COMPUTER-BASED USER INTERFACE EVALUATION BY ANALYSIS OF REPEATING USAGE PATTERNS IN TRANSCRIPTS OF USER SESSIONS

by

Antonio C. Siochi

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science and Applications

APPROVED:

R. W. Ehrich, Chair

D. S. Hix

J. D. Arthur

R. C. Williges

E. A. Brown

April, 1989

Blacksburg, Virginia

COMPUTER-BASED USER INTERFACE EVALUATION BY ANALYSIS OF REPEATING USAGE PATTERNS IN TRANSCRIPTS OF USER SESSIONS

by

Antonio C. Siochi

Committee Chair: Dr. Roger W. Ehrich
Computer Science

(ABSTRACT)

It is generally acknowledged that the production of quality user interfaces requires a thorough understanding of the user and that this involves evaluating the system by observing the user using the system, or by performing human factors experiments. Such methods traditionally involve the use of videotape, protocol analysis, critical incident analysis, etc. These methods require time consuming analyses and may be invasive. In addition, the data obtained through such methods represent a relatively small portion of the use of a system. An alternative approach is to record all user input and system output onto a file, i.e., log the user session. Such transcripts can be collected automatically and over a long period of time. Unfortunately this produces voluminous amounts of data. There is therefore a need for tools and techniques that allow an evaluator to extract from such data potential performance and usability problems. It is hypothesized that repetition of user actions is an important indicator of potential user interface problems.

This research reports on the use of the repetition indicator as a means of studying user session transcripts in the evaluation of user interfaces. The dissertation discusses the algorithms involved, the interactive tool constructed, the results of an extensive application of the technique in the evaluation of a large image-processing system, and extensions and refinements to the technique. Evidence suggests that the hypothesis is justified and that such a technique is convincingly useful.

ACKNOWLEDGEMENTS

Ad Majorem Dei Gloriam!

There are debts which can never be repaid — gifts of time and effort, shared so others may grow. I owe such a debt to my advisor, Dr. Roger W. Ehrich. He has patiently and expertly guided me throughout my voyage of discovery. He has helped me develop my sense of awe and appreciation of the beauty and elegance of mathematics. This dissertation has its roots in his efforts at providing metering tools in User Interface Management Systems. The notion of mrps was born out of his constant prodding for something more than transition matrices. For all his help, I am profoundly grateful.

My thanks are also due to committee members Drs. Robert C. Williges, James D. Arthur, and Ezra A. Brown. Dr. Williges provided advice and encouragement both during my Master's work and this effort, Dr. Arthur taught me about translators (a lesson from which I have employed in the architecture of the mrp tools), and Dr. Brown, a mathematician, encouraged me by his interest in my work.

I am in no small measure indebted to Dr. Deborah Hix, whose friendship I value, and whose penchant for detail has improved my work. She has very graciously served on my advisory committee. I owe my association with her and the DMS group to Dr. H. Rex Hartson. Dr. Hartson has contributed much to my understanding of human-computer interaction through numerous discussions and spirited debate. His extensive library was always open to me.

The DMS group has been my academic home. It is an incredibly stimulating, challenging, and fertile environment for exploring ideas. Drs. Hartson and Hix, through this group,

have provided me with valuable research experience and opportunities for professional growth.

I would also like to thank the other members of the DMS group, both past and present, and especially . has helped me with UNIX, vi, and C, and has served me well as a devil's advocate.

This research was partially supported with funds obtained from grants by the IBM Corporation, Software Productivity Consortium, and the Virginia Center for Innovative Technology. I am also pleased to acknowledge the support of the Computer Science Department, especially the efforts of Dr. D. C. S. Allison, , , and . The office staff have helped me deal with the considerable amount of paperwork required.

I thank my daughter, , for loving me despite my numerous absences. Finally, to my wife, , who has supported me in this work, I owe my sanity. She has given me a home, a sturdy anchor amid the tempests and gales of graduate life. She has been my steady companion in this great adventure.*She did this while working on her own Ph.D.* Mahal, maraming salamat.

DEDICATION

Sa aking mga magulang,

Tanggapin sana ninyo itong munting handog.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGEMENTS iii

DEDICATION v

Chapter | page

I INTRODUCTION 1

II USER INTERFACE EVALUATION 3

Background 3

Analytical Evaluation Methods 3

Empirical Evaluation Methods 6

Research Scope 8

III COMPUTER-BASED INTERFACE ANALYSIS 11

Logging User Sessions 11

Examples 12

Repeating Patterns: The Repetition Hypothesis 14

IV MAXIMAL REPEATING PATTERNS (MRPS) 17

Problem Reduction 17

Definitions 18

Algorithmic Complexity 20

Position Trees 22

Position Identifiers 22

Labelled Trees 22

Properties 25

The Detection Algorithm 31

V TESTING MRP USEFULNESS 35

GIPSY 35

Data Collection 36

Data Analysis 36

The Normalizer 38

The Mrp Tool 40

Procedure 43

VI RESULTS 46

Mrp Tool Results 46

Structured Interview Results 52

U17 Comments 53

U14 Comments 54

Common Comments 56

Known GIPSY Problems 57

Problems With the Technique 59

VII CONCLUSIONS 64

Advantages 64

Limitations 66

Recommendations 68

Controlling the Mrp Flood 68

Formal Investigation 69

A Transcript Analyzer 70

Direct Manipulation Interfaces 73

VIII FUTURE WORK 76

Extensions and Interface Improvements 76

Time Stamping 76

Full Programming Language 76

Similarity Indicator for Command Lines 77

Mrps with Variables 77

Support for Videotape Analysis 78

Behavior Characterization 78

Other Applications of Mrps 79

IX SUMMARY 80

REFERENCES 81

APPENDIX 84

VITA 92

LIST OF FIGURES

Figure page

1. A brute force algorithm to find all repeating substrings. 23

2. Position tree for the string S = "abcabcxab." 27

3. Extracting mrps from a position tree 34

4. Mrp analysis tools and the evaluation process 37

5. The normalizer converts raw transcripts to a standard form. 39

6. Highest level structure of the mrp tool 42

7. The evaluation procedure 45

8. An instance of a type 1 mrp: consecutive invocations of the same command 48

9. An instance of a type 2 mrp: empty command lines 50

10. Plot of number of lines vs. number of mrps detected. 61

11. Distribution of number of positions an mrp occurs at according to its length. 62

12. Distribution of frequency of an mrp according to its length. 63

LIST OF TABLES

Table page

1. Mrp tool functions. 41

2. Selected information from mrp analysis of seventeen transcripts. 47

Chapter I

INTRODUCTION

> Understanding the usability of a product from the users' point of view necessitates developing data collection techniques that can access users' day-to-day life experience.
>
> Whiteside, Bennett, & Holtzblatt [1987]

> The null hypothesis can always be rejected, as long as enough time and money are spent.
>
> Snyder [1987]

The past decade has seen the emergence of a new conciousness in the interactive software community. Users are demanding that their software not only provide computational power, but also that the software be *usable*. Managers are growing aware of the long-term costs associated with poor usability, and developers are beginning to design systems that *fit the user* as well as meet functional requirements.

This push for usability has forced the modification of standard software development methodologies. There is growing support for the process of iterative design and rapid prototyping — a design process in which user feedback is an essential element. In support of this, Carroll & Rosson [1985] argue that design is a dynamic process that does not fit into a strict top-down or hierarchical decomposition scheme. They stress the importance of an iterative design process, supported by empirical evaluations of system prototypes:

> The most important aspect of the empirical approach is that it encourages the discovery of design solutions which on purely analytic grounds might have been missed. ... Our view is that design activity is essentially empirical. This is not because we "don't know enough yet," but because in a design domain we can never know enough.

Hartson & Hix [1989] echo this view in their proposal for a dialogue development lifecycle that proceeds in alternating "waves" of bottom-up and top-down design activities. Bottom-

up activities reflect the creative phase of design and consist of sketching out dialogue scenarios, i.e., concrete examples of what the displays might look like and how the dialogue flows from one interaction to the next. Top-down activities provide a means of structuring the various interactions into an organized view of what the system supports and how the user communicates with the system.

Williges [1987] cites Gould & Lewis's statement that iterative design is an important aspect of user interface development [Gould & Lewis, 1985], and outlines an iterative design process from Williges, Williges, & Elkerton [1987] who outlined the process from the various methodologies they examined.

Whiteside, Bennett, & Holtzblatt [1987] state several reasons for using an iterative design methodology. First is that "User testing is the most reliable way to debug the user interface." Second, the feedback provided by frequent testing increases the amount of control managers have over the development project. Last, because some version of the system must be available for testing, developers can respond more quickly to management decisions which might advance the ship date.

Against this background, the need for evaluating the human-computer system, and specifically its user interface, stands sharp and clear. In response, many methods and techniques for evaluating a user interface have already been developed. Each has its strengths, limitations and costs. The contribution of this research is a new, relatively low-cost evaluation technique based on the detection of repeated user actions in transcripts of user sessions. The algorithm and tools developed to detect this repetition can rapidly identify sections of transcript which potentially show problems users were having with the interface. Because of this capability, there is no need to review all the raw data. This results in faster analyses and makes feasible the analysis of data collected over extended periods of time.

Chapter II
USER INTERFACE EVALUATION

User interface evaluation is an incredibly complex task. This is not surprising since there is little agreement as to where to draw the line between the interface and the rest of the system. Indeed, arguments have been made that testing only the interface while ignoring the rest of the system, the environment in which the system is used, or the documentation supplied with the system, provides limited, myopic results. The complete picture is missing.

It is not the purpose of this research to debate the issue of just where the interface is, much less resolve it. Neither will this research challenge the argument of limited, myopic results, for it is certainly true. What is also true, however, is that every measurement, every test, and every experiment performed, have a cost. It is a question of how many measurements, tests, and experiments can be performed given the limited resources at hand. It is a question of how useful the data collected from such tests are.

This chapter will review various interface evaluation methods and examine their strengths and weaknesses. It concludes with a statement of the scope of the research.

2.1. BACKGROUND

2.1.1. Analytical Evaluation Methods

Analytical evaluation methods consist mainly of studying specifications and designs of user interfaces in order to make judgements, based on specific models and theories of interaction, about the quality of the interface were the design to be implemented. In this sense, such methods are *predictive*.

Among the early analytical methods to appear in the literature are the Keystroke-Level model of Card, Moran, & Newell [1980] and the Action Language of Reisner [1981]. The Keystroke-Level model predicts error-free performance times for specific tasks, using specified methods. Each task is accomplished by performing a specific method, and each method is minutely described in terms of the keystrokes, homing operations, and mental preparations required. Each of these acts has a performance time associated with it, and rules are given for computing the performance time of the method based on the performance times of the acts. The model was reported to have a 21% prediction error when it was tested. The model was intended solely to predict performance times and does not consider other usability aspects such as ease of learning, or error rates. Card, Moran, & Newell [1983] present a fuller picture of this work.

Reisner's Action Language is similar in the sense that it is a grammatical representation of the task structure and sequences of actions required to perform tasks. It is different in that it does not predict performance times, but rather compares different interfaces using three metrics: the number of unique terminal symbols in the grammar, the length of sentences, and the size of the grammar. Based on these metrics, Reisner was successfully able to predict ease of learning and remembering how to perform certain tasks in the systems she was comparing.

Payne & Green [1986] describe a similar meta-notation, which they call Task Action Grammar, or TAG. It is a refinement of the fundamental ideas advanced by Reisner and is an attempt to measure the consistency of an interface. The notational device they introduce is the use of variables in the left- and right-hand sides of production rules. Thus instead of writing several rules such as

```
task-moveForward-oneWord -> "F" + "W"
task-moveBackward-oneWord -> "B" + "W"
task-moveForward-oneChar -> "F" + "C"
task-moveBackward-oneChar -> "B" + "C"
```

Payne & Green collapse the "task-move" rules into one rule where the variables are written in square brackets,

```
task-move[Direction, Unit] -> symbol[Direction] + symbol[Unit]
```

together with the following ancillary rules,

```
symbol[Direction = Forward] -> "F"
symbol[Direction = Backward] -> "B"
symbol[Unit = Word] -> "W"
symbol[Unit = Char] -> "C"
```

thereby exposing the similarity of the various "task-move" rules. By writing interface designs in such a manner, and applying metrics such as the number of such collapsed rules, Payne & Green were able to reach the same conclusions Carroll [1982] reached about the learnability of two experimental languages empirically studied by Carroll.

Kieras & Polson [1985] analyze the cognitive complexity of an interface by formally representing, as a set of production rules, the user knowledge required to perform tasks with the interface. The two representations are combined in order to simulate the interactions on a computer, thereby permitting various measures, such as the number of production rules fired, or the number of keystrokes, to be made. Kieras & Polson report that preliminary efforts with such a simulation produced useful information about the interface they were studying.

