody>
</table>

Averages 15012.6 1201.796798

VIII.1.4.2. Perturbation Achieved Using Local Analysis

In this set of experiments analysis is performed in the local monitor. The event generator program generates 30000 enqueue requests. The local monitor generates 30000 dequeue requests, and approximately 15000 messages get sent to the central monitor. In these experiments the central monitor runs on a Sun 3/50 workstation. The expected result, based on 30000 enqueue requests, is that the event generator should take anywhere from 51-57 seconds longer to execute. The observed result, 52.048, lies within the expected result. We do not consider the time to do the dequeue requests, since they are performed by the local monitor running on a different processor board. If both processes (the event generator and the local monitor) were both running on the same processor board, then the time it takes to do a dequeue, and the time it takes to send a message to the central monitor must be included in the estimation. Unfortunately, on the Encore multiprocessor, we do not have direct control over the scheduler, so each process runs wherever it is scheduled.
In general, this experiment shows that the observed result falls within the range of expected results. The process incurs approximately a 4% increase in total processing time by using local analysis.

Table 12: Encore: Using Local Monitor

<table>
<thead>
<tr>
<th>Number of Hits</th>
<th>Clocktime</th>
<th>clock - base</th>
<th>Time- hit</th>
<th>Standard Deviation</th>
<th>Predicted (1.7)</th>
<th>Predicted (1.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>14955</td>
<td>1261.546366</td>
<td>59.749566</td>
<td>0.003965</td>
<td></td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>14851</td>
<td>1253.841207</td>
<td>52.044409</td>
<td>0.003504</td>
<td></td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>14917</td>
<td>1257.428232</td>
<td>55.626026</td>
<td>0.003729</td>
<td></td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>15152</td>
<td>1244.898422</td>
<td>42.813044</td>
<td>0.002826</td>
<td></td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>15093</td>
<td>1253.269459</td>
<td>51.430145</td>
<td>0.003469</td>
<td></td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>15046</td>
<td>1251.511936</td>
<td>48.715138</td>
<td>0.003394</td>
<td></td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>14910</td>
<td>1256.919244</td>
<td>55.113446</td>
<td>0.003686</td>
<td></td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>15109</td>
<td>1247.606589</td>
<td>45.893791</td>
<td>0.003033</td>
<td></td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>15023</td>
<td>1257.336664</td>
<td>55.536866</td>
<td>0.003697</td>
<td></td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>15083</td>
<td>1254.353720</td>
<td>52.556922</td>
<td>0.003465</td>
<td></td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>Averages</td>
<td>15017.2</td>
<td>1253.844733</td>
<td>52.047036</td>
<td>0.003467</td>
<td>51.000000</td>
<td>57.000000</td>
</tr>
</tbody>
</table>

VIII.1.4.3. Perturbation Achieved Using Sensor Analysis

In this set of experiments analysis performed by the sensors. The expected result is a decrease in the number of enqueue requests, and thus a decrease in perturbation. Since only approximately 15000 enqueue requests are made, it is expected that the process should take an additional 25.5-28.5 seconds to execute. The observed result, 28.146 seconds, falls within this range.

In this experiment, as in the last experiment, the results showed what we expected. In this case, using sensor analysis increases the total processing time by approximately 2.2%.
Table 13: Encore: Using Sensor Analysis

<table>
<thead>
<tr>
<th>Number of Hits</th>
<th>Clocktime</th>
<th>clock - base</th>
<th>Time- hit</th>
<th>Standard Deviation</th>
<th>Predicted(1.7)</th>
<th>Predicted(1.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15036</td>
<td>1235.258248</td>
<td>33.461448</td>
<td>0.002225</td>
<td>25.566300</td>
<td>28.574100</td>
<td></td>
</tr>
<tr>
<td>15033</td>
<td>1225.53917</td>
<td>23.754119</td>
<td>0.001580</td>
<td>25.558100</td>
<td>28.552700</td>
<td></td>
</tr>
<tr>
<td>14870</td>
<td>1230.794110</td>
<td>28.992128</td>
<td>0.001950</td>
<td>25.279000</td>
<td>28.253000</td>
<td></td>
</tr>
<tr>
<td>15186</td>
<td>1241.365078</td>
<td>39.685278</td>
<td>0.002605</td>
<td>25.832000</td>
<td>28.872400</td>
<td></td>
</tr>
<tr>
<td>14619</td>
<td>1233.625564</td>
<td>31.630786</td>
<td>0.002134</td>
<td>25.362300</td>
<td>28.346100</td>
<td></td>
</tr>
<tr>
<td>15036</td>
<td>1237.569942</td>
<td>35.622144</td>
<td>0.002381</td>
<td>25.562000</td>
<td>28.556400</td>
<td></td>
</tr>
<tr>
<td>14870</td>
<td>1228.310027</td>
<td>26.513229</td>
<td>0.001775</td>
<td>25.392900</td>
<td>28.380300</td>
<td></td>
</tr>
<tr>
<td>15160</td>
<td>1227.267584</td>
<td>25.470858</td>
<td>0.001680</td>
<td>25.772000</td>
<td>28.804000</td>
<td></td>
</tr>
<tr>
<td>14889</td>
<td>1237.841651</td>
<td>38.048665</td>
<td>0.002421</td>
<td>25.311300</td>
<td>28.289100</td>
<td></td>
</tr>
<tr>
<td>14859</td>
<td>1229.729636</td>
<td>26.932377</td>
<td>0.001805</td>
<td>25.420100</td>
<td>28.410700</td>
<td></td>
</tr>
</tbody>
</table>

