Providing Intrinsic Support for User Interface Monitoring

by

Jolly Chen

B.S., Massachusetts Institute of Technology (1989)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1990

© Jolly Chen, 1990

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

Signature redacted

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
May 11, 1990

Signature redacted

Certified by ______________________________
Robert W. Scheifler
Principal Research Associate, Laboratory of Computer Science
Thesis Supervisor

Signature redacted

Certified by ______________________________
David Pollock
Project Manager, Hewlett-Packard
Thesis Supervisor

Signature redacted

Accepted by ______________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

AUG 10 1990

LIBRARIES

ARCHIVES

Providing Intrinsic Support for User Interface Monitoring

by

Jolly Chen

Submitted to the Department of Electrical Engineering and Computer Science
on May 11, 1990, in partial fulfillment of the
requirements for the degrees of

Master of Science in Electrical Engineering and Computer Science

Abstract

Sophisticated graphical user interfaces are both difficult and time consuming to create. Researchers now believe that user interfaces must be designed iteratively. Iterative design involves constructing prototypes of user interfaces, evaluating those prototypes, and iterating the process until an acceptable interface is reached. Evaluation of user interfaces requires effective means of recording and analyzing human-computer interactions.

This thesis examines how monitoring mechanisms can be added to user interface architectures to provide intrinsic support for recording human-computer dialogue. The key to this approach is inspection of the communication channels within an interactive program. Dialogue information recorded by intrinsic monitoring differs from dialogue data obtained from low level input capture and high level application-specific recording schemes. Intrinsic monitoring is shown to be useful for answering many of the questions that arise in the evaluation of a user interface. The use of a built-in monitoring mechanism also allows an interactive program to be monitored without modifying application source code.

The Xt Intrinsics of the X Window System was modified as a sample implementation of this monitoring approach. In addition, a tool was developed to analyze the recorded dialogue data.

Thesis Supervisor: Robert W. Scheifler
Title: Principal Research Associate, Laboratory of Computer Science

Thesis Supervisor: David Pollock
Title: Project Manager, Hewlett-Packard

Acknowledgments

I would like to express my thanks to my advisor Bob Scheifler for his guidance and patience in reading over drafts of this thesis. I am also grateful to Dave Pollock, my advisor at HP, for his support of my efforts, especially his willingness to let me pursue my research interests during my co-op assignment at HP. I would also like to express my thanks to Howard Sumner, John Marold, Frank Richichi, and other members of the Clinical Information Systems team at HP for their help and friendly support over the years. I wish also to thank Chris Peterson, Donna Converse, and Ralph Swick for the feedback they gave me during the formulation of my topic and for answering many of my questions about X. I would like to thank Dave Duis and Judy Chen for their humor and wit throughout this whole ordeal. (Isn't it amazing we actually finished?)

In addition, I wish to thank my family and friends for their steadfast love and encouragement. I am thankful most of all for the love and grace of my Lord Jesus Christ and for His gentle reminders of the things eternal and the futility of things temporal.

Contents

1 Overview 1

2 Introduction 4

2.1 Problem: building good user interfaces 4

2.2 Structural model of user interfaces 5

2.3 Iterative design 7

2.4 An evaluation example 8

2.5 The focus of this thesis 10

2.6 Related work 10

2.6.1 Specification of user interfaces 11

2.6.2 User interface monitoring tools 12

3 Dialogue recording 15

3.1 Incidents 15

3.2 Questions to be answered 16

3.3 Need for intrinsic monitoring 16

4 Instrumenting the architecture 18

4.1 Architectural requirements 18

4.1.1 Separation of user interface and application 18

4.1.2 Accessible communication channels 19

4.1.3 Identification of incidents 20

4.2 Implementation requirements 20

4.3 Providing intrinsic monitoring 21

5 Implementation 22

5.1 Xt Intrinsics 22
5.2 A modified Xt Intrinsics 26
5.3 Analysis tool 30

6 Example of use – Xmh **32**
6.1 Instrumenting Xmh 32
6.2 Reasoning about the interface 32

7 Conclusions **35**
7.1 Contributions 35
7.2 Limitations and future work 35

List of Figures

2-1 A simple model of an interactive program 5

2-2 Logical model of a user interface 6

2-3 xmh user interface 8

2-4 Targets of user interface recording 10

5-1 The software structure of Xt Intrinsics-based applications 23

5-2 The logical structure of Xt-based user interfaces 25

5-3 Analysis tool interface 31

Chapter 1

Overview

The goal of this thesis is to demonstrate that user interfaces can be effectively monitored by adding appropriately designed recording mechanisms to the underlying software architecture. The thesis shows how dialogue data recorded by intrinsic monitoring mechanisms help answer many of the questions a human evaluator asks when assessing an interface. The main emphasis of this research is to suggest a new strategy by which dialogue information can be collected. The work does not attempt to address all the issues surrounding how well a user interface realizes a particular user model.

The motivation for this new monitoring strategy is the inadequacy of current monitoring approaches. Current monitoring schemes record dialogue information at one of two extremes: low level input capture or high level application-specific recording. Low level input capture yields information that is not at a high enough level to be useful in reasoning about the user interface. Application-specific recording forces the application programmer to carry the burden of adding hooks for instrumentation. In addition, monitoring mechanisms placed outside of the user interface code cannot see interactions that do not reach the application code.

These shortcomings can be overcome by an *intrinsic* monitoring mechanism. Something that is intrinsic is inherent or built-in. In this case, the approach is to make the monitoring mechanism an inherent part of the user interface architecture. In this work, the term architecture refers broadly to the software framework on top of which the user interface is constructed. Such an architecture includes mechanisms by which interaction objects can be constructed and mechanisms by which interaction objects and application objects can

communicate. The architecture does not include the underlying windowing system or various libraries of specific interaction objects. The user interface architecture is the prime location for placing monitoring mechanisms because it has access to interobject communication. If the monitoring mechanism resides in the architecture, it is transparent to both specific interaction objects as well as the application.

The key to providing intrinsic monitoring is examining the communication channels between the application objects and the interaction objects. The information communicated at this level does not capture the full semantic intent of the user but it does offer a higher level view of the user's action than that which can seen from low level event recording. It allows the user interface evaluator to analyze and reason about the interface in ways that were not previously possible. For example, instead of knowing only that the mouse button was pressed at location (100,300), the evaluator would rather know that the second item on the menu was selected. If a user interface architecture satisfies certain requirements, then this monitoring mechanism can be easily added.

A sample implementation of an intrinsic monitoring mechanism and a tool to analyze the dialogue data show the merits of this approach. The tools are designed to help user interface designers profile and reason about the user interface. In [12], Hartson and Hix pointed out the increasingly important role of the dialogue designer. Knowledge of both human factors and computer science is needed to construct effective user interfaces. Similar expertise is necessary in order to evaluate a user interface. Although the role of the user interface designer should be more clearly defined, in practice, the task is shared by implementors of the interaction objects and implementors of the application objects. Thus, the monitoring tools will be used by builders of interaction abstractions as well as application programmers.

The monitoring mechanism and analysis tool were implemented within the framework of the Xt Intrinsics on top of the X Window System [1] Version 11 Release 4. The Xt Intrinsics is an example of a user interface architecture that can be enhanced to provide built-in monitoring. (The proper noun *Intrinsics*, referring to the Xt Intrinsics, should not to be confused with the word *intrinsic*, used throughout this document to mean "built-in".) Xmh, a mail handler interface, was used as a sample application for evaluating this monitoring approach.

Chapter 2 discusses the problem of building user interfaces and presents a structural

[1] X Window System is a trademark of the Massachusetts Institute of Technology

model for user interfaces. Chapter 3 lists some questions that an interface designer might ask when evaluating an interface and explains why intrinsic monitoring is necessary in order to answer those questions. Chapter 4 describes the architectural and implementation requirements that must be met in order to provide intrinsic monitoring support. Chapter 5 describes the modified Xt Intrinsics as a sample implementation. Chapter 6 shows how data gathered from intrinsic monitoring can be used to evaluate the Xmh user interface. Chapter 7 summarizes this work and points out some limitations and directions for further research.