2.1.2. Empirical Evaluation Methods

In contrast to the analytical techniques are those which empirically measure user performance, satisfaction, or other usability aspects. Essentially, these methods involve placing a user in contact with the system and observing the results.

2.1.2.1. Formal Experimental Techniques

Formal techniques are rooted in the scientific method. Psychologists, seeking the rigor of the physical sciences, applied and adapted the experiment to their domain. Controlled experiments and statistical methods ensured the validity of results. Applied to the evaluation of interfaces, such techniques typically involve designing a statistically sound experiment, using benchmark tasks as a means of controlling bias. Differences in subject expertise are also considered.

The experiment is usually performed in a laboratory, where lighting, temperature, and ambient noise can be held constant from subject to subject. Before the session starts, the subject often completes a questionnaire which seeks to classify the subject in relation to the other subjects or to some user population. The experiment is often videotaped, and the subject may be assigned a standard set of tasks to complete. During the session, the experimenter is required to provide each subject with the same information and must be careful not to ask leading questions. At the end of the session, the user may be asked to answer an exit questionnaire.

The data are later analyzed for verbal protocols, critical incidents, time to accomplish the benchmark tasks, number of errors, and other similar measures. This analysis typically involves viewing the videotape of the session. The time involved in the analysis is therefore at least as long as the session itself but can require many more times that amount. Indeed,

Mackay [Mackay, et al., 1988] has stated that for a one hour session, the analysis can take a day or more to perform.

2.1.2.2. Contextual Research

Whiteside, Bennett, & Holtzblatt [1987] state that a problem with experimental methods performed in a laboratory is the lack of a natural work context. For example, benchmark tasks reflect a third party's judgement about what is important to the user, "... the deliberate restricting of focus..., has the effect of making an a priori value judgement that these operations are at the heart of usability for the system." They advocate a contextual research method, a method of collecting data about the user's actual, everyday experience of using the system.

One way of collecting such data is the contextual interview. This consists of (ideally) visiting users at their place of work, videotaping, and interviewing them as they work. The purpose of the interview is to capture the user's experience of using the system at the moment of that experience. The observer can immediately validate any interpretations by asking the user. In effect, data collection and analysis are performed concurrently.

In a discussion of this technique, Good [1988] stated that the time spent in such field evaluations is typically two to two and a half hours. This represents a thin slice of the user's daily experience with the system. It is also quite expensive to send a contextual research team to each site. An extended visit, although it would certainly reveal a lot more about the customer, would cost correspondingly more. Depending upon the questions to be answered, however, the costs may be justified.

Whiteside, Bennett, & Holtzblatt [1987] also suggest other methods of collecting data such as a gripe key, gripe address, or toll-free number. These methods provide a way for evaluators to discover problems that users have with the system.

2.2. RESEARCH SCOPE

Each of the methods outlined in the previous sections has its own strengths and weaknesses. Analytical techniques have the advantage of not requiring a prototype or running system; however their theoretical underpinnings are still at an early stage of development. The effects of interface constructs and task structure on usability are not yet well understood. Formal experimental techniques tend to provide very exact results, but about narrow and specific areas. However, there are few formal procedures for identifying those interface aspects that require experimental investigation. In addition, experiments are typically expensive and time-consuming. Contextual research techniques can provide results in less time and at less expense than formal experiments; however they are ad hoc, and the results can not be validated in the strict experimental sense. Both the formal experimental techniques and contextual research techniques are invasive as well, although in different ways: the experimental techniques pluck users out of their natural environments, whereas the contextual techniques insert the researcher into that environment.

Brooks [1988] believes that the empirical approach is required, and has noted the conflict between the last two techniques:

> A major issue perplexes and bedevils the computer-human interface community — the tension between
> - narrow truths proved convincingly by statistically sound experiments
> - broad "truths," generally applicable, but supported only by possibly unrepresentative observations.

and goes on to say

> We must always refine our interfaces by tests with real users, and we will always be surprised by those tests. ... Over-generalized findings from other designer's experiences are more apt to be right than the designer's uninformed intuition. ... Any data are better than none.

It is in this spirit of user testing that this research is undertaken. Consequently, this dissertation will focus on an empirical user interface evaluation technique — analyzing transcripts of user sessions.

Transcripts contain all user inputs and system outputs that occur during use at the user's site, and they represent the user's actual work. Transcripts provide excellent information about user performance to an investigator who knows where to look and what to measure. However, there is no general theory to help identify from a transcript those parts of a user interface that contain the potential problems that require more formal investigation. Indeed, the amount of information in a transcript is almost overwhelming, and its very completeness obscures its utility in identifying problems.

A number of indicators of usability have been known for some time and have led to techniques for locating interface problems. These include analysis of user errors, help system use, task performance times, command usage frequencies, and command transitions. Many programs have been written to extract and analyze such performance indicators. On the other hand there is little general software support for transcript analysis, and most of the tools that have been written have been task specific.

The work presented in this dissertation focuses on another indicator of user performance that may be extracted from transcripts. It is based on the hypothesis that repetitions of user actions may provide important clues to potential problems in the interface. The goal of the

research is to determine the feasibility and usefulness of such a method, that is, whether or not repetition analysis tools can be used effectively to detect problems in a user interface.

Chapter III
COMPUTER-BASED INTERFACE ANALYSIS

3.1. LOGGING USER SESSIONS

Traditional methods of data collection involve videotaping the subject. It is well known that this can be a daunting process because of the large amount of time and effort required to analyze the videotape. Even with the advent of analysis tools such as Mackay's [Mackay, et al., 1988], the process can still be tedious.

An alternative data collection technique is that of logging the user session. In general, user input is captured to a file with the aim of automating the data analysis. The advantages of this technique are numerous. Since the data are stored in on-line files, they are accessible to algorithmic analysis and data reduction techniques. Such analytic tools liberate the evaluator from the tedium associated with analysis, thereby encouraging the use of evaluation. Because data collection is automatic, there is no need for an observer or experimenter to be present. This absolutely eliminates any interference due to an observer, and means that data can be collected *in situ,* rather than at a laboratory, and collected from many users at the same time. The analytic tools also enable rapid analysis, thus providing quick feedback to designers or immediate debriefing of subjects. The analytic tools also make possible certain analyses involving vast quantities of data, which would never have been considered with videotape.

Certainly this technique does not capture as much data as videotape does. For that matter, neither videotape nor this technique can capture user *intent*. The question, however, is whether logging user sessions can capture enough data to be useful. In addition, it is essential that the analysis be automated in some fashion, since merely recording user input replaces the hours of videotape with megabytes of files.

3.2. EXAMPLES

There are several examples of this technique in the literature. Cohill & Ehrich [1983] describe a set of programs and routines developed to collect keystrokes and state information, and compress the raw data. Because of limited resources, they could not afford the usual data collection procedures. Their budget allowed for only one experimenter, and the schedule required running two to four subjects at a time. Since the variables they were measuring (time spent in help, number of times help was invoked, frequency of command use) did not require human judgement, an automated technique was used. They inserted calls to the metering routines at strategic points in the code of the system they were investigating, then using their tools, reduced the resulting data to a form suitable for immediate input to SAS [SAS Institute, 1979]. They found the tools they developed to be "extremely convenient," and report that it was better to collect as much data as needed, and reduce that data, rather than skimp on data collection, despite the large amounts of data involved.

As part of the special services of DMS, an early UIMS, Ehrich [1982] provided similar logging routines that tool builders could use in order to provide interface developers with logging services. Olsen & Halversen [1988] also implemented this concept, and studied the issues involved in such an enterprise. Their metrics included performance time, mouse motion, command frequency, command pair frequency, number of physical and logical input events, and visual and physical device swapping. The interface profile used by their UIMS to generate the interface is also used by the metrics computation and report generation program to generate human-readable reports. The reports present, for each metric, a list of commands ranked from worst to best for that metric. This report is then used to detect problems in the interface.

Neal & Simons [1983] utilize a different approach. They set up one computer to intercept, record, and time-stamp each keystroke. The keystroke was then passed to the other computer which ran the application system. An advantage of this method is that no modifications to the application system are required. Analysis was performed by "replaying" the keystroke file, that is, the file was used like a videotape. Unlike videotape, the observer was able to annotate the logfile directly with observations (e.g., critical incidents) or comments, while reviewing the session. The system also provided some analysis help in the form of frequency of occurrence of critical incidents, the time between such incidents, or an incident and the next user keystroke, frequency of use of commands or function keys, time spent in help, total session time, and number of help requests. Neal & Simons found their methodology to be "... very effective for objective evaluation and comparison of software including the user interface design and software documentation."

Good [1985] analyzed existing logging data collected at numerous sites, representing the use of five different text editors. The results were used in the design of a new text editor. Transition frequencies between keystrokes were used to aid in the layout of keys, e.g., the inverted-T layout of the cursor keys .The command set was based on command usage frequencies. The new editor itself was instrumented to log commands, and usage data were collected to determine the judiciousness of the design decisions taken, and to provide feedback for the next design iteration. Good claims that the use of logging data resulted in a measurably easy to use text editor.

Hanson, Kraut, & Farber [1984] collected command usage data on UNIX™ and applied creative statistical analysis techniques to the data. They determined a core set of commands by studying command frequencies. They also constructed a command transition matrix

---

UNIX is a trademark of Bell Laboratories.

from the data. From this matrix, and by applying multivariate grouping techniques, they were able to determine the modularity of each command, i.e., the degree to which a command is used together with many other commands. Treating this same matrix as a contingency matrix enabled Hanson, Kraut, & Farber to determine the sequential dependencies among commands. They prescribed some restructuring of the UNIX interface based on these results.

The examples presented above have a common set of analyses. These revolve around the reduction of data to summary results, such as command usage frequencies, command or keystroke transition frequencies, or time spent in a certain state. Such measures are convenient, but do not completely reflect the kind of information a person can collect from a transcript by reading it.

Reading transcripts does more than convince the reader about the need for analysis tools. One immediately notices certain patterns of command usage, i.e., a sequence of commands that the user repeats. A major contribution of this research is a new way to extract information from transcripts: an algorithm which scans transcripts and detects such repeating patterns.

3.3. REPEATING PATTERNS: THE REPETITION HYPOTHESIS

> A person acts so as to attain his goals through rational action, given the structure of the task and his inputs of information, and bounded by limitations on his knowledge and processing ability...
>
> – The Rationality Principle [Card, Moran, & Newell, 1983]

This research will assume that user behavior at a computer is purposeful, and hence that the user carries out a sequence of tasks to achieve some goal. The user accomplishes tasks by manipulating the computer's input devices and monitoring its output devices in a manner

dictated by the user interface. The user enters commands and data, and observes the resulting output. If the results are satisfactory then the task was accomplished. If the results are not, or an error occurs, the user must respond in some other fashion, for the task was not accomplished. This flow of input, resulting output, and input in response, is recorded in the transcript. Therefore it can be reasonably assumed that the transcript reflects sequences of tasks users carry out. Furthermore, inasmuch as these sequences of actions are made possible, and are partly governed by the interface, the quality of the user's interaction with the system is also reflected by the transcript.

In order to detect those tasks, this research hypothesizes that *repeated* sequences of actions in the transcript indicate a task rather than a random sequence of user actions. Moreover, the greater the number of repetitions of a sequence, the greater the likelihood that the sequence actually represents a task. Repeated sequences of user actions are therefore behaviorally interesting.

Repetition is also interesting of itself. Each action a user performs takes a certain amount of time and involves a chance for errors. Repeating such actions increases both the performance time and the chances for errors. By detecting frequently repeating actions and providing macros for them, it may be possible to reduce performance times and errors.

When errors are generated during the course of repeated actions, it may be the case that users are having problems with a particular task. Users may be trying variations of a command (e.g., changing the order of command arguments) in an attempt to make it work. Such repetitions might indicate problems with, for example, the command syntax or help system of the interface.

This research therefore hypothesizes that repeated sequences of user actions are interesting sequences and may indicate problems with the user interface. The ability to detect such repeating patterns may therefore prove useful to interface evaluators.

Chapter IV
MAXIMAL REPEATING PATTERNS (MRPS)

This chapter will define maximal repeating patterns, mrps, and present an algorithm for detecting such patterns in a user session transcript.

A user session transcript is the complete record of user input actions and system responses generated as a result of using the system. User input actions are extracted from this transcript, and represent the time-ordered sequence of input actions performed by the user. In a command line interface this extract consists of lines of command strings. Other interface styles, such as direct manipulation [Shneiderman, 1983], will have different types of user actions. Such actions, however, can be represented in some textual fashion [Siochi & Hartson, 1989], and thus should yield to the techniques described here. It is therefore sufficient to examine only command line interfaces as a preliminary investigation.