Averages 15008.7778 1233.070578 28.146403 0.001875 0.000349

VIII.1.4.4. Conclusions from the Multiprocessor Environment

The major conclusion we can draw is that the cost model for perturbation holds in the multiprocessor environment. This implies that the latency model holds, since the two are related. This validates the models for perturbation and latency, meaning the monitor can be predictive in this environment. Again, for perturbation, we assume an average number of events. The experimental results demonstrate three things. First, for multiprocessor environments where process scheduling is not controlled by the user, the cost model must be modified to include the fact that local analysis may or may not effect the performance of the target application (i.e. the local monitor may run on a separate processor, thus having no effect on the target application). In cases such as these, the models give a worst case estimation. The second thing these experiments show is that even in multiprocessor environments, placing of analysis close to the source of collection (i.e. in sensors) can improve the performance of the monitor. Finally, these experimental results show that in multiprocessor environments, the time
it takes to access shared data structures is more difficult to quantify. It appears to be more application dependent and related to the synchronization mechanisms used. An expansion of the cost model to include this fact might also be useful.

VIII.2. Measurements of Message Sends in the Network Environment

This section contains a table showing the costs associated with sending messages in the network environment. It is shown to give the reader a better understanding of communication costs.

Table 14: Message Sending Costs

<table>
<thead>
<tr>
<th>Location</th>
<th>Time in Seconds</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same Machine</td>
<td>0.003271</td>
<td>0.000043</td>
</tr>
<tr>
<td>Same Subnet</td>
<td>0.004109</td>
<td>0.000059</td>
</tr>
<tr>
<td>Different Subnet</td>
<td>0.010396</td>
<td>0.000635</td>
</tr>
</tbody>
</table>
VIII.3. Conclusions

There are two major conclusions that can be drawn from the results presented in this chapter. The first is that the experiments validate the cost models for perturbation and latency and show that the models capture the appropriate costs. This fact means that it is possible for our monitor to be predictive. This is an important result since it validates one of the goals of this thesis. The second result that can be drawn is that the algorithm for analysis placement is correct, i.e. placing analysis close to the source of collection has the expected results. This has been shown to be true both in a network environment and in a multiprocessor environment. This result is also important since it means that the monitor's performance can be changed by different placements of analysis. Also, the latency of results can be changed by changing the location of analysis.
In Chapter I we list the goals of this research. Restated, the goals are to investigate the problems associated with providing a real-time monitoring system that is predictive in nature, independent of hardware, language and application, and that can be tailored to meet specific performance requirements of users. We have shown how to construct such a monitoring system, and discussed the problems associated with doing so. In this chapter we discuss how each of these goals has been met, and summarize our experiences with the monitor. We also describe some improvements and future research associated with real-time monitoring.

IX.1. Independence

Our monitoring system has been shown to be application, machine and language independent. This independence is important since it allows the monitor to be used by a variety of users on different target machines.

To achieve this independence, we introduce constructs that allow users to interface with the monitor through the definition of monitoring views of their programs. To define views we introduced a model in which the user describes his program as a set of attributed entities and relationships. We demonstrated that it is
possible to use this model to describe a range of distributed and parallel software. Since views are defined in terms of these attributes, the views are dependent only on attributes. Changes, to the underlying objects, to the hardware, or to the programming languages used do not effect view definitions, unless the attributes of the object also change.

Attribute definitions and view definitions are automatically compiled into monitoring code. This implies ease of use of the system, since the user does not have to directly be involved with the generation of monitoring code. In order to do this compilation the view compiler requires information available from the programming environment like where the objects are to be executed. The compilation step is language, machine and operating system dependent, so the view compiler must be furnished information like: the programming language being used, the communication mechanisms available, etc. In this thesis we have shown that it is straightforward to generate different monitoring code for different target machines, thus demonstrating the independence of our approach.