Chapter 2

Introduction

2.1 Problem: building good user interfaces

As computing power continues to increase rapidly, the usefulness of computers will be measured not only by their ability to compute but also by their ability to interact with human users. Whereas in the past, users typically interacted with computers through textual interfaces on character-based terminals, today's users have at their disposal economical and powerful microcomputers and workstations that support high resolution graphical displays. More powerful CPU's, cheaper memory, and more sophisticated display hardware have contributed to high expectations for user interfaces. Today's interactive programs are expected to provide powerful yet easy to use, "user-friendly" graphical interfaces. These user interfaces must often support output devices that can display numerous pixels in a variety of colors and auxiliary input pointer devices such as mice and trackballs.

In general, user interfaces are both time-consuming and difficult to create; graphical ones are even more so. In [6], Cardelli gives three broad reasons for this. First of all, the **appearance** of an interface must be satisfactory. The size and type of font, the set of colors, the layout of the graphical objects on the screen, etc. are all important concerns in building an interface. These issues cannot be easily resolved in paper designs. Secondly, the user interface must be **functionally smooth**; the user interface must have a proper "look and feel." The kinds of menus employed, the ways that dialog boxes pop up and down, the availability of keyboard accelerators, etc. make up the feel of the interface. Users are attracted to programs with a familiar user interface feel because they are easier to navigate. Thirdly, there is a heavy **programming burden** involved in creating user interfaces. The

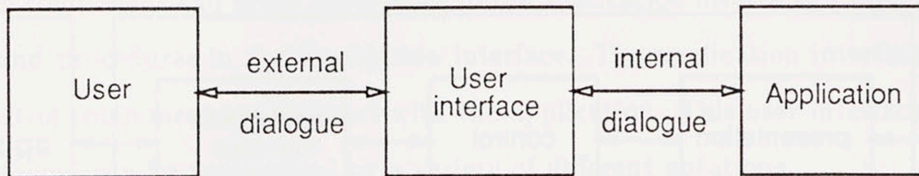

Figure 2-1: A simple model of an interactive program

interface designer must be familiar with the details of the underlying graphical system. Frequently, the user interface must handle asynchronous events from a number of input devices. In addition, user interfaces typically must meet performance requirements in order to maintain an acceptable degree of system responsiveness.

The user interface can account for a substantial fraction of the total code size of the interactive application. In some artificial intelligence applications, 40 to 50 percent of the code and runtime memory is devoted to the interface[4]. The interface code is often complex and cumbersome. In many cases, tight coupling of user interface code and application code results in interactive programs that are hard to debug and modify. Traditional approaches to software design are inadequate in addressing the difficulties specific to the development of interactive programs. Only recently have some methods, techniques and tools been developed to support construction of complex user interfaces.

2.2 Structural model of user interfaces

A structural model is an abstract description of the logical structure of a user interface. The user interface is one piece of the overall interactive program. A simple model of an interactive program is show in Figure 2-1.

This model is fundamental in that it partitions the interactive program into two components: application and user interface. The user interface is the software and hardware that supports human-computer dialogue. It includes graphical displays, keyboards, pointer devices as well as windowing systems and window managers. In this work, however, the

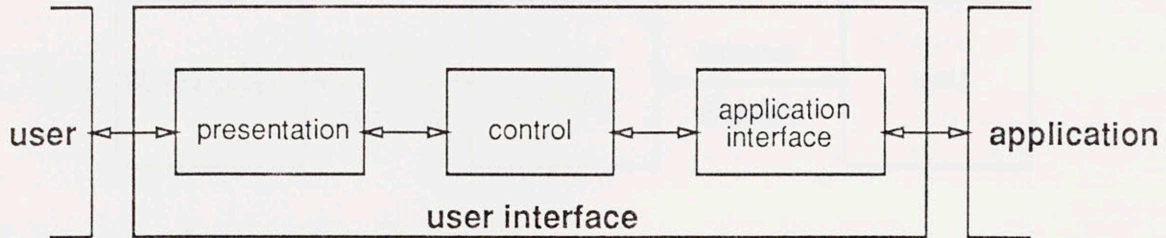

Figure 2-2: Logical model of a user interface

focus is on the software aspects of user interface. Therefore, the external dialogue can be considered communication between the input/output hardware and the user interface software. The application component manages data and performs computations. The internal dialogue is the communication between the user interface and the application.

This model can be refined by examining each of its components. The user can be described abstractly by a task model. A user has a number of tasks he wishes to accomplish. These tasks involve entities that the user is familiar with and may be divided into subtasks. An example of a task is checking one's mail. This task can be broken down into the subtasks of looking in one's mailbox and opening each piece of mail that is there. The task model corresponds to the conceptual layer in the linguistic model[8, 23, 31]. The application component can be described by a model of objects and operations. These objects and operations correspond to data and procedures that the user interface can access. The application objects and operations are usually designed by the programmer to be similar to tasks that the user wishes to accomplish. For example, the objects in a mail handler program are mailboxes, messages, addresses, etc. A mailbox object could have operations such as open and close. A message object supports operations such as read, send, and forward. In this way, the subtask of reading a piece of mail corresponds to the read operation of a message object.

The logical model of a user interface that is similar to the Seeheim model[26] is shown in Figure 2-2. The presentation component is the "look" of the interface. It is responsible for reading the input devices and generating the proper output responses on the screen. The

presentation component processes input from the user into meaningful units called *tokens*. Dialog boxes, command buttons, and menus are some examples of presentation objects. The control component determines the "feel" of the user interface. It receives tokens from the presentation component and sends *commands* to the application interface. Commands map to objects and procedures in the application interface. The application interface specifies how the control component can interact with the application. This user interface model is quite general and can be represented by a variety of different notations.

Although the division into three logical components seems justified, researchers disagree as to how closely those components should be connected. Interaction objects can remain strictly in the presentation component or they can handle presentation as well as control. The control component can bypass the application and send responses directly to the presentation to efficiently provide certain kinds of feedback. On the other hand, the control component and the application interface sometimes need to be tightly linked. The advantages of maintaining a strict logical separation are modularity and modifiability. Some forms of feedback, however, may require application information to be shared across all components.

2.3 Iterative design

Researchers now believe that quality user interfaces can only be constructed by using an iterative design strategy[5]. An effective user interface cannot be produced after a single design pass; there are often subtle design flaws that cannot be detected without the use of prototypes. Iterative design involves constructing prototypes, evaluating those prototypes, and then modifying the design based on the evaluation. The process is repeated until an acceptable user interface is reached.

Tools are now becoming available that enable the interface designer to rapidly construct prototypes; however, few automated tools exist that help the designer evaluate those user interface prototypes. Evaluation of a user interface involves recording dialogue information and using that information to make design changes. An evaluation tool must efficiently record useful dialogue data and allow the designer to analyze the user interface based on that data. Current tools predominantly monitor dialogue at too low a level. As a result, the burden of adding instrumentation to monitor human-computer dialogue is usually placed