4.1. PROBLEM REDUCTION

The problem of detecting repeating sequences of command strings in the extract is equivalent to detecting repeated sequences of characters in a string, where each character in the string represents a command. The complete string then represents the entire extract, and a task would be represented by some substring. Since the interest of this research is on *sequences* of actions, then only substrings of length at least two will be considered.

The problem of detecting repeated substrings presents some difficulties. Consider the string "abcabc". The repeating substrings are "abc", "ab", and "bc". Which repeating substrings should be reported? Recalling that each character in the string really represents some user action, the question is in fact "which substring or pattern is behaviorally interesting?"

Because "ab" and "bc" are substrings of the repeating substring "abc," it is more efficient to report just the substring "abc," since any substring of a repeating substring must also repeat. Apart from the computational expense of finding and reporting all repeating substrings, and the expense of analyzing the prodigious volume of data that would be produced, it is doubtful that reporting "ab" and "bc" in addition to "abc" yields more information than reporting just "abc" as the repeating substring.

Consider further the string "abababab." Is the user performing 4 sets of "ab," or 3 sets of "aba" or "bab," or 2 sets of "abab," etc.? This question cannot be answered from a purely syntactic point of view, but requires knowledge of the semantics of "a" and "b." By reporting only longest repeating substrings, "ababab" in this case, it is possible for the analyst to study whatever other substrings as required, and not be inundated with data. The detection algorithm will therefore be confined to *maximal repeating patterns*, which are now defined.

4.2. DEFINITIONS

A repeating pattern is a substring which appears at more than one place in a string. A maximal repeating pattern, mrp, is a repeating pattern that is as long as possible, else that occurs independently. For example, in the string "abcdyabcdxabc," the substrings "ab", "abc", "abcd", "bc", "bcd", and "cd" are repeating patterns, whereas only "abcd" and "abc" are maximal repeating patterns. "abcd" is an mrp because it is not a substring of any repeating pattern, i.e., it is of maximal length. "abc" is an mrp even though it is a substring of "abcd" because "abc" also appears independently of "abcd", i.e., after the "x." An independently occurring repeating pattern is included in the definition because it may represent a task which is at once a main task and a part of a longer main task. A formal definition is now presented.

Let S be a finite string over an alphabet, $\Sigma = \{ a_1, a_2, ..., a_z \}$. The *length* of S is denoted lg(S).

Let $S = a_0a_1a_2...a_{n-1}$, $a_i \in \Sigma$; thus lg(S) = n. S[i,i+k] denotes the substring of S consisting of the ith through $(i+k)^{th}$ characters of S. If α = S[i,i+k], then α *occurs* at position i. If α = S[j,j+k] as well, then α occurs at position j also, and α is a *repeating pattern* of S.

Let α and β be repeating patterns of S. The *position set* of α, $P(\alpha) = \{ p_1, p_2,..., p_k \}$, is the set of positions at which α occurs. The *span set* of β,

$$SP(\beta) = \bigcup_{p \in P(\beta)} \{ i \,/\, p \leq i < p + lg(\beta) \}$$

is the set of all positions in S which are occupied by the characters of β. Thus if S = "abcdxabcdab," P("ab") = {0,5,9}, SP("ab") = {0,1,5,6,9,10}, and the complement of SP("ab") is {2,3,4,7,8}. Notice that $\overline{SP(\beta)}$ is the set of positions in S which are not occupied by the characters of β. *Thus a substring of β which occurs in this set occurs independently of β.*

Definition. A repeating pattern, α, of S is a *maximal repeating pattern (mrp)* of S, if

1) there exists no β, susbtring of S, such that α is a substring of β, or

2) if such a β exists, then $P(\alpha) \cap \overline{SP(\beta)} \neq$ nil.

Thus for S = "abcdxabcdab", "ab" is an mrp, but "abc" is not:

Let α = ab and β = abcd; then $P(\alpha)$ = {0, 5, 9} and $P(\beta)$ = {0, 5}.

$SP(\beta)$ = {0,1,2,3,5,6,7,8} so $\overline{SP(\beta)}$ = {4,9,10}, hence,

$P(\alpha) \cap \overline{SP(\beta)} = \{0,5,9\} \cap \{4,9,10\} = \{9\}$, so α = "ab" is an mrp.

Let α = "abc" and β as above. $P(\alpha) = \{0,5\}$, thus

$P(\alpha) \cap \overline{SP(\beta)} = \{0,5\} \cap \{4,9,10\}$ = nil, so α = "abc" is not an mrp.

4.3. ALGORITHMIC COMPLEXITY

The detection of mrps, *and reporting the positions at which they occur*, can be a very expensive task. The problem is not one of searching for a specified substring, but one of determining whether patterns exist, and if they do, reporting where they occur. A brute-force algorithm would search the string S for all repeating substrings of length i, varying i from 2 to lg(S)-1, and then to apply criteria 2 of the mrp definition, as illustrated in Figure 1.

Counting each invocation of the compare function as one unit, there are

$$\sum_{m=0}^{i-1} 1$$

or i units. The number of units, U, in the entire algorithm is thus

$$U = \sum_{i=2}^{n-1}\left(\sum_{k=0}^{n-i-1}\left(\sum_{j=k+1}^{n-i} i\right)\right)$$

$$= \sum_{i=2}^{n-1}\left(\sum_{k=0}^{n-i-1} [(n-i-k)i]\right)$$

$$= \sum_{i=2}^{n-1}\left[i(n-i)\left(\frac{n-i+1}{2}\right)\right]$$

which simplifies to

$$\frac{n^4 + 2n^3 - 13n^2 + 10n}{24}$$

and is of order $O(n^4)$, which is for the repeating substring extraction step alone! Admittedly the algorithm could be modified so that once a character comparison fails, the rest of the characters in the substrings are not compared. This still leaves a worst case complexity of $O(n^4)$, where all the characters of S are the same. The best case complexity, $O(n^3)$, would be when only one comparison is made for each substring compared. This occurs when each character in S is distinct:

$$U = \sum_{i=2}^{n-1}\left(\sum_{k=0}^{n-i-1}\left(\sum_{j=k+1}^{n-i} 1\right)\right)$$

$$= \sum_{i=2}^{n-1}\left(\sum_{k=0}^{n-i-1} (n-i-k)\right)$$

$$= \sum_{i=2}^{n-1}\left[(n-i)\left(\frac{n-i+1}{2}\right)\right]$$

which simplifies to

$$\frac{4n^3 - 9n^2 + 5n + 6}{48}$$

Unfortunately, such an algorithm may be too time-consuming for an interface analyst. If it takes too long to generate the list of mrps, the analyst may be discouraged from using the detection algorithm.

4.4. POSITION TREES

A position tree is a data structure that represents a certain class of substrings of a string, and the positions at which they occur. The tree has many uses, among which are determining whether an arbitrary string is a substring of the string from which the tree was constructed, and finding the longest repeating substring. The particular variation of the position tree that will be presented here can be constructed from a string in $O(n^2)$ time. In the worst case, there are $O(n^2)$ nodes in the tree. There is a variation called a compact position tree, which can be constructed in $O(n)$ time and has $O(n)$ nodes [Aho, Hopcroft, & Ullman, 1974].

4.4.1. Position Identifiers

Let $S \in \Sigma^*$, Σ a finite alphabet. A substring, α, *identifies* position i in S if α occurs only at position i. If α is the shortest possible such substring, then α is the *position identifier* of i. It has been shown [Aho, Hopcroft, & Ullman, 1974; Weiner, 1973] that every position in S$, where $ is a character not found in S, is uniquely identified by a position identifier. The set $I = \{\alpha_i / \alpha_i$ is the position identifier of i and $0 \leq i \leq lg(S)\}$ is the set of all position identifiers of S$.

4.4.2. Labelled Trees

A *labelled tree*, T, is defined as $T = (N, \Sigma, L, n_0, \hat{\delta})$, where

N is the set of all nodes in the tree

Σ is the alphabet from which the labels are drawn

$L = N \times \Sigma \times N$, is the set of labelled arcs in the tree, where if $(n,a,m) \in L$ then there exists no $k \in N$ such that $(k,a,m) \in L$.

```
/ *
** find all repeating substrings of length i
** append each repeating substring to the list of repeating substrings
* /
procedure bruteForce
{
int  i;      /* length of a substring */
int  k;      /* position of leftmost length i substring in the current comparison pass */
int  j;      /* position of the length i substring being compared to substring at k */
int  m;      /* offset of the position currently being compared in the 2 substrings */

for (i=2; i ≤ n-1; i++)          /* for each length from 2 to n-1 */
  for (k=0; k ≤ n-i-1; k++)      /* start each scan from position k*/
    for (j=k+1; j ≤ n-i; j++)    /* compare each succeeding substring with the one at k */
     for (m=0; m < i; m++)       /* compare each character in the two substrings */
       compare (S[k+m], S[j+m]);
/* delete non-maximal substrings from the list of repeating substrings */
deleteNonMax ();

}/* end bruteForce */
```

Figure 1. A brute force algorithm to find all repeating substrings.

$n_0 \in N$, is the root of the tree, and

$\hat{\delta}: N \times \Sigma^* \rightarrow N$, is defined as

$$\hat{\delta}(n, \alpha) = \begin{cases} n, \text{ if } \alpha = \lambda \\ \hat{\delta}(\delta(n, a), \beta), \text{ if } \alpha = a\beta \end{cases}$$

where $\delta: N \times \Sigma \rightarrow N$ is the successor function

$$\delta(n,a) = \begin{cases} \varnothing \text{ if } \nexists\, m \in N / (n, a, m) \in L \\ m \text{ if } \exists\, m \in N / (n, a, m) \in L \end{cases}$$

and

1) $\delta(n,a) \neq n$ for all $n \in N$, $a \in \Sigma$

2) $\delta(n,a) \neq n_0$ for all $n \in N$, $a \in \Sigma$

3) if $\delta(n,a) = m$, then no such $k \in N$ exists such that $\delta(k,a) = m$

with similar restrictions for $\hat{\delta}: N \times \Sigma^* \rightarrow N$.

Definition. $n \in N$ is a *leaf* of T if for all $a \in \Sigma$, $\delta(n,a) = \text{nil}$.

Definition. The predecessor function of a node returns the parent of that node:

$$\text{pred}(n) = \begin{cases} \varnothing, \text{ if } n = n_0 \\ m, \text{ iff } \delta(m, a) = n \text{ for some } a \in \Sigma \end{cases}$$

Definition. The *depth* of a node is given by:

$$\text{depth}(n) = \begin{cases} 0, \text{ if } n = n_0 \\ \text{depth}(\text{pred}(n)) + 1, \text{ if } n \neq n_0 \end{cases}$$

Definition. The *walk* of α, a substring of S, is defined as

$$w(\alpha) = \hat{\delta}(n_0, \alpha).$$

Now the position tree, T, constructed from a string, S, is a labelled tree whose leaves are in 1-1 correspondence with the positions of S, and whose arcs are labelled by elements of Σ. Each path from the root to a leaf corresponds to the position identifier for the position specified by that leaf, that is, $\{w(\alpha) / \alpha \in I\}$ is the set of leaves in T.

Figure 2 shows a string, the set I of position identifiers of S, and the position tree constructed from S. The rest of this section will introduce concepts and properties of position trees which are exploited in the mrp detection algorithm.

4.4.3. Properties

Lemma 1. A prefix of any repeating substring of S also repeats.

Let $\alpha\beta$ repeat in S. Clearly, α repeats as well.

Lemma 2. If α is a substring of S that does not repeat, then any substring of S, $\alpha\beta$, does not repeat either.

Let α occur exactly once in S. Assume there exists $\alpha\beta$, substring of S, such that $\alpha\beta$ occurs more than once in S. By Lemma 1, since α is a prefix of $\alpha\beta$, then α must repeat, a contradiction. Hence no such substring $\alpha\beta$ exists, and the lemma is true.

Theorem 1. Let $\alpha \in$ I, occurring at i. Any proper prefix of α repeats in S.

By definition, α is the shortest substring of S that occurs only at i. Let $\alpha = \beta\gamma$, $\gamma \neq \lambda$, so β occurs at i as well. Assume β does not repeat. But β is shorter

than α. This contradicts the premise that $\alpha \in I$, hence β must repeat, and the theorem is true.

Theorem 2. Any repeating substring α is a proper prefix of at least two position identifiers. If $B = \{\beta / \alpha\sigma = \beta, \beta \in I\} \rightarrow |B| \geq 2$.

Since α repeats, it occurs in at least 2 distinct positions, p and q. By the definition of I, there exist β and $\gamma \in I$ such that β occurs at p and γ at q. Consider β: either β is a proper prefix of α, or $\alpha = \beta$, or α is a proper prefix of β.