IX.2. Predictive Capability

Our monitor has the capability to be somewhat predictive in nature, i.e. under certain conditions, it is possible for the monitor to statically determine whether a view specified by the user can be monitored and still meet the correctness constraints specified by the user. Toward this end, cost models were defined for two quantifiable costs associated with monitoring, perturbation and latency. We showed that these cost models are correct, both by intuitive argumentation and by experimentation. Experimentation also proved that these cost models accurately predict the perturbation and latency in both a network and a multiprocessor environment. To be totally
predictive, the monitor would need to know the number of events that occur, N in Equations (V.1) and (V.2). Since it is not realistic to assume the monitor knows the number of events that occur it is impossible for the monitor to be totally predictive. Given that we assume the number of events will be low, however, we can make the monitor somewhat predictive in nature.

IX.3. Performance

We demonstrated two things about the performance of our monitor. First, we showed that the performance of the monitor is predictable from the cost models for perturbation and latency. Second, we showed that in environments where communication costs are high, the placement of analysis close to the source of collection are essential to providing an efficient monitoring system. We showed that in environments with lower communication costs, the placement of analysis close to the source of collection increases the overall performance of the monitor and decreases the potential for bottlenecks.

We presented extensions to view definitions that allow users to provide the monitor with information concerning user specific correctness and performance constraints. Using these constraints, the monitor is then able to generate a monitoring configuration that is capable of meeting these constraints.

The monitor has been used to perform a number of real-time adaptations. This demonstrates the monitor's capability of performing efficiently in real-time environments.
IX.4. Tool Integration

Integration with other tools in a real-time environment is important. Our monitor has been integrated with a number of tools using an extended entity relationship data model. This approach to tool integration allows the monitor to share information in a consistent manner with other tools. The extension of action routines to the data model allows the monitor to interact with other tools within the confines of the database. This allows for a uniform interface model between the monitor and other tools in the programming environment. This method of tool integration allows the tools to be added to and removed from the programming environment without changing the monitor.

Displaying monitoring information to the user is important. We showed why views are a logical choice for the unit of display to the user. We demonstrated the infeasibility of displaying all information to the user, and how that approach changes the real-time characteristics of the monitor. We also suggested the concept of adding graphical display information to view definitions. This allows the specification of how a view is to be displayed to be kept separate from the implementation details of how a view actually gets displayed. This increases the independence of view definitions.
IX.5. Extensions to Our Research

There are many areas of possible extensions to our research, and many new interesting research ideas that have resulted from this work.

One possible extension concerns view definition, where two modifications are possible. The first is to expand view definitions to allow users to define their own performance models. This expansion would allow the user to more precisely define performance constraints. The expansions might include statements that allow users more control over how the monitor attempts to optimize collection and analysis. For instance, it might allow the user to specify that all the analysis associated with a specific view should be performed on one specific machine. This would be especially useful when the monitor is providing information for a tool that is distributed and has its own optimization algorithms, like as the adaptation controller. It requires that the compiler includes more view information in its generation phase. A second modification to view definitions is to allow for temporal operators, such as BEFORE and AFTER. This would be of particular use in real-time systems where users wish to use time ordering when doing specification. For instance, a user might wish to define a view that is true when event 'A' occurs before event 'B'. Both these modifications increase the power of the monitor and aid in the overall usability of the monitoring system.

Modifying the cost models is also an area of possible further research. Modifications are necessary that allow the models to account for interference caused by monitoring, including the interference caused by messages generated by the monitor. This may be a difficult modification since interference is somewhat
application dependent. This may require dynamic evaluation of the models. Another extension to the cost models is to more accurately capture what happens on multiprocessors where multiple machines are used and interference may or may not occur due to local analysis. Again, this may require dynamic evaluation of the models. This would allow the monitor to be more accurate in its predictive quality.

Another research area of interest is to allow dynamic modifications to the monitor. This includes dynamically switching between sampling and tracing and dynamically changing what analysis is being performed. It involved the construction of a self adapting monitor system. This would be interesting since statically determining the best possible monitoring configuration is impossible, unless all data and performance aspects of the system are known.

Another interesting area of research is in the distribution of the database. We suggested some possible distributions in Chapter VI. It would be interesting to see if such distributions were possible and how the performance of the monitor benefits because of these distributions.

One final area of possible research is the application of mapping algorithms [30, 32, 5] to the analysis placement algorithm. This would allow the monitor to work efficiently in architectures, like hypercubes, where certain communications are much less expensive than others. It would also be interesting to create a mapping algorithm for the topology of networks that exists at Ohio State.
This appendix lists the non-terminals, the terminals and the classes for the attribute definition and the view definition ALOEs.