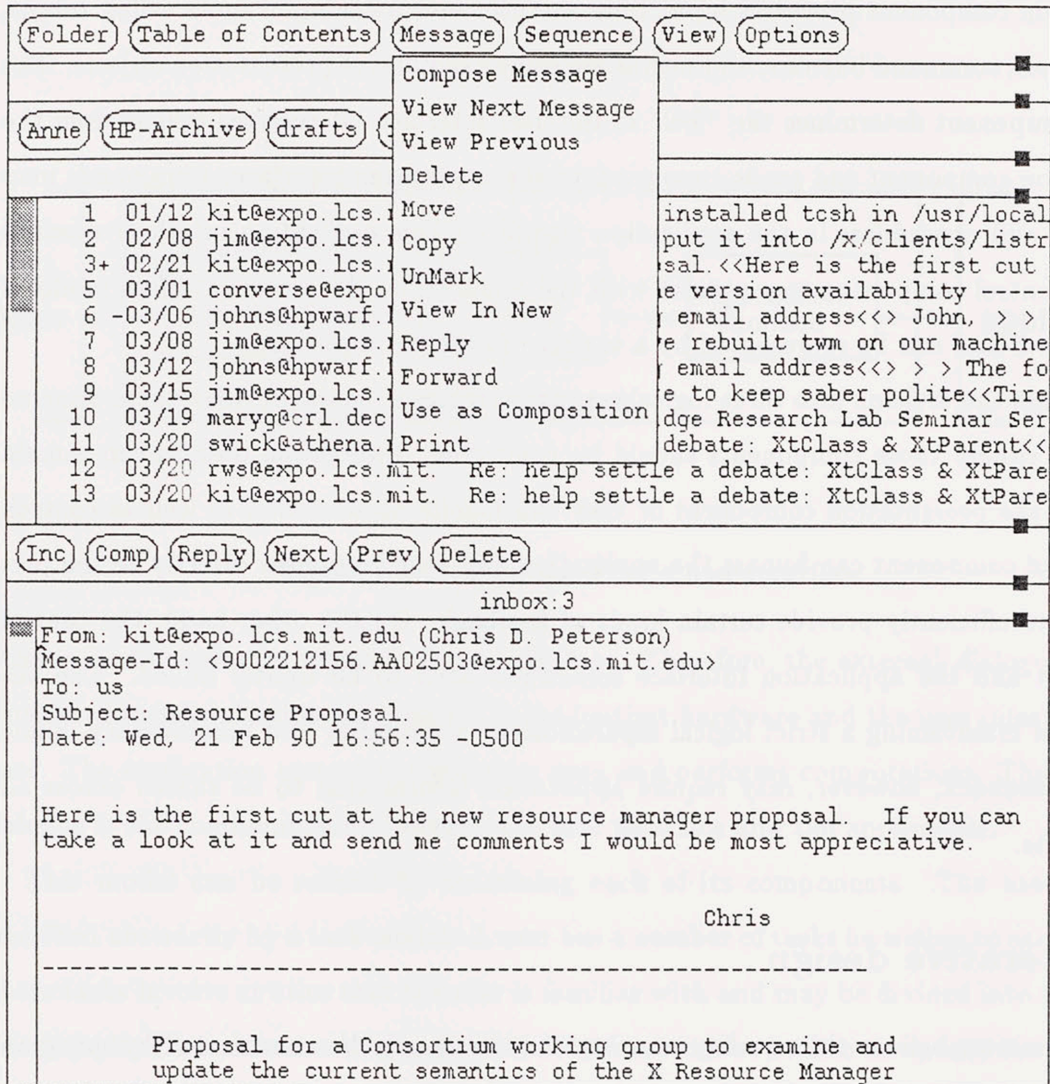

Figure 2-3: xmh user interface

on the application programmer. Furthermore, current recording methods do not capture all the right kinds of information. The evaluator usually does not have enough quantitative data to measure the effectiveness of a user interface.

2.4 An evaluation example

Consider the mail handler interface in Figure 2-3. This particular user interface provides a number of different ways to accomplish the same task. For instance, to view the next mail message, the user may 1) click on the command button labeled Next with any mouse button, 2) click on the appropriate message in the list of message headers with the middle

button, 3) chose the View Next Message option from the Message menu, or 4) use Meta-space as a keyboard accelerator. A question that an interface designer may ask is: which of the four ways to view the next mail message does the user use most? He may wish to get a breakdown of percentage of use of each of the four ways. This is the an example of a question that cannot be answered from quantitative data gathered with current recording tools. With a low level capture tool, he could record every single pointer movement and buttonpress. Information at that level of detail, however, is not sufficient to determine when the user intended to perform a view next message operation. An alternative would be to add a hook to the application code at the place where the view operation is performed. Adding explicit instrumentation becomes extremely tedious when many different aspects of the interface need to be recorded. Even instrumenting the application fails to capture some information. For example, if the interface designer wanted to find out how many Folder menu items were highlighted and not selected as the user scanned for the right folder, he could not merely record from the application end because the application is only notified when a menu item has been selected, not when it is highlighted.

Another question that an interface designer may ask is: what is the next mail action that the user does after he views the piece of mail? Again, this is a question that cannot be easily answered with current tools. The difficult comes not only from discerning when the user has viewed a message but also in finding the next mail action. It is not interesting to know that the user has moved a pointer to a certain location. It is much more useful to know that 50% of the time the user files the message, 35% of the time the user discards the message, 9% of the time the message the user replies to the message, and 1% of the time the user forwards the message. If that were the breakdown, the user interface could be altered so that there is an easy way for the user to save a message. Similarly, the forward command could be moved from a top level button to an item in a pulldown menu because it is rarely used.

These questions and other like them are examples of ways that the user interface designer would to like reason about the interface. Such questions about the user interface will be discussed more generally in Section 3.2.

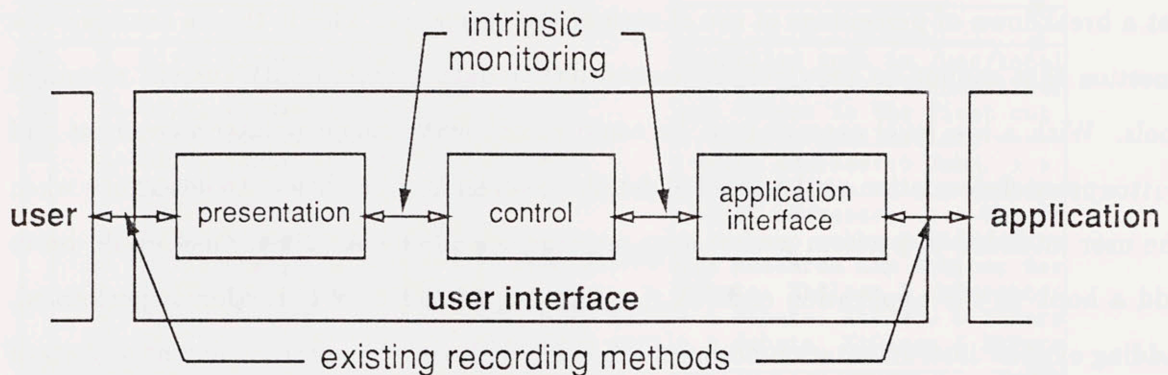

Figure 2-4: Targets of user interface recording

2.5 The focus of this thesis

This work focuses on recording user interfaces: specifically, how to efficiently monitor user interfaces by building recording capabilities into the user interface architecture itself. Existing tools only record communications outside the user interface block as seen in Figure 2-4. The aim is to show that intrinsic monitoring support can be added to obtain dialogue information that is not otherwise accessible with current tools. This information corresponds to communication between the presentation, control, and application interface components. This type of dialogue information is valuable in answering many of the questions that an interface designer asks. A modified Xt Intrinsics and an evaluation of the Xmh mail interface are presented as a sample implementation and use of this approach.

2.6 Related work

Although there has been a significant amount of work done in the area of user interface specifications and user interface development tools, relatively little has been done in the way of providing automated support for user interface monitoring. Current methods of monitoring human-computer dialogue center around either very low level, fine grain recording or high level application-oriented recording.

2.6.1 Specification of user interfaces

Researchers have proposed a number of different notations for specifying user interfaces. Special-purpose languages of various forms have been designed for use with user interface development systems. The notations include state-transition diagrams[15, 35], context-free grammars[24], event languages[10, 13], and object-oriented languages[2, 32]. Some user interface development systems allow graphical specification via direct layout and direct manipulation. Examples include Cardelli's dialog editor[6], Peridot[20], TAE+[33], and Garnet[21].

In the transition network model, the user interface is represented as a set of states. Transitions between states are associated with certain input tokens. The state transitions can also invoke application procedures and cause output to be displayed. One problem with this approach is that the application interface is not very clean. Connections between the states and the application procedures are made through global variables associated with transitions. Another problem with transition networks is that they do not represent modeless, highly interactive interfaces well. For instance, if the user can request help at any time in the interface, every state must have a transition to the help state. In order to add intrinsic monitoring to transition network systems, recording mechanisms should be added at the primary communication channel, the transitions themselves.

Most grammar-based systems use an extended Backus-Naur form notation. The grammar specifies the structure of the dialogue; the tokens in the grammar correspond to user actions and program actions are associated with productions in the grammar. Grammar-based systems suffer some of the same shortcomings as transition networks. Intrinsic monitoring can be added to grammar based systems by instrumenting the parser to record when productions are used.