If β is a prefix of α, then because β does not repeat, by Lemma 2 α does not repeat. This contradicts the premise that α repeats, hence β is not a proper prefix of α.

If $\alpha = \beta$, then since α repeats, β repeats. This contradicts the fact that $\beta \in I$, hence $\alpha \neq \beta$.

Thus α is a proper prefix of β, and by symmetry α is a proper prefix of γ, hence $B = \{\beta / \alpha\sigma = \beta, \beta \in I\} \rightarrow |B| \geq 2$.

Theorem 3. Let α be a repeating substring of S, and T the position tree constructed from S. The set of all positions at which α occurs is equal to the set of leaves in the tree of $w(\alpha)$. That is, if $P(\alpha)$ is the set of positions at which α occurs, B as defined in Theorem 2, i.e., B is the set of position identifiers which have α as a proper prefix, and

$$L = \bigcup_{\beta \in B} \left\{ n / \hat{\delta}(w(\alpha), \gamma) = n, \beta = \alpha\gamma \right\}$$

then $P(\alpha) = L$.

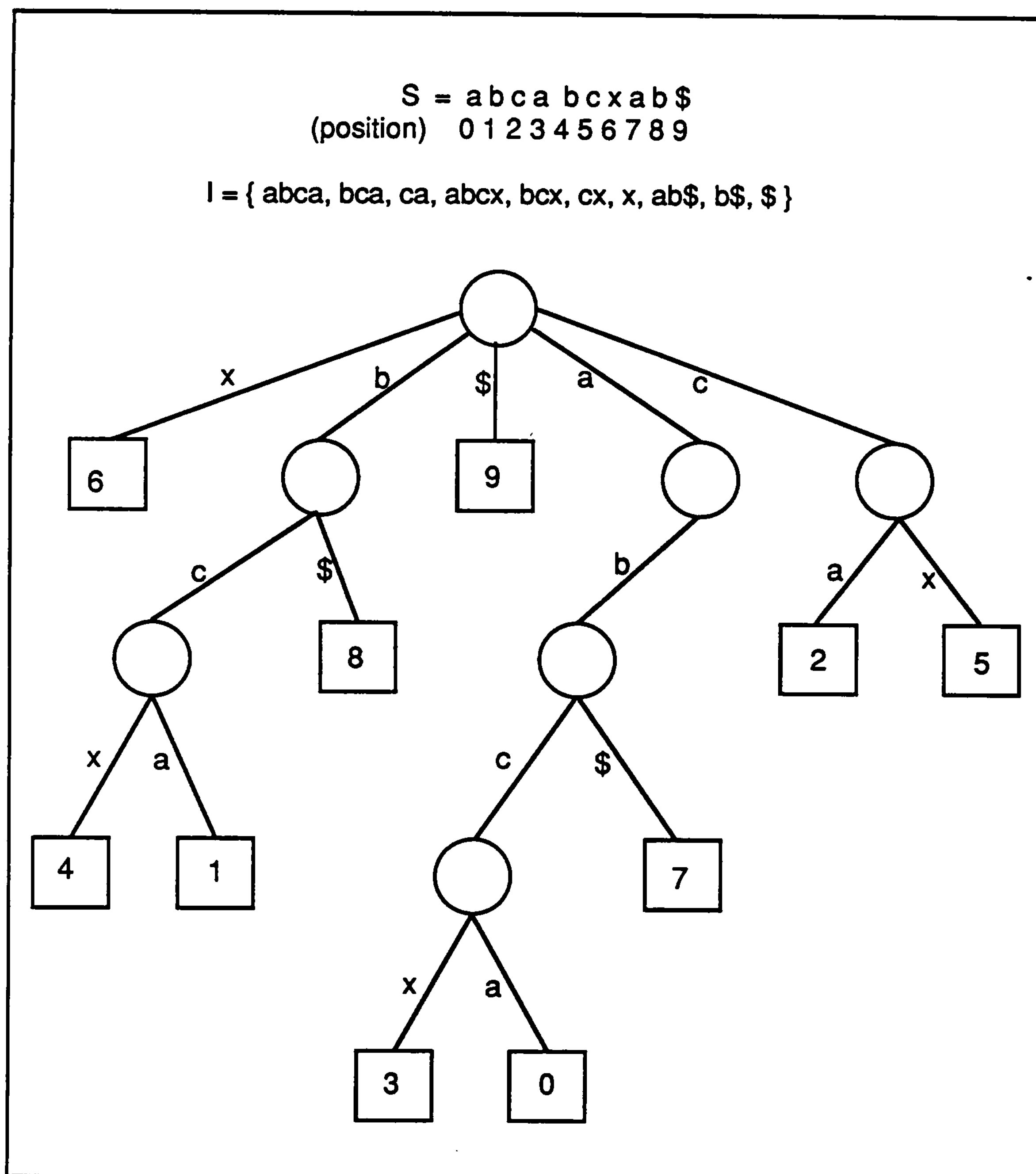

Figure 2. Position tree for the string S = “abcabcxab.”

Since α repeats, then by Theorem 2 α is a proper prefix of at least two position identifiers. For each $\beta \in B$, α occurs at the position where β occurs, hence

$$\bigcup_{\beta \in B} P(\beta) \subseteq P(\alpha)$$

But $P(\beta) = \{n \,/\, w(\beta) = n\}$

$$= \{n \,/\, \hat{\delta}(w(\alpha),\gamma) = n,\ \alpha\gamma = \beta\},$$

hence

$$L = \bigcup_{\beta \in B} \{n \,/\, \hat{\delta}(w(\alpha),\gamma) = n,\ \alpha\gamma = \beta\} \subseteq P(\alpha)$$

Assume that α occurs at some position $p \notin L$, i.e. that $P(\alpha) \not\subseteq L$. By definition of I, there exists $\beta \in I$ such that β occurs at p. Since α occurs at p, then either β is a proper prefix of α, or $\alpha = \beta$, or α is a proper prefix of β.

If β is a proper prefix of α, then α does not repeat by Lemma 2, contradicting the premise that α repeats, hence β is not a proper prefix of α.

If $\alpha = \beta$ then β repeats, contradicting $\beta \in I$, hence $\alpha \neq \beta$.

If α is a proper prefix of β, then since β also occurs at p, then $\beta \in B$, and

$$p \in \bigcup_{\beta \in B} P(\beta) \rightarrow p \in L$$

a contradiction to the assumption, therefore

$$P(\alpha) \subseteq L, \text{ hence } P(\alpha) = L.$$

Theorem 4. The set, R, of all repeating substrings of S is the set of substrings of S whose walks terminate at an interior node in T, the position tree constructed from S:

$$R = \{\alpha / \delta(w(\alpha),a) \neq \varnothing \text{ for some } a \in \Sigma\}$$

Let $K = \{\alpha / \alpha\gamma \in I, \gamma \neq \lambda\}$. By Theorem 1, for all $\alpha \in K$, α repeats in S, therefore

$$K \subseteq R.$$

Assume that there exists a $\beta \in R$ such that $\beta \notin K$, i.e., that $R \not\subseteq K$. But since β is a repeating string, then by Theorem 2 β is a proper prefix of some position identifier of S. Thus $\beta \in K$, a contradiction, and therefore $R \subseteq K$ hence $K = R$.

Now for $\alpha \in R$, by the definition of a position tree, $w(\alpha)$ is an interior node of T, that is, $\delta(w(\alpha),a) \neq$ nil for some $a \in \Sigma$, thus

$$R = \{\alpha / \delta(w(\alpha),a) \neq \varnothing \text{ for some } a \in \Sigma\}.$$

Lemma 3. The function

$$\phi(n, \alpha) = \begin{cases} n, \text{ if } \alpha = \lambda \\ \delta(\phi(n, \beta), c), \text{ if } \alpha = \beta c \end{cases}$$

is equivalent to the function $\hat{\delta}$, i.e., $\hat{\delta}(n, \alpha) = \phi(n, \alpha)$.

The proof is by induction on the length of α. If $\alpha = \lambda$, then $\hat{\delta}(n, \alpha) = \phi(n, \alpha) = n$. Let $lg(\alpha) = 1$, and assume without loss of generality that $\alpha = a$, then

$$\hat{\delta}(n, \alpha) = \delta(n, \alpha) = \phi(n, \alpha).$$

Assume the lemma is true for $lg(\alpha) = k$. Now let $lg(\alpha) = k + 1$, and $\alpha = a\beta c$. Thus:

$\hat{\delta}(n, \alpha) = \hat{\delta}(\delta(n, a), \beta c)$

$= \phi(\delta(n, a), \beta c)$ by inductive hypothesis, since $lg(\beta c) = k$.

$= \delta(\phi(\delta(n, a), \beta), c)$ by definition of ϕ.

$= \delta(\hat{\delta}(\delta(n, a), \beta), c)$ by inductive hypothesis, since $lg(\beta) < k$.

$= \delta(\hat{\delta}(n, a\beta), c)$ by definition of $\hat{\delta}$

$= \delta(\phi(n, a\beta), c)$ by inductive hypothesis, since $lg(a\beta) = k$.

$= \phi(n, a\beta c)$ by definition of ϕ.

$= \phi(n, \alpha)$

hence the lemma is true.

Lemma 4. $depth(w(\alpha)) = depth(w(\beta)) + 1$, for $\alpha = \beta c$.

By Lemma 3 $w(\alpha) = \delta(\phi(n_0,\beta), c) = \delta(w(\beta), b)$. Thus

$depth(w(\alpha)) = depth(pred(w(\alpha))) + 1$

$= depth(pred(\delta(w(\beta), b))) + 1$

$= depth(w(\beta)) + 1$

Corollary. Let $\alpha \in R$, then $lg(\alpha) = depth(w(\alpha))$.

Part 1. $lg(\alpha) = n \rightarrow depth(w(\alpha)) = n$:

Let $lg(\alpha) = 1$, and assume that $\alpha = a$. Then $pred(w(\alpha)) = n_0$, so

$$depth(w(\alpha)) = 1.$$

Assume part 1 is true for $lg(\alpha) = k$. Let $\alpha = \beta b$ such that $lg(\alpha) = k + 1$, so

$$k + 1 = lg(\beta) + lg(b) = lg(\beta) + 1$$

thus $lg(\beta) = k$, and by the inductive hypothesis, $depth(w(\beta)) = k$. By Lemma 4, $depth(w(\alpha)) = depth(w(\beta)) + 1$, thus $lg(\alpha) = n$ implies $depth(w(\alpha)) = n$.

Part 2. $depth(w(\alpha)) = n \rightarrow lg(\alpha) = n$:

Let $\alpha = \beta b$, $\beta = \lambda$, then $depth(w(\alpha)) = 1$. Now $lg(\alpha) = lg(\beta) + lg(b) =1$, thus part 2 is true for the base case. Assume it is true for $depth(w(\alpha)) = k$. Let $depth(w(\alpha)) = k + 1$.

By Lemma 4, $depth(w(\alpha)) = depth(w(\beta)) + 1$, thus $depth(w(\beta)) = k$, and by the inductive hypothesis, $lg(\beta) = k$. But $lg(\alpha) = lg(\beta) + 1$, so $lg(\alpha) = k + 1$, thus $depth(w(\alpha)) = n$ implies $lg(\alpha) = n$. Hence $lg(\alpha) = depth(w(\alpha))$.

From this corollary and Theorem 4, it is easily seen that a longest repeating substring of S corresponds to the path from the root of T to an interior node of greatest depth, i.e., the parent of a leaf of greatest depth.

4.5. THE DETECTION ALGORITHM

The detection algorithm uses a position tree to identify all repeating substrings of the extract string, and the positions at which they occur. These substrings and their associated positions are kept in a list sorted by the length of the substrings. The non-maximal repeating substrings are then culled from this list by applying condition 2 of the mrp definition. The general idea of the algorithm will be illustrated by detecting the mrps of the string S = "abcabcxab."

The position tree is constructed from the extract string using Weiner's algorithm D [Weiner, 1973]. Figure 2 (see the end of section 4.4.2) shows the resulting position tree. Note that the longest repeating substring, "abc," is an mrp since there is no repeating substring that contains it. Because a longest repeating substring of S corresponds to the path from the root to an interior node of greatest depth, the culling algorithm starts from a leaf, D, of greatest depth, and deletes leaves of lesser depth from the tree if the positions they represent are in the substring denoted by the path from the root to the parent of D. Thus if d is a leaf of lesser depth than D, and l is the length of the path associated with D, then d is deleted if $D < d < D+l$. Consider leaf 3 as an example. The path from the root to the parent of 3 is the substring "abc." Leaf 4 is deleted because $3 < 4 < 3 + 3$. Note that leaf 7 is not deleted. Once all leaves of lesser depth have been tested, the process is repeated using the leaf of next greatest depth. After all the leaves have been processed, any remaining leaves are the positions at which an mrp occurs, and the paths from the root to each of these remaining leaves are the mrps of S.