A.1. Grammar for Attribute Definitions

This section contains the grammar for the attribute definitions.
Non-Terminals:

PROGRAM =
  <template> | daemon: <none> | synonym: <none> |
  precedence: <none> | scope: TRUE | infop: 3 |
  attributes:
    <none>
  unparsing:
    (0) traproot: FALSE | trapdoor: <none>
         "@0@n@q@n"
    (2) traproot: FALSE | trapdoor: <none>
         "@0"

SINGLE =
  objname | daemon: <none> |
    synonym: <none> |
  precedence: <none> | scope: FALSE | infop: 19 |
  attributes:
    <none>
  unparsing:
    (0) traproot: FALSE | trapdoor: <none>
         "@1"
    (2) traproot: FALSE | trapdoor: <none>
         "@1 #"
MULTIPLE =
  objname object_name | daemon: <none> |
   synonym: <none> |
  precedence: <none> | scope: FALSE | infop: 20 |
  attributes: <none>
  unparsing: (0,2) traproot: FALSE | trapdoor: <none>
   "@1 @2"

OBJLIST =
  object_name | daemon: <none> |
   synonym: <none> |
  precedence: <none> | scope: FALSE | infop: 18 |
  attributes: <none>
  unparsing: (0) traproot: FALSE | trapdoor: <none>
   "@1"
  (2) traproot: FALSE | trapdoor: <none>
   "@1"

VAR =
  var | daemon: <none> | synonym: <none> |
  precedence: <none> | scope: FALSE |
   infop: 14 |
  attributes: <none>
  unparsing: (0) traproot: FALSE | trapdoor: <none>
   "@1"
TEMPLATE =
    entity_name entity_type oblist attrs |
    daemon: <none> | synonym: <none> |
    precedence: <none> | scope: FALSE | infop: 11 |
    attributes:
        <none>
unparsing:
    (0) traproot: FALSE | trapdoor: <none>
        "ATTRIBUTE DEFINITION @1
        @nTYPE @2 @nFOR OBJECT
        @3@n@4@nEND ATTRIBUTE DEFINITION"
    (2) traproot: FALSE | trapdoor: <none>
        "@1@n@2@n@3@n@4#@n"

ATTRS =
    <attrdefn> | daemon: <none> | synonym: <none> |
    precedence: <none> | scope: FALSE | infop: 13 |
    attributes:
        <none>
unparsing:
    (0) traproot: FALSE | trapdoor: <none>
        "@0@n"
    (2) traproot: FALSE | trapdoor: <none>
        "@0@n"

ATTRDEFN =
    type attrname definition | daemon: <none> |
    synonym: <none> |
    precedence: <none> | scope: FALSE | infop: 7 |
    attributes:
        <none>
unparsing:
    (0) traproot: FALSE | trapdoor: <none>
        "( @1 ) @2 : @3"
    (2) traproot: FALSE | trapdoor: <none>
        "@1 @2 @3"
DEFINITION =
exp | daemon: <none> | synonym: <none> |
precedence: <none> | scope: FALSE | infop: 8 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"@1"
(2) traproot: FALSE | trapdoor: <none>
"@1#"

PLUS EXP =
exp exp | daemon: <none> | synonym: <none> |
precedence: <none> | scope: FALSE | infop: 9 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"@1 + @2"
(2) traproot: FALSE | trapdoor: <none>
"@1 + @2"

MINUS EXP =
exp exp | daemon: <none> | synonym: <none> |
precedence: <none> | scope: FALSE | infop: 10 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"@1 - @2"
(2) traproot: FALSE | trapdoor: <none>
"@1 - @2"
Classes:

attrdefn = ATTRDEFN
attrname = STRING
definition = DEFINITION
exp = PLUSEXP MINUSEXP CONST VAR
template = TEMPLATE
objname = OBJNAME
objlist = OBJLIST
attrs = ATTRS
type = STRING
entity_name = STRING
var = STRING
object_name = MULTIPLE SINGLE
entity_type = STRING
Terminals:

STRING =
{rep} | daemon: <none> | synonym: <none> |
lex: lexstring | infop: 4 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"@v"

CONST =
{rep} | daemon: <none> | synonym: <none> |
lex: lexinteger | infop: 5 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"@v"

OBJNAME =
{rep} | daemon: <none> | synonym: <none> |
lex: lexstring | infop: 12 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"@v"
A.2. Grammar for View Definition

In this section we present the grammar used to define views.