In the event model, input devices are sources of events. Events are sent to event handlers that can call application procedures, generate output events, or change the state of the system by enabling/disabling other event handlers. In [11], Green shows that the transition network and grammar models can be converted to the event model and that the event model has the greatest descriptive power. The event model is quite central to the monitoring approach in this thesis and will be revisited in Chapter 3. Object oriented systems are similar to event based systems except they provide more structure and framework through the use of encapsulation, class hierarchy, and inheritance. Object orientation is also important to

the intrinsic monitoring approach of this work.

2.6.2 User interface monitoring tools

Although much work is being done in the area of user interface development systems, few systems allow monitoring of user interfaces. In a sampler of twelve representative UIMS's selected for breadth and variety of approach, only four supported interface monitoring, one of which supported only keystroke capture[12]. The interface monitoring tools that do exist predominantly perform fine grain low level input capture or operate outside the actual interface.

Low level capture

Keystroke capture has been a popular method of recording human-computer dialogue at a fine grain of detail. In earlier, character-based user interfaces, keystrokes could be recorded and played back for analysis[22]. Later, as other input devices became more popular, input capture mechanisms were extended to handle pointer devices. The X Window System Version 11 Release 3 distribution contained a document describing a proposal for an input synthesis extension[37]. This extension facilitates monitoring, recording, and testing at the level of events like pointer movements and keypresses. The extension has since been implemented and commercial vendors have experimented with enhancements on it[16]. This extension only applies to low level event capture and has been used mainly to test the efficiency of X servers.

Two systems that go beyond low level input capture are RAPID/USE[35] and the ZOG[38].

RAPID/USE

RAPID/USE[35] is a transition network based UIMS that supports evaluation with input logging. A transition network can be represented as a set of nodes and arcs that represent user interface states and results of user actions. For example, each selection in a menu would be associated with an arc leaving the menu node. Two forms of logging are available: a raw keystroke file and a transition log which records transition, input, output, action, and time stamp for each state transition. Transition logs can be "replayed" to serve as test data. Although transition logs are useful, the way the dialogue information is accumulated and

presented can be improved. An example is given in [35] where the letters "err" are included in every node name that represents a user error so that the designer can later search for and tabulate all the occurrences of the string "err" in the log file. Clearly, this data can be presented and manipulated in a better way. The recording mechanism in RAPID/USE is also closely linked with the transition network based approach. It cannot be used with interfaces that were not designed and built with RAPID/USE.

ZOG

ZOG was a general-purpose human-computer interface developed at the Carnegie-Mellon University Computer Science Department. ZOG was based on the concept of hierarchical frames. In [38], Yoder, McCracken, and Akscyn described how the ZOG user interface was instrumented to collect data about system performance and user behavior. The intrinsic monitoring approach in this thesis was influenced by the "instrumentation" approach used in ZOG. While the ZOG instrumentation was suitable for frame-based ZOG interfaces, a more general mechanism is needed for today's graphical user interfaces. ZOG dealt only with menu-driven, modal interfaces. Today's interfaces operate via direct manipulation and the user has freedom to perform a number of tasks at any point. The ZOG instrumentation recorded program activations because menu choices were associated with program executions. In graphical user interfaces, those activations are not always program executions. Sometimes user actions generate feedback directly from the user interface component and not the application.

Other evaluation tools

Some evaluation tools do not interact directly with the user interface. Examples of these include video cameras and questionnaires. Videotaping a user session has the advantage of recording a user's reactions and verbal comments. It provides a qualitative method of evaluation. It is difficult, however, to obtain quantitative data from a video-recorded session. Questionnaires are also helpful in gathering user feedback. Users may encounter difficulties that software designers did not foresee. Questionnaires give the end user a chance to point out some of these problems. Although questionnaires can be quantitative in nature, they do not serve well in providing such data as how often a particular menu was selected, how long the user hesitated, etc. The questionnaires themselves may also be

quite complex. Research shows that questionnaires, like user interfaces, may need to be iteratively designed[27]! Tools like video cameras and questionnaires are useful, but they must be complemented by monitoring tools that work directly with the user interface.

Chapter 3

Dialogue recording

This chapter describes the kind of information that intrinsic monitoring is intended to record. The first section introduces the concept of *incidents.* The second section lists some questions that can be answered by recording incidents. The third section explains that incidents are best recorded with a built-in recording mechanism.

3.1 Incidents

An *incident* is defined as a unit of information communicated between the presentation, control, and application interface components. It includes tokens passed from the presentation to the control components as well as the commands passed from the control to the application interface. Because these three components may not always be strictly separated, incidents also include presentation to presentation and control to control communication. In actual implementations, a single object may handle functionality that belongs in two or even three of the components in the user interface model. In those cases, incidents include both intraobject and interobject communication.

Incidents differ from low level input events. Input primitives such as keypresses and pointer motions are units of information communicated between the user and the presentation. Incidents may consist of a sequence of input events. For example, a complete drag gesture may include a button press, a number of motion events, and a button release. Those events may constitute one incident that results in a call to an application procedure. On the other hand, incidents are not the same as application operations, either. Several incidents may be needed in order to specify the parameters in a call to an application procedure.

3.2 Questions to be answered

The purpose of recording incidents is to answer questions that interface designers have about the user interface. The need to record dialogue at the incident level stems from the fact that these questions cannot be answered adequately from data gathered by other recording means. Below are some representative questions that serve as motivation for incident monitoring.

- How often does a certain incident occur?
- How long does it take for the program to handle a certain incident?
- How often is a certain sequence of incidents invoked?
- How often is a certain class of tokens (double click, drag, etc.) used?
- What are the most frequently occurring sequences of incidents?
- What is the distribution of different tokens used to invoke the same command? The same application action may be invoked by a button on the screen, an item in a menu, or a keyboard accelerator. What is the relatively frequency of usage of these tokens in relation to each other?
- When does the user go through long periods of waiting, confusion, or frustration? Repeated errors or excessive idle time are indications of this.
- How often are alarm or alert situations triggered? What were the user's responses in those situation? These situations can be traced by examining when certain dialog boxes are popped up.

3.3 Need for intrinsic monitoring

The questions above cannot be easily answered by merely examining low level input events. For example, knowing that a mouse button was pressed at a certain coordinate is not enough to determine if that button press occurred within a graphical button or within a menu item. The low level event information is only useful if something is known about the state of the screen at the time the event was recorded. Thus, the evaluators who examine the event

log must be intimately familiar with the the workings of the program and the layout of the screen at any given point. Even then, it is very difficult to examine recorded event information in isolation. Factors like system load can come into play. For example, on a lightly loaded system, pop-up menus may appear almost instantaneously. The evaluator would then assume that a button press at a certain coordinate corresponds to a certain item on a pop-up menu. If the system load should change, the pop-up menu may appear much later than expected. The user could have clicked prematurely on a menu that has not appeared on the screen yet. It is almost impossible for the user interface evaluator to distinguish between these two situations merely by examining the log of recorded events. Similarly, it is difficult to recognize patterns of incidents. Just by examining raw input events, it would be extremely difficult to determine how often a certain menu item has been selected. It is also hard to measure how long it takes for the program to respond to a certain incident.

In the same way, it is difficult to distinguish between incidents just by recording dialogue information from the application component. By the time the application component is notified of a user-initiated command, information may been lost concerning how that command was invoked. Although recording at the application level is useful for determining *which* commands were invoked, it provides an incomplete picture of *how* those commands were invoked. This is especially true in cases where a command can be invoked in a number of different but equivalent ways. Even in cases where the application is notified of what exact events caused the incident, recording incidents from the application is difficult because there are many entry points into the application from the user interface. If recording was to be done at the application end, every interesting application procedure that can be called from the user interface would need to be instrumented. Every one of those procedures would need to have hooks to handle the dialogue data. That would be a tedious effort if more than a few procedures are involved. Instead, the incident monitoring mechanism should be centralized. Since a framework for communication already exists in the user interface architecture, the best way to record incidents is to instrument the user interface architecture.

Chapter 4

Instrumenting the architecture

This chapter describes how a user interface architecture can be instrumented to provide intrinsic support for monitoring. The first two sections outline the necessary architectural and implementation requirements. The last section explains how the monitoring mechanism can be added.

4.1 Architectural requirements

A user interface architecture must meet certain requirements in order to allow easy addition of intrinsic monitoring support. Fortunately, these requirements are already met by many common architectures. They are also characteristics that will be prevalent in future architectures.