This algorithm works because it is possible to extract all the repeating substrings (and the positions at which they occur) from the position tree, and because the method of deleting leaves is equivalent to satisfying condition 2 of the mrp definition. The first part of this claim follows directly from Theorems 3 and 4. The second part requires more elaboration.

Let β be a longest repeating substring and α be a substring of β. $P(\beta)$ is the set of positions at which β occurs, and by Theorem 3 is the set of leaves in the subtree whose root is the terminal node of the walk of β.

Let $D \in P(\beta)$. Then

$$\{ i / D \leq i < D + lg(\beta) \} \subseteq SP(\beta).$$

Let $d \in P(\alpha)$. If there exists $D \in P(\beta)$ such that $D < d < D + lg(\beta)$ then

$d \in SP(\beta)$, and therefore $d \notin \overline{SP(\beta)}$. If for all $d \in P(\alpha)$, $d \notin \overline{SP(\beta)}$, then

$P(\alpha) \cap \overline{SP(\beta)} = nil$ and thus by condition 2 of the mrp definition, α is not an mrp.

The time complexity of this algorithm can be determined by analyzing the pseudo-code shown in Figure 3. Since the positions in S correspond to the leaves in the position tree, the position tree must be traversed in order to extract each leaf. As a leaf is extracted, it is insert-sorted into the depth list. This means that each leaf in the tree is visited once. Thus the time to produce the depth list is O(N), where N is the number of nodes in the tree. Since in the worst case, $N = n^2$, where n is the length of S [Weiner, 1973], then the function buildDepthList () has time complexity $O(n^2)$. In the second part of the algorithm, the maximum number of iterations of the inner "for" loop is less than n^2 as well, since the maximum number of positions in the depth list is n. The algorithm thus has a worst case time complexity of $O(n^2)$. Since the algorithm to construct the position tree has time complexity $O(n^2)$ as well, the complete algorithm has a worst case time complexity of $O(n^2)$, much less than the $O(n^4)$ worst case complexity for the brute-force method.

```
/ *
* *  given a position tree, extract the mrps from it
* /
procedure extractMRP (positionTree)
{
/* build the auxillary data structure (the depth list). This is a list of positions in S */
/* representing the repeating substrings of S, sorted by the length of the substrings.*/
depthList = nil;
depthList = buildDepthList (positionTree, depthList);

/* Eliminate positions in S based on the leaf deletion criteria explained earlier */
for each position in depthList {
  lb = computeLowerBound (position);
  ub = computeUpperBound (position);

  for each position in depthlist {
   if (lb < position < ub) then
     delete position from depthList;
   } /* end middle for loop */

  } /* end outer for loop */

} /* end extractMRP () */
```

Figure 3. Extracting mrps from a position tree.

Chapter V
TESTING MRP USEFULNESS

The usefulness of the mrps was determined by analyzing the interface of GIPSY, a large and complex system in regular use at the Spatial Data Analysis Laboratory, or SDA Lab, at Virginia Tech. GIPSY was selected as the testbed because of this fact, and because its users were known to complain about it being hard to use. Thus if the mrp analysis did not reveal any problems with the interface, it was unlikely that mrp analysis would prove useful in interface evaluation. If the mrps did point to problems with the interface, then that would indicate the method was promising, and that further research into the method should be undertaken.

5.1. GIPSY

GIPSY is an image processing system designed to run on the VAX™ series of computers [Garland & Ehrich, 1987]. At present, it supports over three hundred and fifty image processing algorithms, ranging from the classical to the most advanced. It is in use at numerous sites throughout the United States.

GIPSY has a command line interface, a help system, and a menu system. Several GIPSY commands are complex enough to have their own command line interface, resulting in highly modal interaction. In effect, GIPSY is a manager of several programs, some of which are simple and involve minimal or no interaction with the user, while others are quite complex and require a considerable amount of interaction with the user. For example, the command MHIST, which produces a histogram, returns immediately after it is invoked, while the command EXSIF, which allows the user to examine a standard image file, has so many sub-commands that it has its own command line interpreter. The user is in another

VAX is a trademark of Digital Equipment Corporation

mode when EXSIF is running, and must quit EXSIF to invoke other GIPSY commands. GIPSY also provides users with a means of passing operating system commands back to that system, thus allowing users to perform tasks such as file and disk management without leaving GIPSY.

5.2. DATA COLLECTION

GIPSY was modified to record all user keystrokes and system output on a user session basis. All keyboard and screen activity was recorded since it was not known precisely what information would prove useful.

The data were collected over three months, producing over twelve and a half megabytes or three hundred thousand lines of transcript. As the raw transcripts grew in size they were periodically moved to the machine on which the analysis would take place, and a new collection period was started. This was done to minimize disruption of normal SDA Lab operations. There were thus as many files as there were users for each collection period.

The user population of the SDA Lab at the time varied from expert GIPSY users to students in an introductory image processing class.

5.3. DATA ANALYSIS

The tools used in this test were a normalizer and the mrp tool. Figure 4 shows their use in the evaluation process. These tools were designed as proof-of-concept software, i.e., the design goal was to determine whether or not the mrps pointed to potential problems and enabled evaluators to develop insights about the interface they were analyzing. The tools themselves had the minimal usability required for the tests, e.g., the mrp tool had a simple command line interface with minimal help. As implemented, they would not be suitable for use in the daily operations of a software development organization.

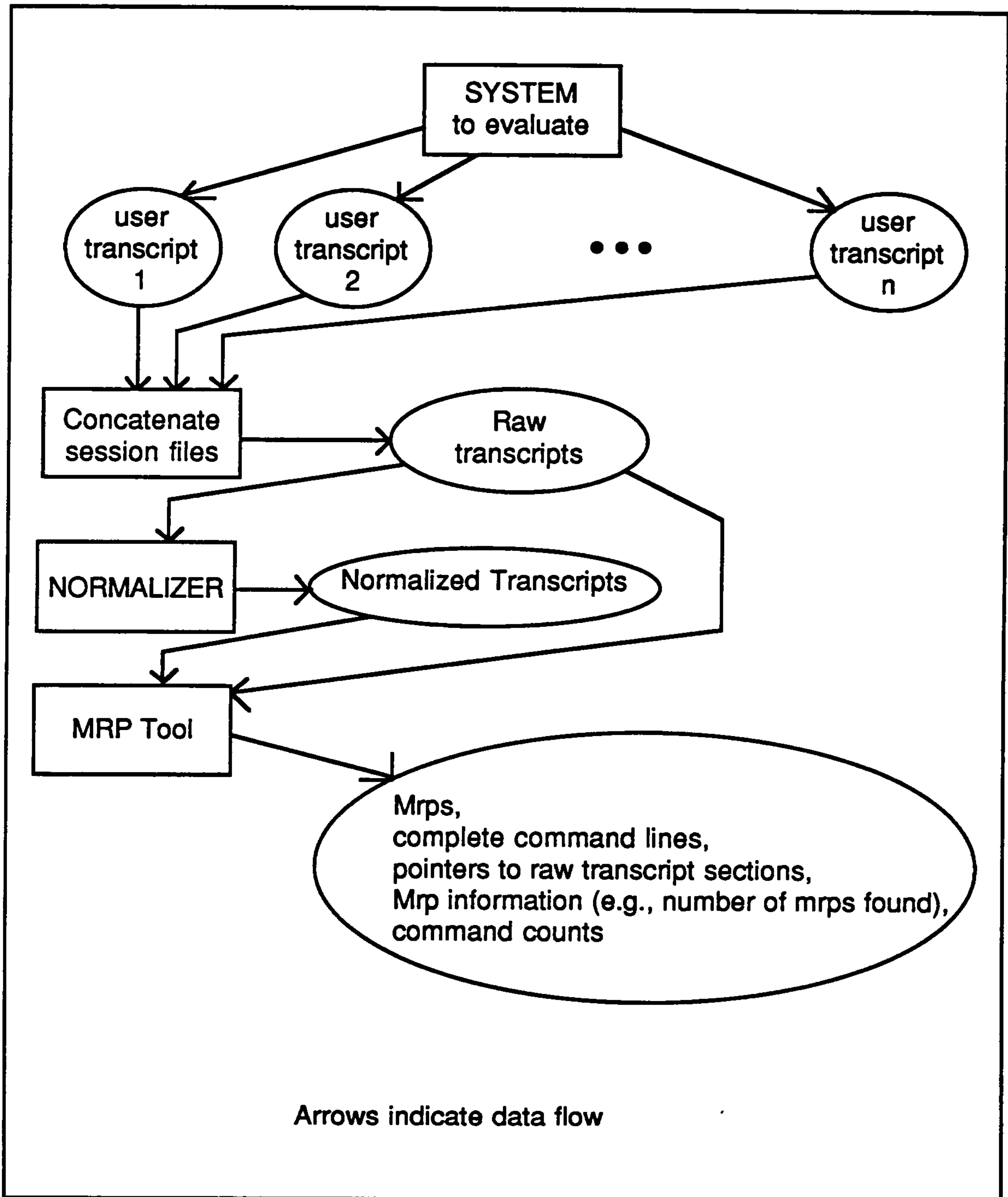

Figure 4. Mrp analysis tools and the evaluation process.

5.3.1. The Normalizer

The normalizer translates raw transcripts into a standard form that the mrp tool uses, thereby keeping the mrp analysis tool independent of the formats which data logging routines might use. As a result, the transcripts of any system can be analyzed without changing the mrp analyzer, provided a normalizer can be written for that system's transcripts. This technique is similar to that used in compiler construction, where the parser is kept machine independent by separating the code generator from the parser.

The normalizer consists of a couple of AWK programs and a C-shell script. AWK is a pattern scanning and processing language [Aho, Weinberger, & Kernighan, 1978], while C-shell is one of the command line interpreters available for the UNIX operating system. Figure 5 shows the transformations carried out by the AWK programs. One AWK program parses the raw transcripts to extract user input lines, and marks each extracted line with its line number in the original file. This enables the evaluator to locate the sections of the raw transcript indicated as interesting by the mrp tool. The second AWK program extracts the command portion of the input line, producing a file containing the sequence of commands invoked by the user. Each command appears on a separate line in this file, which is the input for the mrp tool. A C-shell script integrates the AWK programs so that the evaluator does not have to invoke each program separately.

In this version of the normalizer, only the top-level GIPSY commands are extracted. No subcommands, such as those of EXSIF, are included in the normalized transcript in order to keep the analysis simple.

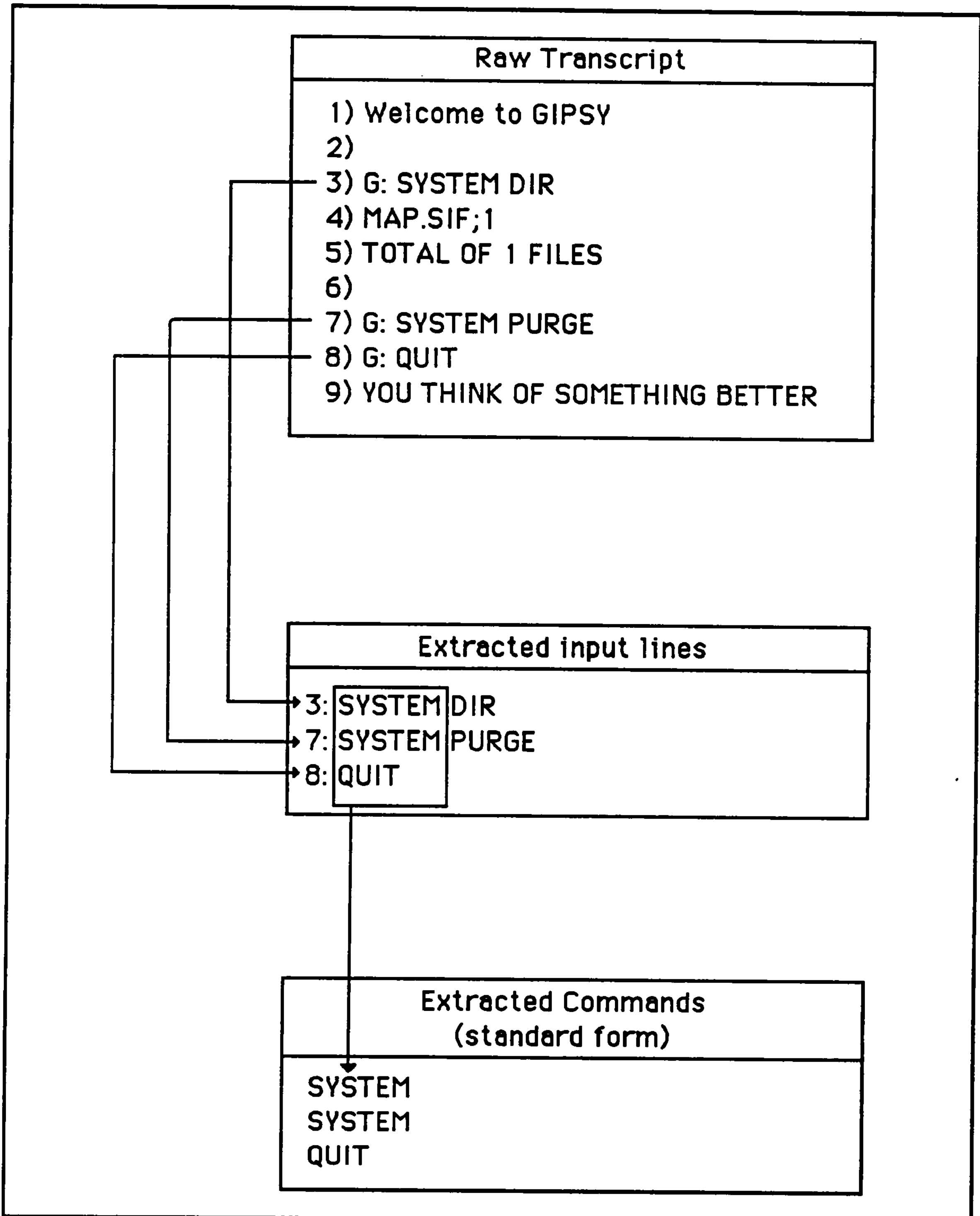

Figure 5. The normalizer converts raw transcripts to a standard form.