Non-Terminals:

\[ \text{VIEWS} = \]
\[ <v\_def> | \text{daemon: <none> | synonym: <none> | precedence: <none> | scope: TRUE | infop: 3 | attributes: <none> | unparsing: } \]
\[ (0) \text{traproot: FALSE | trapdoor: <none> "@0@q@n@n@n" } \]
\[ (2) \text{traproot: FALSE | trapdoor: <none> "@0" } \]

\[ \text{NOT\_TOTALLY\_CORRECT} = \]
\[ \text{Int\_correctness | daemon: <none> | synonym: <none> | precedence: <none> | scope: FALSE | infop: 26 | attributes: <none> | unparsing: } \]
\[ (0) \text{traproot: FALSE | trapdoor: <none> " correct to within (@1 milliseconds)" } \]
\[ (2) \text{traproot: FALSE | trapdoor: <none> "# @1 " } \]

\[ \text{PERFORMANCE} = \]
\[ \text{perf\_value | daemon: <none> | synonym: <none> | precedence: <none> | scope: FALSE | infop: 27 | attributes: <none> | unparsing: } \]
\[ (0) \text{traproot: FALSE | trapdoor: <none> "@n within (@1 milliseconds)" } \]
\[ (2) \text{traproot: FALSE | trapdoor: <none> "@1 " } \]
V_DEF =
  view number view_type activity_period
  action return | daemon: <none> |
  synonym: <none> |
  precedence: <none> | scope: FALSE |
  infop: 12 |
  attributes:
    <none>
  unparsing:
    (0) traproot: FALSE | trapdoor: <none>
      "@nVIEW DEF @1 OF TYPE @2@n
      @3@n @4@n @5@nEND VIEW DEF@n"
    (2) traproot: FALSE | trapdoor: <none>
      "@1 @2@n@3@n@4@n@5@n"

ACTIVITY_PERIOD =
  exp correctness_value | daemon: <none> |
  synonym: <none> |
  precedence: <none> | scope: FALSE | infop: 13 |
  attributes:
    <none>
  unparsing:
    (0) traproot: FALSE | trapdoor: <none>
      "@1@n@2"
    (2) traproot: FALSE | trapdoor: <none>
      "@1 @2"

CONST_EXP =
  hostname proc_id attribute_name op const |
  daemon: <none> | synonym: <none> |
  precedence: <none> | scope: FALSE | infop: 14 |
  attributes:
    <none>
  unparsing:
    (0) traproot: FALSE | trapdoor: <none>
      "(0 1 0 2 0 3 0 4 0 5 )"
    (2) traproot: FALSE | trapdoor: <none>
      "c# @1 @2 @3 @4 @5 " 
VAR_EXP =

    hostname proc_id attribute_name op hostname
    proc_id attribute_name | daemon: <none> |
    synonym: <none> |
    precedence: <none> | scope: FALSE | infop: 15 |
    attributes:
        <none>
    unparsing:
        (0) traproot: FALSE | trapdoor: <none>
            "(01 @2 @3 @4 @5 @6 @7)"
        (2) traproot: FALSE | trapdoor: <none>
            "#v @1 @2 @3 @4 @5 @6 @7"

ACTION =

    command proc_id hostname port_number
    performance |
    daemon: <none> | synonym: <none> |
    precedence: <none> | scope: FALSE | infop: 16 |
    attributes:
        <none>
    unparsing:
        (0) traproot: FALSE | trapdoor: <none>
            "@1 @2 @3 @4 @5"
        (2) traproot: FALSE | trapdoor: <none>
            "@1 @2 @3 @4 @5"

RET_EXP =

    ret_exp | daemon: <none> | synonym: <none> |
    precedence: <none> | scope: FALSE | infop: 17 |
    attributes:
        <none>
    unparsing:
        (0) traproot: FALSE | trapdoor: <none>
            "RETURN @1"
        (2) traproot: FALSE | trapdoor: <none>
            "@1"
COMP_EXP =
  exp binop exp | daemon: <none> |
  synonym: <none> |
  precedence: <none> | scope: FALSE | infop: 18 |
  attributes:
    <none>
  unparsing:
    (0,2) traproot: FALSE | trapdoor: <none>
      "@1 @2 @3"

RET_EXP1 =
  attribute_type attribute_name |
    daemon: <none> |
    synonym: <none> |
  precedence: <none> | scope: FALSE | infop: 19 |
  attributes:
    <none>
  unparsing:
    (0) traproot: FALSE | trapdoor: <none>
      "(@1 @2)"
    (2) traproot: FALSE | trapdoor: <none>
      "@1 @2 #"