4.1.1 Separation of user interface and application

The notion that an interactive program needs to be separated into a user interface component and an application component is not new. Szekely[34] and Ciccarelli[7] both discuss this issue at length. In [12], Hartson and Hix point out the need for *dialogue independence*. Hurley and Sibert[14] present a model to describe the interaction between the user interface and the application. Some researchers, however, question the strict separation imposed by existing UIMS's[17, 18, 28]. *Application frameworks* have been proposed as an alternative to UIMS's and toolkits[30, 36]. Regardless of their positions on the proper degree of separation between the user interface and the application, most designers would agree that some sort of separation is required.

If there were no separation between the user interface and the application, intrinsic monitoring would become the same as explicit application instrumentation. One major benefit of intrinsic monitoring is the ability to record incidents without changing the application source. If the user interface is intertwined with the application code, any code changes would involve changing the application source. Programs that do not have some kind of separation are difficult to monitor because every such program would need to be explicitly refitted with recording hooks.

This first architectural requirement is simply that an architecture must be in place. A user interface architecture provides means by which the application and the user interface can communicate. An architecture is not to be confused with a library of interaction objects. Such libraries allow common reuse of interactors but they do not necessarily define uniform mechanisms for communication between interactors and the application.

4.1.2 Accessible communication channels

In order to record the information communicated between the user interface and the application components, the communication channels of the user interface architecture must be accessible. The architecture must not only support separation and communication between the two components, but that communication must be "interceptable". This means that the communication channels must either be externally visible or sufficiently modularized so that localized code modifications can be made. The communication channels must be sufficiently accessible so that an external mechanism can be attached to monitor the stream of information flowing between the user interface and the application.

In the best case, the monitoring mechanism is able to inspect not only what is being passed but also who is sending and receiving the information. If the sender is known, the monitoring mechanism can query it for more detail information about the incident. If the receiver is known, more can be discerned about the meaning of the incident. Unfortunately, the receiver is not always known because a user interface object does not have a good handle on the name of the application object. Often the user interface object has only a pointer to an application procedure and therefore, little knowledge of the identity of that procedure and the purpose it serves. In those cases, the user interface designer needs to know a priori which application objects are associated with which user interface objects when he analyzes the dialogue data.

With respect to the user interface architecture, this requirement translates into the need for either a central dispatcher for routing communication or a very localized portion of code where communication is handled.

4.1.3 Identification of incidents

A third important requirement that an architecture must meet is the ability to identify and name incidents. This requirement may be less obvious than the first two. If a monitoring mechanism cannot distinguish between different incidents, it is still able to record them. One option is to simply record all incidents that pass through the communication channels. This is not completely satisfactory, however, because the designer should be able to select the incidents he wishes to monitor. This ability to specify incidents is important in filtering out unnecessary incidents because the volume of dialogue data generated is typically quite large. User interface architectures that permit naming and classification of incidents are the most suitable.

Related to the ability to identify incidents is the ability to identify the senders and receivers. Identifying the sending and receiving objects is harder than identifying incidents because not all objects can be named. Application procedures, for example, often do not have names that are visible within the communication channels. Filtering, if possible, on the senders and receivers will decrease further the volume of dialogue data recorded.

4.2 Implementation requirements

In addition to requirements for the user interface architecture, the implementation of the monitoring mechanism must meet some practical implementation requirements.

- Unobtrusiveness

  The monitoring tool must be inconspicuous. The less an application program needs to be modified in order to take advantage of the monitoring mechanism, the more immediately useful the tool will be. If the monitoring support is completely contained within the user interface architecture, the application code should not need to be changed at all.

- Efficiency

The monitoring mechanism must be highly efficient. Any performance overhead incurred must be low. It is crucial that the instrumented interface resemble, in every respect, the original interface. In particular, the responsiveness of the monitored interface must be similar to the original interface. The more that the speed or feel of the original interface is compromised, the less accurate the recorded dialogue data will be.

- Storage management

  Dialogue data measured at the incident granularity level grows in volume very quickly. The monitoring mechanism must have some way of storing and managing this large collection of data. The performance of the recording mechanism should not degrade as the collection of data grows larger and larger.

- Ability to specify incidents

  The monitoring tool should be able not only to access incidents but also to allow the user interface evaluator to specify incidents of interest. The ability to specify incidents will have secondary benefits with respect to the efficiency and storage management capability of the tool.

4.3 Providing intrinsic monitoring

If the architectural requirements outlined above are met, the task of making modifications to provide monitoring support becomes quite straightforward. The changes involve adding an observer at the appropriate location in the communication channels. The automated observer monitors the communications traffic and notes those incidents which have been specified by the user interface designer. If an interesting incident passes, the observer logs the information about the incident for later analysis. If the sender of the incident is identifiable, the observer can query the sender for more detailed information about the incident. An example of useful detail is the internal state of the interactor at the time of the incident. The entire observer is directly grafted onto the communication channel and becomes a part of the user interface architecture. Thus, the monitoring mechanism is intrinsic to the architecture and not tied to specific interaction objects or the application component.

Chapter 5

Implementation

This chapter describes a sample implementation of the intrinsic monitoring mechanism and a post-processing analysis tool. The sample implementation is a modified version of the Xt Intrinsics library. The first section gives a brief overview of the Xt Intrinsics architecture. The second section discusses the modifications made. The third section explains some of the functionality provided by the analysis tool.

5.1 Xt Intrinsics

The Xt Intrinsics[19] is a policy-free substrate on top of the basic X Window System library, Xlib[29]. The Xt layer provides functions and structures for extending the basic programming abstractions provided by Xlib. Xt is a base layer on top of which a wide variety of toolkits and application environments can be built.

The main advantages of Xt are user interface abstractions called *widgets*, the conventions by which these widgets can interoperate, and resource management. Xt supplies mechanisms for both intercomponent and intracomponent interactions as well as a hierarchical partitioning of widgets into classes. Each widget instance belongs to a widget class, and widget classes may inherit features from their superclasses. Widgets have properties called *resources* that can be easily customized with the X resource database.

The Intrinsics is intended to be "mechanism not policy" in that it does not mandate a certain user interface style. Individual widget implementations and widget sets define the style and consistency of the user interface. The Athena Widget Set[25], Motif[1][9], and

[1] Motif is a registered trademark of Open Software Foundation

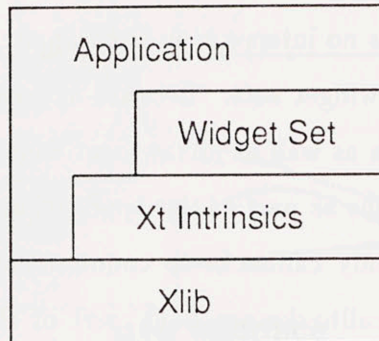

Figure 5-1: The software structure of Xt Intrinsics-based applications

OPEN LOOK[2][1], are examples of widget sets. Xt merely provides the architectural model for constructing and composing widgets that can easily interact with each other and with application components. A typical application would be built atop a widget set, a subset of Xt Intrinsic functions, and a smaller set of Xlib functions. See Figure 5-1.

Xt provides a set of consistent mechanisms through which the application can interact with the widget set. The communication channels in the Xt model are *callbacks* and *actions*. A callback is a procedure associated with a widget that is invoked under certain pre-specified conditions. For example, the PushButton widget in Motif has an activate callback that is invoked when the user presses and releases the active mouse button. A callback is *registered* with the widget by the application. Thus, callbacks are always used for communication between user interface objects and application objects. An action is a procedure that is invoked as a result of a certain sequence of events. Often a sequence of events triggers a sequence of actions. An example of an action is the highlight action of the command widget in the Athena Widget Set. When the pointer enters the window the command widget, the foreground and background colors of the widget are reversed and the widget is highlighted. Actions differ from callbacks in that the conditions under which callbacks are invoked are fixed whereas the conditions under which actions are invoked are dependent on

[2]OPEN LOOK is a registered trademark of AT&T

translation tables. Translation tables are mappings of events into actions. Actions are also different because each widget has some default actions that it can service. Callbacks must be registered by the application but actions can be defined by both the widget set and the application. The highlight action is an example of an action that is built into the command widget. Highlighting a widget requires no intervention from application code.