5.3.2. The Mrp Tool

The mrp tool allows an interface analyst to extract the mrps from a normalized transcript. The functions provided by this tool are listed in Table 1, and the tool's structure is shown in Figure 6. The tool generates a list of the mrps detected in the specified normalized transcript. This list can be traversed forwards and backwards, and individual mrps can be marked to allow later direct access to them. This list can be filtered, i.e., mrps from the list can be selected on the basis of their length, or the number of positions at which they occur. Information about the number of mrps detected, the maximum, average, and minimum lengths is also available. The combination of these two functions allows an evaluator to focus on certain mrps, e.g., mrps of length greater than five, or which occur more than the average number of times. Access to the complete command lines and to the raw transcripts is also provided by this tool, as is the ability to perform operating system commands (in this case, UNIX) from within the tool. There is also a limited form of macro capability, which when combined with the C-shell allows the same operations to be performed automatically on several different normalized transcripts.

Once in the mrp tool, an evaluator starts by issuing the mkpt command, specifying which normalized transcript file to use, then running the mkmrp command. At this point, the info command provides information about the mrps detected. If a filter command is used, a second mrp list is generated. Each successive invocation of the filter command is applied to the second list. The evaluator can switch between the main, unfiltered list, and the second list using the go command. The evaluator then studies the mrp lists, the complete command lines, and sections of the raw transcript. Any interesting information provided by the tool, e.g., the mrp list, can be saved to a file.

Table 1. Mrp tool functions.

- **mkpt** FILENAME: makes the position tree from the file named as argument
- **mkmrp** : makes the mrp list from the current position tree
- **show** X | first | next | prev | NUMBER | all | "": types the mrp marked with X, the first, next, or previous mrp, the nth mrp, all mrps, or the current mrp.
- **mark** ? | X <COMMENTS>: list the marks, or mark the current mrp with the character X and comments
- **filter** (num | len < | = | > NUMBER): filters the mrplist by number of occurrences or length of mrp
- **go** main | filter: goes to the main mrp list or the filter list
- **details** first | next | prev | all | "": types the command lines corresponding to the positions of the current mrp. arguments are similar to the show command
- **wid** NUMBER: set the number of lines to print before and after each mrp detail
- **raw** : view the raw transcript file
- **info** : show information about the current mrp list
- **stats** : show command usage statistics
- **save** <all> mrp | details | info | stats FILENAME: saves the current mrp, details, information, or statistics to FILENAME. if "all" is specified, saves all mrps, etc. *replaces* contents of existing file.
- **append** <all> mrp | details | info | stats FILENAME: same as save, except appends to FILENAME
- **readfrom** <FILENAME>: reads mrp tool commands from FILENAME. if argument is missing, reads from the keyboard
- **!** unix command string: shell escape; argument is a command line which is sent to UNIX for execution
- **#** <COMMENTS>: use this as first word on a line to mark the rest of the line as a comment.
- **quit** : exit the mrp tool

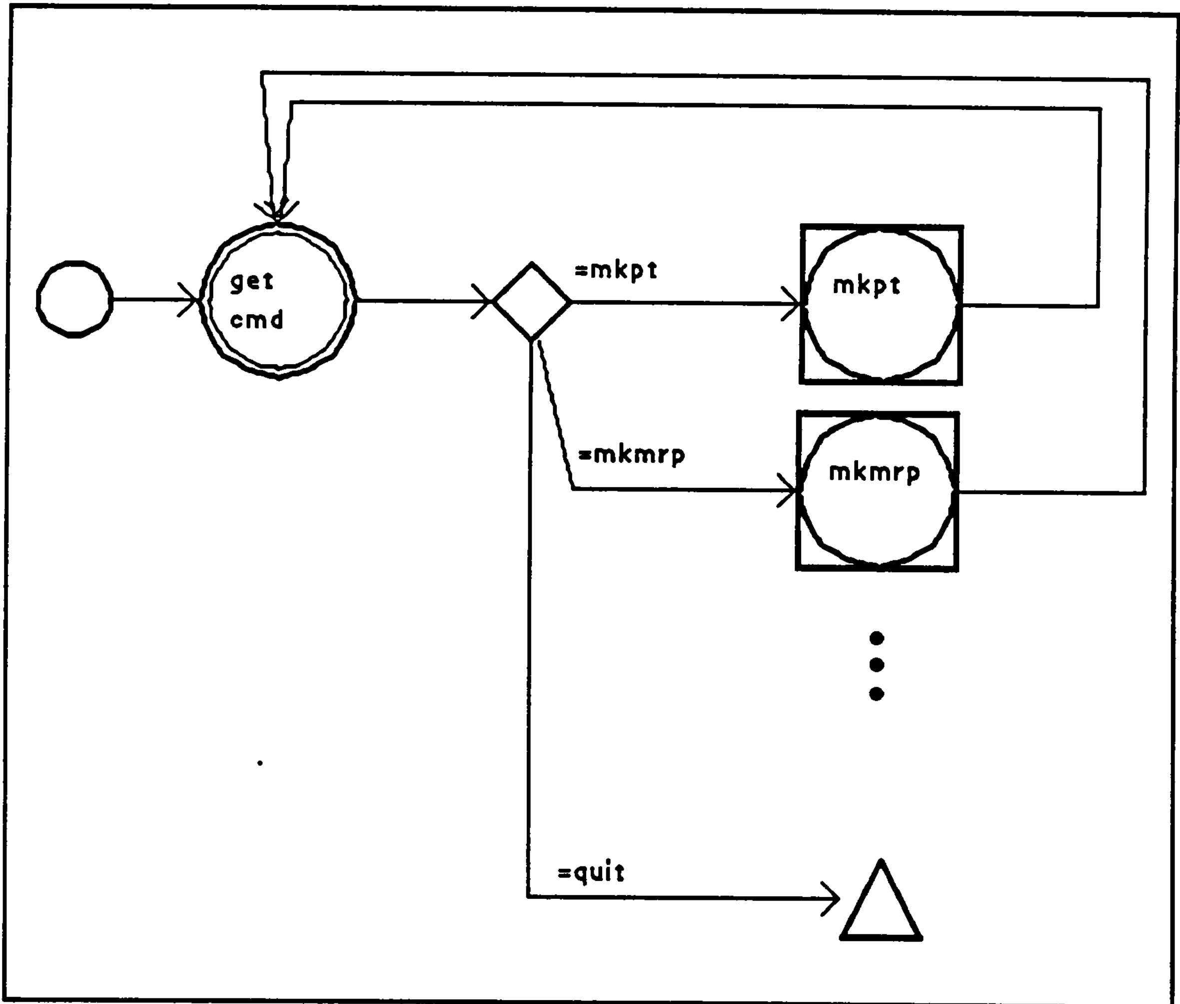

Figure 6. Highest level structure of the mrp tool.

A command occurrence table is generated as a by-product of the mrp detection operation. The evaluator uses a separate program to compute statistics such as usage frequency, number of sessions represented by the transcript, and maximum, average, and minimum number of commands executed per session.

It is important to realize that the mrp tool is not a "data summarizer," but an identifier of potentially interesting episodes in the transcript. The tool is valuable in two ways: an evaluator does not have to read the entire transcript to find repeating patterns, and the tool may also uncover patterns the evaluator might miss. It remains, however, the evaluator's job to determine the significance of individual mrps.

5.3.3. Procedure

Figure 7 shows the procedure used in the test. The details of the mrp analysis phase were shown in Figure 4. In this analysis phase, the first step was to concatenate in chronological order the collection period transcripts of each user into a single raw transcript representing all the sessions of that single user. This reduced the number of files to be analyzed, and allowed a single analysis across all sessions of that user. After the raw transcripts were translated into a standard format, mrps were extracted using the mrp tool. The mrp list was scanned by the evaluator, and where necessary, the complete command lines corresponding to the mrps were reviewed and the raw transcript sections corresponding to the mrps examined.

At this point a fair number of GIPSY problems were detected, however, several mrps appeared anomalous or insignificant. These were investigated by interviewing two GIPSY users. The evaluator generated a list of questions to ask selected users. The users were asked to validate the deductions made by the evaluator, or to explain certain episodes in the transcript which were indicated as interesting by the mrp tool but which appeared

anomalous to the evaluator. This step was essentially a debriefing structured by the results of the mrp analysis. The findings were then summarized and are presented in the next chapter.

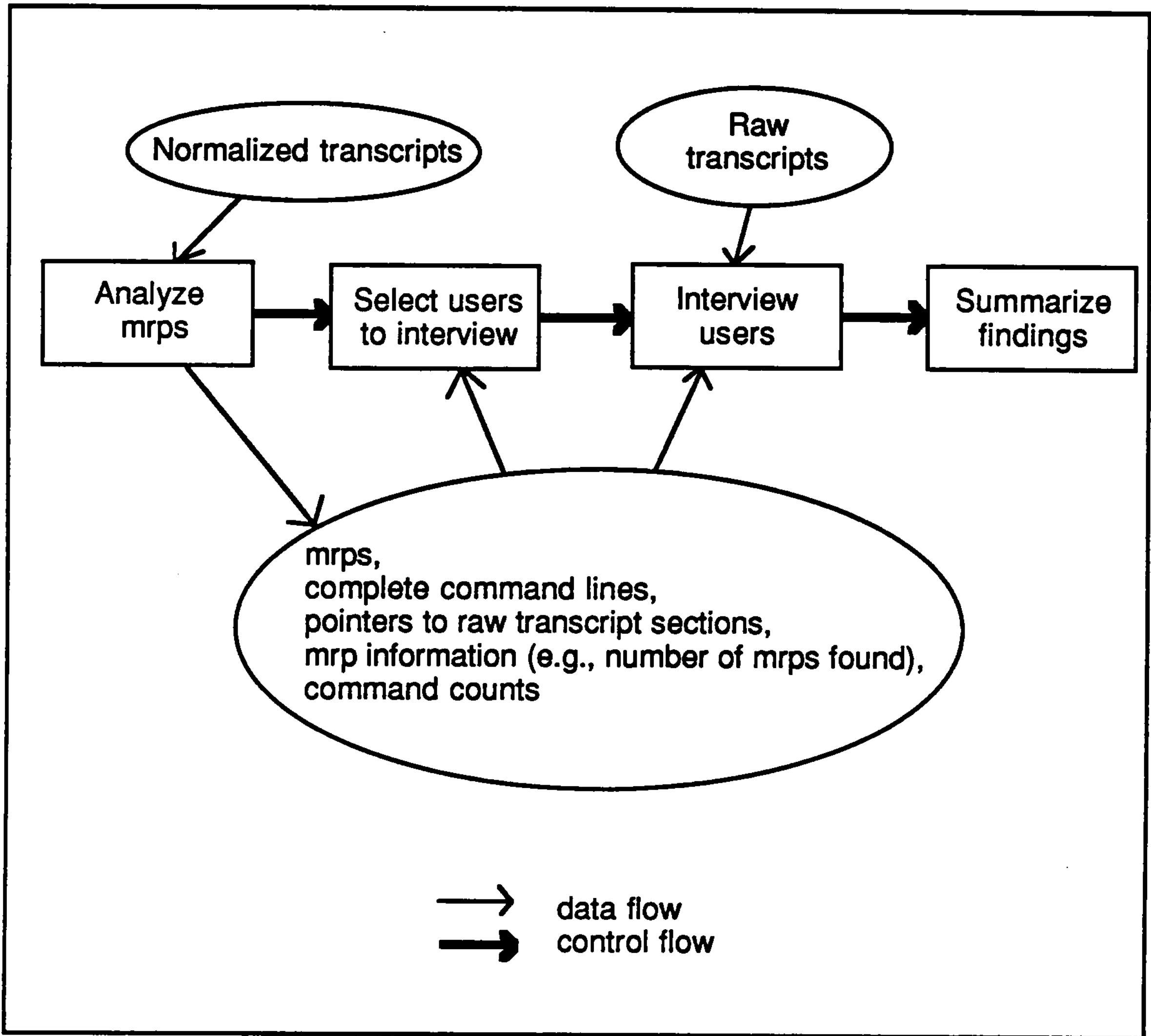

Figure 7. The Evaluation Procedure.