RET_EXP2 =
  attribute_type attribute_name ret_op ret_exp |
    daemon: <none> | synonym: <none> |
  precedence: <none> | scope: FALSE |
  infop: 20 |
  attributes:
    <none>
  unparsing:
    (0) traproot: FALSE | trapdoor: <none>
      "(@1 @2) @3 @4"
    (2) traproot: FALSE | trapdoor: <none>
      "@1 @2 @3 @4"
Classes:

view_number =
    INT_CONST

activity_period =
    ACTIVITY_PERIOD

action =
    ACTION

return =
    RET_EXP

exp =
    CONST_EXP VAR_EXP COMP_EXP

proc_id =
    INT_CONST

attribute_name =
    STRING
op =
    GTHAN LTHAN EQ

const =
    INT_CONST

command =
    NOTIFY RETRIEVE NOTIFY_BEGIN_AND_END

hostname =
    STRING

port_number =
    INT_CONST

ret_exp =
    RET_EXP1 RET_EXP2

binop =
    AND OR

attribute_type =
    STRING

ret_op =
    COMMA

v_def =
    V_DEF

correctness_value =
    TOTAL_CORRECTNESS NOT_TOTALLY_CORRECT

int_correctness =
    INT_CORRECTNESS

performance =
    PERFORMANCE

perf_value =
    INT_CONST

view_type =
    STRING
Terminals:

STRING =
    {rep} | daemon: <none> | synonym: <none> |
    lex: lexstring | infop: 4 |
    attributes:
        <none>
    unparsing:
        (0) traproot: FALSE | trapdoor: <none>
            "@v"

TOTAL_CORRECTNESS =
    {static} | daemon: <none> | synonym: <none> |
    lex: <none> | infop: 24 |
    attributes:
        <none>
    unparsing:
        (0) traproot: FALSE | trapdoor: <none>
            "Correct to within (5 microseconds)"
        (2) traproot: FALSE | trapdoor: <none>
            "# 5"

INT_CORRECTNESS =
    {rep} | daemon: <none> | synonym: <none> |
    lex: <none> | infop: 25 |
    attributes:
        <none>
    unparsing:
        (0) traproot: FALSE | trapdoor: <none>
            "@v"
RETRIEVE =
{static} | daemon: <none> | synonym: <none> |
lex: <none> | infop: 23 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"RETRIEVE"

NOTIFY =
{static} | daemon: <none> | synonym: <none> |
lex: <none> | infop: 28 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"NOTIFY"

NOTIFY_BEGIN_AND_END =
{static} | daemon: <none> | synonym: <none> |
lex: <none> | infop: 29 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"NOTIFY_BEGIN_AND_END"
INT_CONST =
    {rep} | daemon: <none> | synonym: <none> |
    lex: lexinteger | infop: 5 |
    attributes:
        <none>
    unparsing:
        (0) traproot: FALSE | trapdoor: <none>
        "@v"

LTHAN =
    {static} | daemon: <none> | synonym: <none> |
    lex: <none> | infop: 6 |
    attributes:
        <none>
    unparsing:
        (0) traproot: FALSE | trapdoor: <none>
        "<@

GTHAN =
    {static} | daemon: <none> | synonym: <none> |
    lex: <none> | infop: 7 |
    attributes:
        <none>
    unparsing:
        (0) traproot: FALSE | trapdoor: <none>
        "@>"
EQ =
{static} | daemon: <none> | synonym: <none> |
lex: <none> | infop: 8 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"=="

AND =
{static} | daemon: <none> | synonym: <none> |
lex: <none> | infop: 9 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"&@n"

OR =
{static} | daemon: <none> | synonym: <none> |
lex: <none> | infop: 10 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
"|@n"

COMMA =
{static} | daemon: <none> | synonym: <none> |
lex: <none> | infop: 11 |
attributes:
<none>
unparsing:
(0) traproot: FALSE | trapdoor: <none>
","@n"
(2) traproot: FALSE | trapdoor: <none>
"@n"
APPENDIX B
Database Operations

This appendix contains a list of common database operations. Database operations are described using a C-like syntax. They can be grouped into:

General Operations on Database Entries: Entities, relationships and sets are registered with the database process using the _cdb_template_reg construct:

_cdb_template_reg("type_def","selector",
"containslist","relateslist",
"attributeslist",ActionRoutineName)

The type_def of a previously created database entry can be retrieved with the _cdb_ent_gettemplate call.

_cdb_ent_gettemplate(EntityName)
where:
EntityName: This is the name returned when the function _cdb_ent_create is invoked.

In any definition the relateslist is valid only when defining a relationship, while the containslist is valid only when defining a set. For example the following code fragments show the template registration calls for an entity, a relationship and a set respectively.
Operations on Entities: Two operations common to all database entries are the _cdb_ent_create and the _cdb_ent_delete operations. The syntax for these calls is as follows:

```c
_cdb_ent_create("type_def","ExtraAttributes");
```

where:

type_def: Must be previously registered using a _cdb_template_reg() call.