The use of actions differs across widget sets. Because actions are flexible enough to be used as interwidget communication as well as intrawidget communication, some widget sets allow the application to use actions as part of the application interface. Other widget sets maintain consistency by using only callbacks to communicate between widgets and application objects. If actions and callbacks are both part of the application interface, they must be recorded in order to provide a complete picture of the conversation between the user interface and the application. On the other hand, if actions are not part of the application interface, the user interface designer may or may not want to record actions depending on his evaluation needs. As mentioned earlier, the user interface designer can be considered both a widget designer and an application programmer. As a widget designer, the evaluator would want to view all the intrawidget communication in the form of actions. As an application programmer, however, he may not be interested in actions that never reach the application.

Xt-based applications follow the structural model given in Section 2.2. Interactive programs are separated into application and interface but widgets can encompass both presentation and control aspects. Callbacks are the means by which the control and the application interface communicate. The application interface is determined by what callbacks are registered with the widgets. Actions, because they can be application defined, can serve the same function as callbacks. In addition, they constitute the dialogue between the presentation and control. This dialogue becomes intra-widget communication when the widget includes both the presentation and control. Figure 5-2 shows how Xt fits into the logical model of a user interface. The widget boxes are representative of the many widget instances that make the user interface. Arcs represent actions. The straight lines from the widgets to the application interface represent callbacks.

This model assumes that applications never deal directly with X events. In this model, X events correspond to units of information communicated between the user and the presentation, and are thus not visible to the application. Xlib does not make any such restrictions

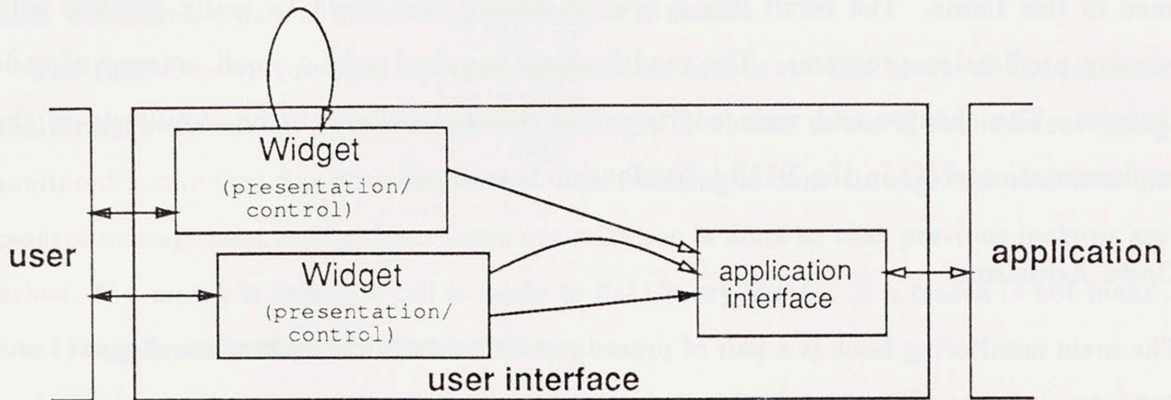

Figure 5-2: The logical structure of Xt-based user interfaces

and in the absence of a user interface architecture like Xt, applications respond to events directly. In such applications, events need to be considered incidents and the processing of events needs to be monitored.

The Xt Intrinsics is an ideal architecture to target the sample implementation because it meets the requirements specified in Section 4.1. It allows partitioning of application and user interface because it provides a consistent interface for communication between the two. Callbacks and actions are incidents and widgets, and application objects are senders and receivers. The handling of callbacks and actions is contained within localized portions of the Xt source code. Thus, it is easy to access and modify the communication channels. Widgets in Xt are designed specifically as extensible abstractions. The X11R4 version of the Xt library has special provisions for extending widget class structures. As a result, query methods can be added to widget objects through extensions. These query objects allow the observing mechanism to ask the sender of an incident for more detail. Finally, incidents can be easily specified with resources because widget classes and widget instances can be named in Xt.

5.2 A modified Xt Intrinsics

The Xt Intrinsics library was modified according to the intrinsic monitoring approach outlined in this thesis. The result was a new Xt library that could be easily relinked with existing application programs. The modifications required only a small amount of code changes. The changes and code additions are detailed below. Some familiarity of the implementation of Xt in the X11R4 distribution is assumed.

Code Additions

The main monitoring hook is a pair of procedures called `XtIncidentMonitor_Begin()` and `XtIncidentMonitor_End()`:

```
/* procedure to call at the beginning of the incident */
Boolean XtIncidentMonitor_Begin(incidentType, widget, name, contextInfo)
    XtEnum incidentType; /* CALLBACK_INCIDENT or ACTION_INCIDENT */
    Widget widget;
    String name;  /* descriptive incident name */
    IncidentContext contextInfo; /* context information about the incident */
            /* the interpretation of this is dependent on the incidentType */

where IncidentContext is defined by:

typedef struct _IncidentContextRec
    XtPointer value;
    XrmQuark record_type;
    struct _IncidentContextRec* next;
 IncidentContextRec, *IncidentContext;


/* procedure to call at the end of the incident */
void XtIncidentMonitor_End()
```

These procedures are used to enclose the area of code that determine the behavior of the incident; that is, they are to be called before and after the invocation of procedures in a callback or action list. A pair of monitoring procedures is needed because widget information as well as starting and ending times must be obtained. If a single monitoring procedure was used, then it must be called after the end of the incident so that the ending

time can be calculated. In some situations, the widget information may be invalid at the end of the incident. An example of this is the destroyCallback incident. Upon the completion of the destroyCallback incident, the widget has been freed and the widget pointer is no longer valid. Therefore, a pair of monitoring procedures is needed.

The functionality provided by these procedures constitute the heart of the intrinsic monitoring mechanism. `XtIncidentMonitor_Begin()` checks to see if the incident being monitored is one that has been requested. This checking is done through the standard Xt resource management mechanism. Some optimization is done so that previous lookups are cached. If a match is found, a call is made to `CallQueryProc()`. If a match is not found, `XtIncidentMonitor_Begin()` returns false.

QueryProcs are query methods that are attached to widget classes. If a particular widget class does not export a query method, then `CallQueryProc()` will look up the class hierarchy for a superclass that does. The `InstallQueryProc()` procedure is provided for convenient attachment of query methods to existing widget classes. The widget set designer is the one who would be writing and installing these query methods. `InstallQueryProc()` makes use of the Xt class extension mechanisms. Below are the headers for `CallQueryProc()` and `InstallQueryProc()`.

```
Boolean CallQueryProc(w, call_data)
    Widget w;
    XtPointer call_data;

void InstallQueryProc(wClass, queryProc, closure)
    WidgetClass wClass;
    XtQueryProc queryProc;
    XtPointer closure;
```

where XtQueryProc is:

```
typedef void (*XtQueryProc)(
/*  Widget w;
    XtPointer client_data;
    XtPointer call_data
*/
);
```

The query method is responsible for storing information in the area pointed to by call_data. This information will be stored along with other incident details given in the contextInfo parameter to XtIncidentMonitor_Begin(). The contextInfo parameter points to details about what events caused the incident. More precisely, the contextInfo parameter contains some pointers to the internal translation state from which the exact events can be extracted. XtIncidentMonitor_Begin() also computes the starting time of the incident as part of the incident information. All the incident information is actually stored on a stack awaiting output by XtIncidentMonitor_End(). A stack is used to insure that nested incidents, i.e., an incident caused by the handling of another incident, will be recorded in order.

After the program responds to the incident, i.e., all the procedures in the callback or action list have been invoked, XtIncidentMonitor_End() is called if XtIncidentMonitor_Begin() returned true. If XtIncidentMonitor_Begin() returned false, that means the incident in question was not requested and no further work needs to be done to store the incident details. If XtIncidentMonitor_Begin() returned true, XtIncidentMonitor_End() is called. XtIncidentMonitor_End() computes the ending time of the incident. The start and stop times are optimistic estimates of how long it takes for the application to respond to the incident. The difference between the stop and start time is at least as long as the user must wait before he can do something again. The user may need to wait even longer if there are further delays through the user interface code. Presently, XtIncidentMonitor_End() outputs ASCII text to an auxiliary process. The output can be further compressed and reformatted by the auxiliary process. Currently, an awk[3] script performs some simple compression, the most useful of which is to create a dictionary of commonly occurring string phrases. That dictionary maps a long string into a small integer, thereby reducing the size of the data. The incident details that are currently stored are incident name, incident type, widget instance, widget class, start time, duration, time since last incident, incident description, and widget description. The incident description is the sequence of X events that caused the incident. The widget description is the information obtained by querying the widget that sent the incident. It is a string that can contain any arbitrary information about the widget state. The incident detail is shown in the bottom pane of the main window of the analysis tool (Figure 5-3).