Chapter VI
RESULTS

The mrp tool was used to extract the list of mrps for each normalized transcript. An analysis of these lists revealed several interesting and specific problems with GIPSY, yet several questions remained. Two separate structured interviews of two users were conducted to answer those questions. The interviews validated certain deductions made in the first analysis and clarified the anomalies of that stage. Both stages in this analysis revealed twelve problems, two of which were known before the study was done.

6.1. MRP TOOL RESULTS

This section describes four problems which were discovered by using the mrp tool and analyzing the detected mrps. These problems involve consecutive invocations of the same command, system response time, macros, and GIPSY sessions where nothing was done.

The transcripts of seventeen users, totaling 17,086 command lines, were analyzed over an eight hour period. Table 2 gives, for each user, the number of mrps detected, the length of the longest mrp, and whether or not two types of mrps were found.

Some of the mrps were considered noteworthy on the basis of violating expected usage patterns of commands in general. For example, mrps consisting of consecutive invocations of the same command (see Figure 8) were detected in the transcripts of users. These are the type 1 mrps reported in Table 2. Type 1 mrps may indicate that a user needs to perform the same command on several objects, or that the user is "fine-tuning" a single object. For example, a user may have a list of files that need to be converted from one format to another, or a user may be debugging a runfile (GIPSY terminology for a macro). In the first case, a possible remedy could be to allow an arbitrary number of arguments for each such command. The second case demands a closer study of the nature of the "fine-tuning." The fact that 82% of the sample users exhibit this mrp type suggests that this indicates a problem inherent in the interface design, rather than a collection of user idiosyncrasies.

Table 2. Selected information from mrp analysis of seventeen transcripts.

User	Number of Lines	Number of Mrps	Length of longest Mrp	Type 1 Mrp ?	Type 2 Mrp ?	mrps per command line
U01	118	26	8	0	1	.22
U02	85	13	17	1	0	.15
U03	2728	520	26	1	0	.19
U04	76	10	9	0	1	.13
U05	50	13	4	1	0	.26
U06	223	34	6	1	1	.15
U07	220	41	6	1	0	.19
U08	4778	951	29	1	1	.20
U09	66	9	7	1	0	.14
U10	234	44	6	1	0	.19
U11	2867	681	21	1	1	.24
U12	22	3	6	1	0	.14
U13	121	14	26	0	0	.12
U14	1079	268	25	1	1	.25
U15	2248	476	19	1	1	.21
U16	1262	220	37	1	1	.17
U17	909	200	17	1	0	.22
Totals	17086	3523	269	14	8	average =.21

Average number of mrps per user = 207

Average length of longest mrp = 16

82% of users had type 1 mrp

47% of users had type 2 mrp

```
0) DSPLY

1) DSPLY

2) DSPLY

3) DSPLY

4) DSPLY

5) DSPLY

6) DSPLY

at:  656   657   658   1137   1138   1139   1565   1574...

Total number of positions =   8.
```

Figure 8. An instance of a type 1 mrp: consecutive invocations of the same command.

The commands which were found in type 1 mrps are the following (command definitions are from Garland & Ehrich [1987]):

BINSIF - Convert binary data to Standard Image File (SIF) format
CHRSIF - Convert a character file to a SIF file
DSPLY - Display an image in the SIF format
EXPL - Explain GIPSY commands
INVIMG - Invert the gray levels of an image
MHIST - Computes histogram of an integer image
RUN - Switch command input from terminal to RUN file
SBIMG - Extract a subimage
SIFCHR - Convert a SIF image into a character file
SYSTEM - Run local system command

Type 2 mrps consist of consecutive lines where no commands were entered (see Figure 9). These indicate anomalous use of the command line terminator, which for GIPSY is the carriage return. This mrp type may be due to factors such as poor keyboard design, defective keyboards, or long response times. It may also be due to objects falling on the keyboard and striking the return key. However, the high proportion of users who exhibit this mrp, and the long experience designers have with keyboards suggest that system response time is the more likely cause.

In the mrps that follow, the notation "X A > B" means that the user invoked command X using file A as input and file B as output. Notice that most of these mrps use the output of one command as input to a succeeding command. Such mrps suggest the development of specific macros, and even identify the parameters and local variables of the macro, in addition to the sequence of commands that make up its body.

```
 0)

 1)

 2)

 3)

 4)

 5)

 6)

at:   511   667   668   669   670   671   672   673...

Total number of positions =   21.
```

Figure 9. An instance of a type 2 mrp: empty command lines.

```
235 times:
CHRSIF A > B
EXSIF  B
DSPLY  B
```

This mrp indicates that a user tends to modify a file that was just converted with CHRSIF. The DSPLY command is used to verify the modifications. Because this mrp occurs a great many times, a macro which combines these commands is desirable.

```
73 times:
DSPLY A
DELETE A
```

This mrp demonstrates a deletion strategy of the user: invoke the DSPLY command before deleting a file to confirm that the file should be deleted. This could easily be incorporated as an option for the DELETE command. Note that the user is not necessarily searching for which file to delete, since the exact file name is specified and therefore known. A searching behavior would have been indicated by the use of the SYSTEM DIRECTORY command to list the files available, before invoking the DELETE command.

```
36 times:
BINSIF A > A
DSPLY A
```

```
26 times:
CHRSIF A > B
PLTSIF  B > C
PRINT  C
```

```
19 times:
SOBEL A > B
DSPLY B
```

```
13 times:
TRSLD  A > B
PLTSYM B > C
```

These four mrps, as well as the previous two, show the user confirming the effects of an image manipulation command. They indicate the strong need for feedback in the interface (one of the previously known problems). Instead of typing the DSPLY command each time, the image manipulation commands could have an option to automatically redisplay the image after processing it. This would cut down on the amount of typing the user has to do, thus reducing the time between image manipulation and feedback.

```
29 times:
STOP
STOP
```

The last mrp is highly unusual because it shows that in several sessions users did not invoke any GIPSY commands, i.e., users would run GIPSY and then quit. It is quite likely that users simply forgot to perform some command in the local operating system, or simply decided to do something else.

6.2. STRUCTURED INTERVIEW RESULTS

After the studying the mrp tool results, it became clear that some mrps could only be explained by asking users what they had been doing at the time. This section describes nine problems (one of these was already identified in the first stage) studied by interviewing two users. The nine involve the commands EZPLOT, PRTPF, LWPIC, and REPL, the frequent need for local operating system commands, the response time and error messages of the SYSTEM command, the failure of the SD local system command from within GIPSY, the GIPSY parser, and GIPSY runfiles.

Two of the seventeen users, U17 and U14, were selected to be interviewed approximately one year after the data were collected. Both users were GIPSY experts and did GIPSY development work. The basis for the selection was availability, since most of the users could no longer be contacted, e.g., the students in the image processing class.

In each interview, the user was presented the list of mrps detected in that user's transcript and asked to remember what he had been doing or trying to do. When viewing the mrps alone, both users could not remember what they had been doing, but when the mrp tool was used to show the complete command lines corresponding to the mrps, they were able to remember some tasks. When shown sections of the raw transcript corresponding to the

mrps, both users remembered almost all the tasks. This is a remarkable result, considering the time that had elapsed.

6.2.1. U17 Comments

User U17 was interviewed for approximately one and a half hours. The first question asked of U17 involved the following mrp:

```
mrp#   0
  0)   STOP
  1)   EDGEX2
  2)   STOP
  3)   EZPLOT
  4)   STOP
  5)   EDGEX2
  6)   STOP
  7)   EZPLOT
  8)   STOP
  9)   EDGEX2
 10)   STOP
 11)   EZPLOT
 12)   STOP
 13)   EDGEX2
 14)   STOP
 15)   EZPLOT
 16)   STOP
at:    636   640
Total number of positions =   2.
```

This mrp showed a series of sessions where single commands were invoked. Calculations based on the number of sessions represented by the transcript and the total number of commands indicated an average of three commands per session. When questioned about this, U17 remarked that he used a lot of batch files.

While viewing this mrp, U17 also remarked that EZPLOT did not have good support for multiple plots; although a series of plots can be made, EZPLOT computes a minmax range

only for the first in that series, and assumes that this range is to be used for the rest of the plots. This mrp thus also shows compensatory behavior — the use of batch files to perform multiple plots.

U17 also had several mrps which involved the SYSTEM command. These turned out to be used for file manipulation and disk maintenance operations, e.g., purging or deleting files to create more free disk space, indicating a real need for operating system functions from within GIPSY.

6.2.2. U14 Comments

A most interesting use of the complete detail provided by the raw transcript came about in the investigation of this mrp:

```
mrp#    1
  0)    RUN REORDER.RUN
  1)    PRTPF
  2)    %P
  3)    %P
  4)    %P
  5)    %P
  6)    %P
  7)    %P
  8)    %P
  9)    %P
 10)    %P
at:     251        266
Total number of positions =    2.
```

The %P command is a command that allows the user to repeat a previous command. In this case the raw transcript indicated that U14 was using the PRTPF command to test for the length of a file. U14 performed a binary search by successively specifying values to the PRTPF command depending upon whether an error (attempt to access a non-existent

record) occurred. The command that provides information about property files did not show how long a file was. This led to the binary search behavior of U14. When U14 was questioned about this, he verified the deduction of a binary search and its use to determine a file's length.

Another interesting mrp from U14 was

```
ARITHM    A > B
EXSIF     B
LWPIC     B > C
```

This mrp occurs eight times in U14's logfile. U14 is apparently modifying file A before printing it out. Debriefing U14 confirmed this: U14 stated that LWPIC could not print binary image files, so U14 had to edit the binary image files first. Further probing revealed that EXSIF has a subcommand, REPL, which allows a user to substitute one pixel value with another, over all pixels. The new value, however, is limited by the maximum value among the pixels. Because LWPIC could not print binary image files, the user needed to increase the contrast in a binary image (i.e., maximum pixel value = 1) and thus could not use the REPL subcommand of EXSIF. U14 had to use another command, ARITHM, to increase contrast in order to print the image. In addition this user would often use a text editor on the postscript file generated by LWPIC to increase the scale of the image. This indicates a need for the LWPIC command to accept two arguments — contrast and scale.

Note that although the mrp did identify the need for a macro, in this case the macro was *compensatory* behavior. The real problem was traced to the LWPIC command. This result illustrates the fact that mrps identify repeating usage patterns, but not the *reasons* for the repetition, nor solutions to the problems identified by the repetition. Examination of the contexts in which the mrp occurs is required to deduce those reasons.

U14 was also asked about the type 1 mrp where EXPL was the repeated command. This was considered an anomaly as U14 was an expert user. U14 replied that other users would ask him questions about GIPSY, and he used EXPL to help him answer those questions. This is an interesting result because it shows that (presumably) naive users would ask other users questions rather than use the help system. This merits further investigation to determine whether there are inadequacies in the help system that stimulate this behavior.

6.2.3. Common Comments

Some problems appeared in both interviews. One common mrp was "stop;stop" indicating a session where no GIPSY commands were invoked. A reasonable guess was that the user forgot to do something before invoking GIPSY. This was interesting because of the high frequency of the SYSTEM command, which allowed users to pass commands from GIPSY to the local operating system. Probing both users on this point revealed that they perceived the response time for the SYSTEM command to be excessive, and had developed a decision rule to the effect that for some commands, it is faster to leave GIPSY, invoke the operating system commands, then return to GIPSY, than to use the GIPSY SYSTEM command.