ExtraAttributes: List of any instance-specific attributes (if any).

```c
_cdb_ent_delete("EntityName");
```

where:

EntityName: Name of a previously created entity.

Again, this is the result of a _cdb_ent_create() function call.

Operations on Attributes: The two primary operations on attributes are the _cdb_get_attr and the _cdb_set_attr operations. The syntax of these constructs is as follows:
cdb_get_attr("EntityName","AttributeName",Index)
where:
EntityName: A previously created entry name.
AttributeName: Name of the attribute.
Index: Integer offset if AttributeName represents an array.

cdb_set_attr("EntityName","AttributeName",Index)
where:
EntityName: A previously created entity name.
AttributeName: Name of the attribute.
Index: Integer offset if AttributeName represents an array.

Operations on Relationships: Operations that are unique to relationships are the cdb_set_link and the cdb_ent_get. The cdb_ent_setlink() call is used to set up a link from the relationship to a component entity.

cdb_ent_setlink(R_Name, "FieldName", TargetEntity)
where:
R_Name: A previously created relationship. This is analogous to EntityName.
FieldName: Field to which to link.
TargetEntity: A previously created entity that is to be linked in.

The cdb_ent_get() call returns the identifier of the entity that is linked to the component field of a relationship. The call is as follows:

cdb_ent_get(StartEntity, PtrChain)
where:
StartEntity: A previously created relationship.
PtrChain: A string of the form "FieldName->"

Operations on Sets: The operations defined on sets allow initialization, creation, element access, union, intersection, diff, and assignment.

1. Initialize a set and sets its contents to NULL.
   cdb_set_init()

2. Return True if Element is in SetName.
_cdb_set_inset(SetName, Element)
where:
SetName: Previously created with a
        _cdb_ent_create() call.
Element: Previously created entry.

3. Return a pointer to an element in the SetName. The call is useful for
   examining elements of a set in sequence.
   _cdb_set_removeelt( ref(SetName) )
where:
SetName: A previously created set.

4. Create a new set that is the union of SetName1 and SetName2.
   _cdb_set_union(SetName1, SetName2)
where:
SetName1: A previously created set.
SetName2: A previously created set.

5. Create a new set that is the intersection of SetName1 and SetName2.
   _cdb_set_inter(SetName1, SetName2)
where:
SetName1: A previously created set.
SetName2: A previously created set.

6. Create a new set that is the difference of SetName1 and SetName2.
   _cdb_set_diff(SetName1, SetName2)
where:
SetName1: A previously created set.
SetName2: A previously created set.

7. Create a new set that contains all elements of SetName1
   _cdb_ent_ALL(SetName1)
where:
SetName1: A previously created set.

8. Return a new set whose elements are all of type "type_def".
   _cdb_sch_ALL(type_def)
where:
type_def: A previously registered entry.

9. Return a new singleton set that has EntityName as its only element.
   _cdb_set_makeset(EntityName)
where:
EntityName: A previously created entity.
10. Assign all elements in SetName to the type_def.

\[ \text{\_cdb\_set\_assign(type\_def, SetName)} \]

where:

- type\_def: A previously created type\_def.
- SetName: A previously created set.

This appendix contains a list of the attributes that are monitored in the RTDM project.

1. bytesread: This contains the number of bytes read since the previous clock TIC.

2. ndpackets: This is the number of ND packets detected.

3. ndlen: The accumulative length of the ND packets detected.

4. icmppackets: The number of ICMP packets detected.

5. ippackets: The number of IP packets detected.

6. totpackets: Total number of packets detected.

7. unknocnt: Number of unknown packets detected.

8. unknolen: Accumulative length of the unknown packets.

9. arppackets: The number of ARP packets detected.

10. revarppackets: The number of REVARP packets detected.

11. ippacklen: Accumulative IP packets length.

12. ipmaxlen: The maximum length of IP packets detected until now.

13. lenipin: The accumulative length of the IP packets came in our subnet through the gateway. Of course here only one gateway is taken into consideration but usually here every SUN subnet has only one gateway:

14. noipin: The number of IP packets came in our subnet through the gateway. (see lenipin).

15. lenipout: The accumulative length of IP packets went out of our subnet through the gateway. (see lenipin).
16. noipout: The number of IP packets went out of our subnet through the gateway. (see lenipin).

17. tcp packets: The number of TCP packets detected.

18. tcp len: The accumulative length of TCP packets detected.

19. ftp packets: The number of FTP packets detected.

20. ftp len: The accumulative length of FTP packets detected.

21. telnet packets: The number of TELNET packets detected.