Code Modifications

A small number of changes was made to the following Xt source files.

- Initialize.c

  An addition was made to `XtAppInitialize()` to initialize the monitoring mechanism along with the regular Xt initialization routine.

- Callback.c

  Since all callbacks are eventually made in `_XtCallCallbacks()`, that is where the monitoring mechanism sits. First, the name of the callback list, if there is one, is determined. Then, a call is made to `XtIncidentMonitor_Begin()`. Following that, each callback in the list is invoked. Finally, `XtIncidentMonitor_End()` is called.

- TMstate.c

  The changes are essentially the same as those in Callback.c except modifications are made in two places in `_XtTranslateEvent()`. Regular actions and accelerator actions are handled slightly differently but with respect to incident monitoring, they are essentially the same.

Incident specification

If a few simple conventions are followed, incidents can be specified just like regular Xt resources. Resources are specified with a name/value pair. The name is the concatenation of the application name, the widget name, and the resource name with a period between each. The widget name is an instance, class, or instance/class hierarchy. The hierarchy includes the widget and its ancestors. The asterisk symbol can be used as a wildcard.

Incidents should be specified by their callback or action instance name or class name. The words Callback and Action denote the respective classes. The incident is specified as if it were part of a larger application called XtMonitor. The actual application name follows XtMonitor. The value of the incident resource can be either `On` or `Off`. Below are some sample specifications and what they mean:

`XtMonitor.Xmh*Action:  On`

- monitor all actions in Xmh-class programs

```
XtMonitor.Xmh*comp*Callback:  On
```

- monitor all callbacks in the comp widget and all its children in Xmh-class programs

```
XtMonitor.xmh.xmh.tocMenu.inc.callback:  On
```

- monitor the callback in the inc button on the tocMenu menu

```
XtMonitor.Xmh*XmhIncorporateNewMail:  Off
```

- do not monitor the action XmhIncorporateNewMail in Xmh

```
XtMonitor.xmh-jc*toc.ScrollBar*Action:  On
```

- monitor all actions of Scrollbar widgets inside the widget toc in the xmh-jc program.

5.3 Analysis tool

An analysis tool was implemented to process the data gathered from the instrumented Xt Intrinsics. The tool was essentially a database designed specifically for storing and manipulating incident details. The tool itself had a graphical user interface (Figure 5-3). The tool's main features are filtering and sorting capabilities. The analysis tool is able to filter the data by any combination of data fields. Filtering of sequences of incidents is also supported. For example, the designer can request all incidents that occurred in widgets of the Text class that had duration greater than 0.5 seconds. Similarly, an interface designer can request all three-incident sequences that begin with an incident named **help** and end with an incident named **cancel**. Filtering is cumulative so successive filtering of the data is possible. The tool also allows the interface designer to sort in any order by any field or any combination of fields. In addition, the tool can provide summation information as well as context information. Summation information indicates the total occurrences of each of the different kinds of incidents. Context information indicates where in the overall sequence of incidents a certain incident occurred.

These features are designed to assist the user interface designer analyze the recorded dialogue data. The designer, by using various combinations of sorts and filters, can answer the questions posed in Section 3.2 and thus, reason about the interface being evaluated. The graphical interface helps makes it easy for the designer to define sorts and filters. Many other features can be added to provide even more sophisticated analysis of the recorded dialogue data.

```
Show All   Summary   Reorder   Quit

Incident Name : ANY
Incident Type : ANY
Number of Occurrences : ANY
Widget Instance : ANY
Widget Class : ANY
Start Time : ANY
Duration : ANY
Time since last incident : ANY
Incident Description : ANY
Widget Description : ANY
Filter Action : SHOW HIDE

Apply   Prev Match_Rec   Next Match_Rec   Add Match_Rec   Clear Filter

 1   0.000001 highlight                        button2       <EnterNotif
 1   0.000001 reset                            button2       <LeaveNotif
 1   0.000001 highlight                        button2       <EnterNotif
 1   0.000001 set                              button2       <ButtonPres
 1   1.360000 XmhComposeMessage                button2       <ButtonPres
 2   0.000002 reset                            button2       <ButtonRele
 2   0.180001 callback                         textSource    <KeyPress>
 1   0.020002 insert-char                      comp          <KeyPress>
 2   0.019997 delete-previous-character        comp          <KeyPress>D
 3   0.000003 insert-char                      comp          <KeyPress>
 2   0.000002 delete-previous-character        comp          <KeyPress>D
21   0.200013 insert-char                      comp          <KeyPress>i
 1   0.060000 next-line                        comp          Ctrl<KeyPre

MATCHES: 862 (100.000%)  TOTAL DURATION : 119.542290 (100.000%)

Incident Name              : delete-previous-character
Incident Type              : Action
Number of Occurrences      : 2
Widget Instance            : xmh.xmh.comp
Widget Class               : ApplicationShell.TopLevelShell.Paned.Text
Start Time                 : 638742903.420000 (Thu Mar 29 15:35:03 1990
)
Duration                   : 0.000002
Time since last incident   : 0.639997
Incident Description       : <KeyPress>Delete: delete-previous-character()
```

Figure 5-3: Analysis tool interface

Chapter 6

Example of use – Xmh

This chapter discusses how an application is actually evaluated with the instrumented Xt Intrinsics presented in Section 5.2. The Xmh mail handler was used as the target application. Xmh is the X interface to the Rand MH Message Handling System. Xmh, in most respects, is only a graphical interface to MH. All the actual mail handling commands are handled by calls to MH. The Xmh interface employs a rich variety of widgets ranging from command buttons to pulldown menus to text widgets. It is built using widgets from the Athena widget set built atop Xt. Thus, it serves well as a target for testing the modified Xt Intrinsics.

6.1 Instrumenting Xmh

Instrumenting Xmh was extremely easy because Xmh itself did not need to be changed. To produce an instrumented version of Xmh, only a relink with the new Xt library was necessary.

6.2 Reasoning about the interface

This section explains how the analysis tool can help answer some of the questions posed in Section 3.2. For each question, the kind of filtering necessary will be described, but the exact method of specifying that filter with the analysis tool will not be given in great detail because the analysis tool is only a sample implementation. The important point is that the recorded data can be manipulated to provide answers. Below are some sample questions.

- How often does the incident XmhViewNextMessage occur?

A simple filter on the incident name will find all the occurrences of the XmhViewNext-Message incident. By examining and totalling all such incidents found, the designer can see how long it took for the program to handle all such incidents.

- What tokens does the user use to invoke the command to view the next message?

  This is a slightly different question from the previous one because the interface may allow the user to compose messages in a number of different ways. The incident XmhViewNextMessage may be associated with a certain command button or a certain keyboard accelerator but an item in the Message menu also allows the user to view the message. The menu does not use the same XmhViewNextMessage action to communicate with the application. In such a case, the user interface designer must filter not only by the incident name XmhViewNextMessage but also the `next` widget in the `messageMenu` widget. The designer must know in advance the different ways to invoke the same application command. A better user interface design would associate the XmhViewNextMessage action with all possible ways to view the next message, including selecting an item on the pulldown menu.

- What does the user do immediately after viewing a message?

  To answer this, the interface designer can create a filter to search for five-incident sequences that have as their first incident either the XmhViewNextMessage or the `messageMenu.next` widget. This allows him to see the four incidents that occurred right after user viewed a message. A subsequent filter can be used to show when the user performed specific actions. In this way, the designer can tell, for example, that 5% of the time the user replied to the message viewed.

- How often does the user try to incorporate new mail when there isn't any?

  When the user tries to do this, a dialog box pops up to inform him of the error. A search for an incident sequence that begins with the incident named XmhIncorporate-NewMail followed by any other incident that occurs within a dialog widget will show these occurrences.