22. telnet len: The accumulative length of TELNET packets detected.

23. print packets: The number of PRINTER packets detected.

24. print len: The accumulative length of PRINTER packets detected.

25. shell packets: The number of SHELL packets detected.

26. shell len: The accumulative length of SHELL packets detected.

27. trpc packets: The number of SUN RPC (TCP) packets detected.

28. trpc len: The accumulative length of SUN RPC (TCP) packets detected.

29. udp packets: The number of UDP packets detected.

30. udp len: The accumulative length of UDP packets detected.

31. urpc packets: The number of SUN RPC (UDP) packets detected.

32. urpc len: The accumulative length of SUN RPC (UDP) packets detected.

33. route packets: The number of ROUTE (UDP) packets detected.

34. route len: The accumulative length of ROUTE (UDP) packets detected.

35. nobroadcip: The number of broadcasted IP packets.

36. lenbroadcip: The length of the broadcasted IP packets.
References

A Methodology for Interactive Computer Service Measurement.  

An Approach to High Level Debugging of Distributed Systems.  

Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior.  

The Entity-Relationship Model - Toward a Unified View of Data.  

Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems.  

Supporting Objects Invocations on Networks and Multiprocessors.  

Bugnet: A Debugging System for Parallel Programming Environments.  

[8] Fitch, J.  
Profiling a Large Program.  

Views for Tools in Integrated Environments.  

Graphical Views of Parallel Programs.  
1987.

*Programming With Objects in the Real-Time Domain*.  
[12] Paul Harter, Dennis Heimbigner and Roger King.  
IDD: An Interactive Distributed Debugger. 
In Proceedings of the 5th International Conference on Distributed Systems.  

Software Development Tools.  

Monitoring Distributed Systems.  

Semantics for Structure Editing Environments.  

[16] David Alex Lamb.  
Sharing Intermediate Representations: The Interface Description Language.  
PhD thesis, Computer Science Department, Carnegie-Mellon University,  

Distributed Snapshots: Determining Global States of Distributed Systems.  

[18] Leslie Lamport.  
Synchronizing Time Servers.  

[19] Leblanc, T., Mellor-Crummey, J.  
Debugging Parallel Programs With Instant Replay.  

Implementing Relational Views of Programs.  

Supporting Reverse Execution for Parallel Programming.  
In Proceedings of the Workshop on Parallel and Distributed Debugging.  

An Incremental Programming Environment.  

[23] Barton Miller and Cui-Qing Yang.  
IPS: An Interactive and Automatic Performance Measurement Tool for  
Parallel and Distributed Systems.  
*Breakpoints and Halting in Distributed Programs.*  

The GANDALF Project.  

[26] Nutt, G.J.  
*A Survey of Remote Monitors.*  

An Intelligent Bus Tracer.  

Monitoring Program Execution: A Survey.  

[29] Rajiv Ramnath.  
*The Programming of Adaptable Parallel Software.*  

Nearest-Neighbor Mapping of Finite Element Graphs onto Processors Meshes.  

[31] Satterthwaite, E.  
Debugging Tools for High Level Languages.  

*Tailoring Software for Multiple Processor Systems.*  

*An Integrated Programming/Operating System for Parallel Programming.*  

[34] Dave Ogle, Karsten Schwan, and Richard Snodgrass.  
*The Real-Time Collection and Analysis of Dynamic Information in a Distributed System.*  
A System for Parallel Programming. 
In 9th International Conference on Software Engineering, Monterey, CA, 
Awarded best paper.

Graphical Views of Parallel Programs. 
Also available as technical report, Department of Computer and Information 
Science, Ohio State University, OSU-CISRC-TR-85-11.

The Display of Temporal Information. 

[38] Richard Snodgrass. 
In Proceedings of the Software Engineering Symposium on Practical Software 
SIGSOFT/SIGPLAN, April, 1984.

Filtering Monitoring Data. 
Available as technical report TR86-001, UNC-CSD.

[40] Richard Snodgrass. 
A Relational Approach to Monitoring Complex Systems. 
Also available as TR85-035 from the Department of Computer Science, Univ. 
of North Carolina at Chapel Hill.

[41] Richard Snodgrass. 
Monitoring Distributed Systems. 

The Temporal Query Language TQUEL. 

[43] Unger, B., Birtwistle, G., Cleary, J., Dewar, A. 
A Distributed Software Prototyping and Simulation Environment: Jade. 
In Proceedings of SCS Conference on Intelligent Simulation Environments. 

*Dynamic Improvements in the Performance of Parallel and Distributed Software.*  

[46] Waite, W. M.  
A Sampling for Applications Programs.  

[47] Yuval, G.  