- How often does the user perform drag gestures with the button 1?

  This involves searching for incident sequences so that the first incident has the <ButtonPress>1 event, arbitrary middle incidents have Button1<MotionNotify>

events, and the last incident has a <ButtonRelease>1 event. The events are part of the incident detail description. These gestures can be recognized only if there are incidents in place to respond to all parts of the gesture. In this case, if Button1<MotionNotify> events never triggered any incidents, then this gesture cannot be detected.

- When does the user go through long periods of waiting?

  The analysis tool calculates the time since the last incident. A sort can be done by that field to show the incidents with the longest time since last incident. This time is not always indicative of wait time because there is no way to know if the user is actually sitting idle or has left the terminal. Still, this information can be valuable if used in conjunction with other monitoring mechanisms such as videotaping. Although videotaping can show when the user is waiting, incident monitoring can provide quantitative data on what the user does immediately after a long period of waiting.

Chapter 7

Conclusions

7.1 Contributions

The main contribution of this thesis is a novel approach to acquiring dialogue information. This work shows that user interfaces can be effectively monitored by adding recording mechanisms to the user interface architecture. A mechanism internal to the architecture has access to communication information that would otherwise not be captured by low level input or high level application-specific recording schemes. Intrinsic monitoring yields dialogue information at the proper level of detail for answering the questions that arise in evaluating an user interface. By making the recording mechanism a part of the user interface architecture itself, benefits of dialogue monitoring can be realized without making changes to the application source.

7.2 Limitations and future work

The main limitation of the intrinsic monitoring strategy presented here is its inability to take into account incidents that originate from the application component. Output incidents are completely ignored. Output incidents would be actions initiated by the application component. Examples include popping up a widget on the screen, making an active menu item inactive, and registering another callback to a pushbutton widget. Such incidents need to be recorded in order to provide a clear picture of both directions of the human-computer conversation. Just monitoring the input incidents forces the user interface designer to have intimate knowledge of the application state in order to understand the dialogue data.

The current approach also does not support playback well. The prospect of playing back incidents seems possible but exceedingly difficult. The entire user session cannot be replayed without additional lower level event information as well as output incidents. Lower level event information is needed to reproduce the user actions that did not trigger incidents, and output incidents are necessary for synchronizing playback. Even so, it would be useful to replay the incidents at some level even if the session cannot be completely reproduced.

A promising direction for future research is integrating different kinds of recording tools together. Intrinsic monitoring should be combined with low level and high level monitoring mechanisms to provide a complete user interface evaluation tool. Monitoring tools should be used not only to reason about interfaces in the way described in this thesis but also to facilitate testing, profiling, and debugging of user interfaces. User interface evaluation tools should reach a stage where the designer can use them as programmers now use debuggers. The quantitative data acquired by these tools should be used by experts in cognitive psychology and human factors to study and understand how people use computers. Good evaluation tools will enhance the productivity of user interface designers and enable them to create better user interfaces in the future.

Bibliography

[1] An OPEN LOOK Toolkit for the X Window System.

[2] Creating User Interfaces with Open Dialogue. Apollo Computer, Inc, January 1989.

[3] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. *The Awk Programming Language*. Addison-Wesley, 1988.

[4] D.G. Bobrow, S. Mittal, and M.J. Stefik. Expert Systems: Perils and Promise. *Communications of the ACM*, pages 880–894, September 1986.

[5] W.A. Buxton and R. Sniderman. Iteration in the Design of Human-Computer Interface. In *Proceedings of the 13th Annual Meeting of the Human Factors Association of Canada*, pages 72–84.

[6] L. Cardelli. Building User Interfaces by Direct Manipulation. Technical Report 27, DEC Systems Research Center, 1987.

[7] IV E.C. Ciccarelli. Presentation Based User Interfaces. Master's thesis, Massachusetts Institute of Technology, August 1984.

[8] J. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, 1982.

[9] Open Software Foundation. *Application Environment Specification (AES) User Environment Volume*. Prentice Hall, 1990.

[10] M. Green. The University of Alberta User Interface Management System. In *Proceedings of SIGGRAPH*, pages 205–213. ACM SIGGRAPH, July 1985.

[11] M. Green. A Survey of Three Dialogue Models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.

[12] H.R. Hartson and D. Hix. Human-Computer Interface Development: Concepts and Systems. *ACM Computing Surveys*, 21(1):5–93, March 1989.

[13] R.D. Hill. Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction — The Sassafras UIMS. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

[14] W.D. Hurley and J.L. Sibert. Modeling User Interface-Application Interactions. *IEEE Software*, pages 71–77, January 1989.

[15] R.J.K. Jacob. A Specification Language for Direct-Manipulation User Interfaces. *ACM Transactions on Graphics*, 5(4):283–317, October 1986.

[16] A.G. Jamison. Experiences Developing X Server Support for the Programmed Control of a X Window Workstation. 4th Annual X Technical Conference, Boston MA, 1990.

[17] M.A. Linton, J.M. Vlissides, and P.R. Calder. Composing User Interfaces with Interviews. *IEEE Computer*, pages 8–22, February 1989.

[18] J.E. Lovgren. Are We Boxing Ourselves in with the UIMS Box? In *Proceedings of the 31st Annual Meeting of the Human Factors Society*, pages 22–24, 1987.

[19] J. McCormack, P. Asente, and R. Swick. X Toolkit Intrinsics - C Language Interface. X11R4 distribution.

[20] B.A. Myers. *Creating User Interfaces by Demonstration.* Academic Press, 1988.

[21] B.A. Myers. An Object-Oriented, Constraint-Based, User Interface Development Environment for X in CommonLisp. 4th Annual X Technical Conference, Boston MA, 1990.

[22] A.S. Neal and R.M. Simons. Playback: A Method for Evaluating the Usability of Software and Its Documentation. In *Proceedings of SIGCHI*, pages 78–82, December 1983.

[23] W. Newman and R. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979.

[24] D.R. Olsen, Jr. and E.P. Dempsey. SYNGRAPH: A graphical user interface generator. *Computer Graphics*, pages 43–50, 1983.

[25] C.D. Peterson. Athena Widget Set - C Language X Interface. X11R4 distribution.

[26] G.D. Pfaff, editor. *User Interface Management Systems*. Springer-Verlag, 1983. Proceedings of the Workshop on User Interface Management Systems held in Seeheim, FRG, Novemeber 1-3, 1983.

[27] R.W. Root and S. Draper. Questionnaires as a Software Evaluation Tool. In *Proceedings of SIGCHI*, pages 83–87, December 1983.

[28] M.B. Rosson, S. Maass, and W.A. Kellogg. Designing for Designers: An Analysis of Design Practices in the Real World. In *Proceedings of SIGCHI+GI*, pages 137–142, 1987.

[29] R.W. Scheifler, J. Gettys, and R. Newman. *X Window System - C Library and Protocol Reference*. Digital Press, 1988.

[30] K.J. Schmucker. MacApp: An Application Framework. *Byte*, pages 189–193, August 1986.

[31] B. Schneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 1987.

[32] J.L. Sibert, W.D. Hurley, and T.W. Bleser. An Object-Oriented User Interface Management System. In *Proceedings of SIGGRAPH*, pages 259–268. ACM SIGGRAPH, August 1986.

[33] M.R. Szczur and P. Miller. Transportable Applications Environment (TAE) PLUS, Experiences in "Object"ively Modernizing a User Interface Environment. In *Proceedings of OOPSLA*, pages 58–70, 1988.

[34] Pedro Szekely. *Separating the User Interface from the Functionality of Application Programs*. PhD thesis, Carnegie-Mellon University, January 1988.

[35] A.I Wasserman and D.T. Shewmake. *The Role of Prototypes in User Software Engineering (USE) Methodology*, volume 1, pages 191–210. Ablex, 1985.

[36] A. Weinand, E. Gamma, and R. Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structure Programming*, 10(2):63–87, 1989.

[37] L. Woestman. X11 Input Synthesis Extension Proposal. X11R3 distribution.

[38] E. Yoder, D. McCracken, and R. Akscyn. Instrumenting a Human-Computer Interface For Development and Evaluation. In *Human-Computer Interactions – INTERACT '84*, pages 907–912. Elsevier Science Publishers B.V. (North-Holland), 1984.

2360-71