This mrp prompted both users to make further specific comments about the interface. For example, both users said that another reason for not using the SYSTEM command was that the error messages it returns are not as clear as those returned by the operating system: "It takes you a while to realize what's wrong." U14 started to recall several other problems with the SYSTEM command. He stated that SD, a local system command, did not work when called via the SYSTEM command. Also discovered, while scanning the complete command lines, was a problem with calling the operating system command, DIRECTORY/SINCE=1-MAR-1988, via the SYSTEM command. GIPSY apparently

interprets this as a GIPSY command, instead of a command to be passed on to the operating system, because GIPSY attempts to substitute the MARK command for this command. This is most likely due to a bug in the GIPSY parser.

Another problem both users complained about was that runfiles, i.e., GIPSY macros, caused GIPSY to terminate when the runfile terminates. U17 explained that mrp# 2 was an attempt to change this problematic behavior by making modifications to the runfile, then running it to see whether the modification stopped GIPSY from terminating when the runfile did.

```
mrp#   2
  0)   RUN OVER.RUN
  1)   $
  2)   RUN OVER.RUN
  3)   $
  4)   RUN OVER.RUN
  5)   $
  6)   RUN OVER.RUN
  7)   $
  8)   RUN OVER.RUN
  9)   $
 10)   RUN OVER.RUN
 11)   $
 12)   RUN OVER.RUN
 13)   $
at:     41  43
Total number of positions =   2.
```

6.3 KNOWN GIPSY PROBLEMS

Several GIPSY problems were known to the evaluator before the mrp analysis was performed. These are:

- difficult to find and determine which command to use
- help entries are not written well

- error messages not meaningful to user
- command names are poorly abbreviated
- command guesser often guesses wrong command
- slow input methods
- system response time can be slow
- poor feedback from image manipulation commands

Mrp analysis was able to detect the third and last of these problems. The error messages problem was detected when the mrp "stop;stop" was shown to the two users and both users stated that the error messages were difficult to understand (see section 6.2.3). The feedback problem was indicated by the six mrps involving the DSPLY, PLTSIF, and PLTSYM commands (see section 6.1).

Although only two of the previously known problems were detected, mrp analysis discovered ten *unknown* problems. This can be explained in part because the newly discovered problems are very specific, dealing with particular commands such as LWPIC. Such problems are usually discovered only after prolonged use, hence would not be found during a relatively brief evaluation of the system. GIPSY users also did not respond to requests from the developers for comments and problems users encountered [Ehrich, 1989], therefore specific problems were not likely to be known to the developers.

It is interesting to note that relatively few GIPSY problems were detected. There are two reasons for this result. First, GIPSY users tended to use relatively few commands, ranging from four to sixty commands out of a total three hundred fifty available. Because there were no usage data for the unused commands it was not possible detect errors associated with those commands. Second, a large number of mrps were produced. This forced the evaluator to examine only those mrps which occurred more than the average number of times. This reason is examined further in the next section.

6.4. PROBLEMS WITH THE TECHNIQUE

A problem with the technique was that the number of detected mrps reach into the hundreds for almost half of the transcript files. If this were the case in general, then the problem of tedium is resurrected. Table 2, however, indicates that on average one mrp will occur for every twenty one command lines. Figure 10 shows that the relationship between number of command lines in a transcript and number of detected mrps is linear. This means that part of the reason why so many mrps were detected is because there was so much information to begin with.

A related problem with the detected mrps is the large amount of noise present. Some mrps did not appear to indicate anything interesting about the interface, while other mrps were actually substrings of larger mrps, and repeated information provided by the longer mrp. The first case involved mrps which occurred at only two positions, thus this type of noise may be attributable to chance. Some sort of statistical filtering of the mrps, i.e., reporting only those occurring at greater than chance levels, may reduce this kind of noise. The second type of noise is due to the definition of an mrp and is probably the main contributing factor to the large number of mrps detected, considering the vastly greater number of shorter mrps (See Figures 11 and 12).

A potential data collection problem was also identified. The analysis tool assumes that the normalized transcript represents the strict chronological sequence of commands a user invokes during an interactive session. The chronological sequencing assumption can be violated if a user can log in from more than one terminal at the same time. When session data are recorded to the transcript file, the data from both GIPSY sessions may be interleaved. The interactiveness assumption can be violated in systems in which users posses the capability of submitting batch jobs, as they do with GIPSY (as illustrated by the

mrp# 0 anomaly of U17). In addition, there is little this approach can do about masquerades, i.e., someone other than the legitimate user logging on as that user. These are problems that should be addressed in the design of the data collection routines, which ideally should be part of the operating system or the UIMS used to build the interface.

A final problem involves systems where user-definable commands are allowed. How does one differentiate between native system and user-written commands? This is especially difficult if the command is a native one modified by a user, and that user kept the same command name, as U17 did with the EDGE command. Short of interviewing the user, there is no expedient way to determine whether or not a command is a modified version of a native command. This problem, however, is not as severe as may be imagined. Whatever the function of that command, what matters is how it is used, and this is revealed by the mrp analysis.

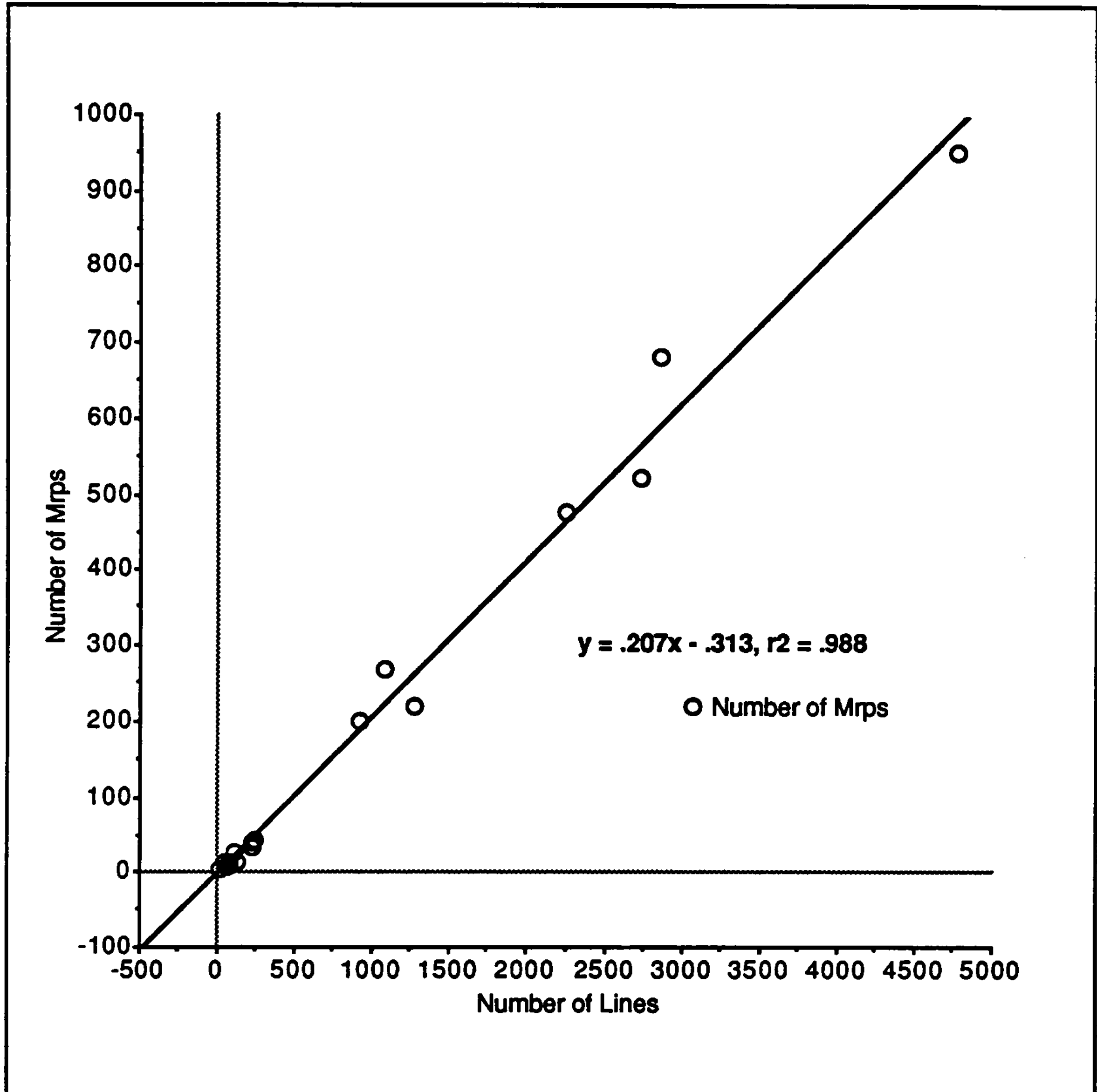

Figure 10. Plot of number of lines vs. number of mrps detected.

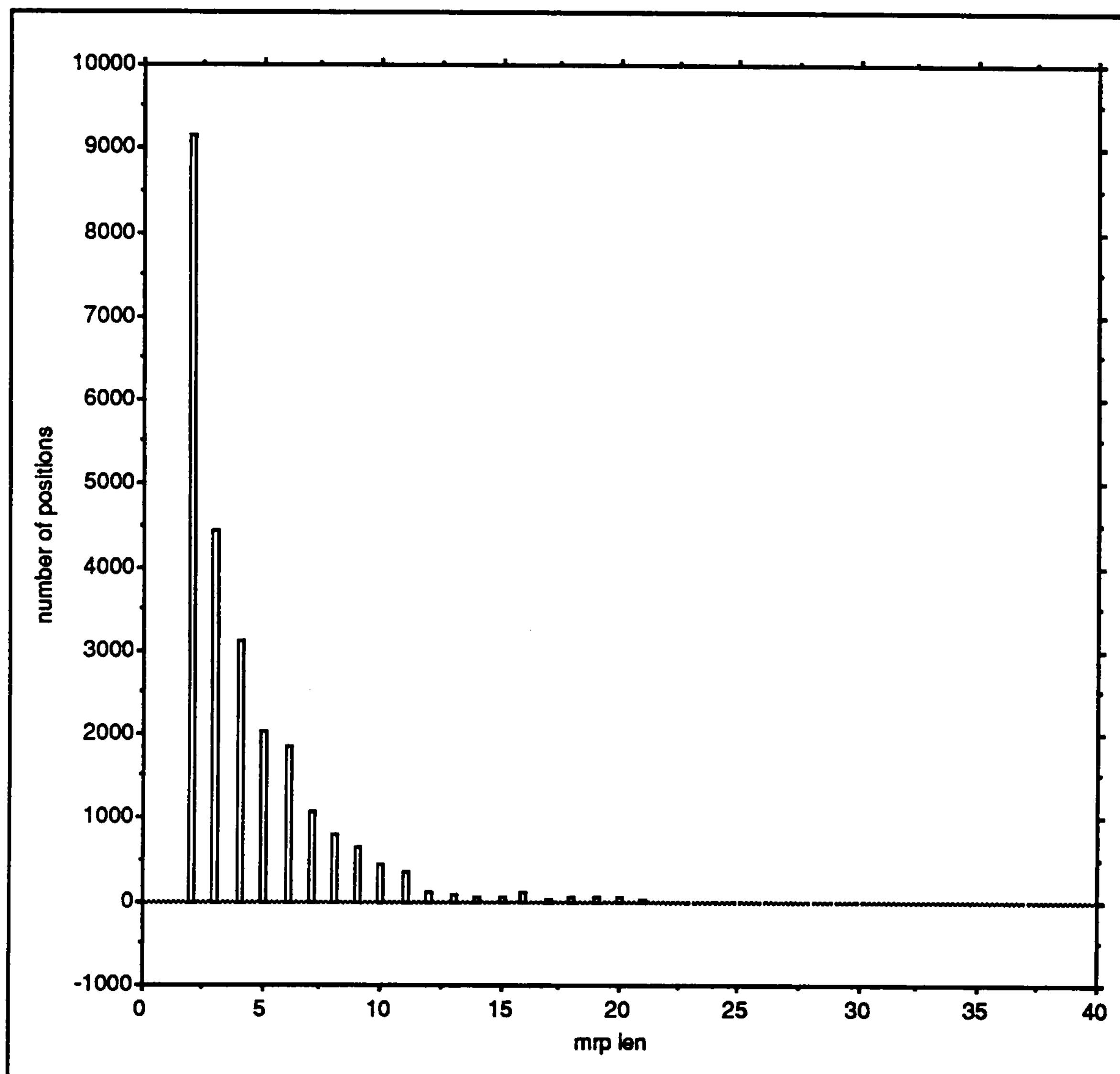

Figure 11. Distribution of number of positions an mrp occurs at according to its length.

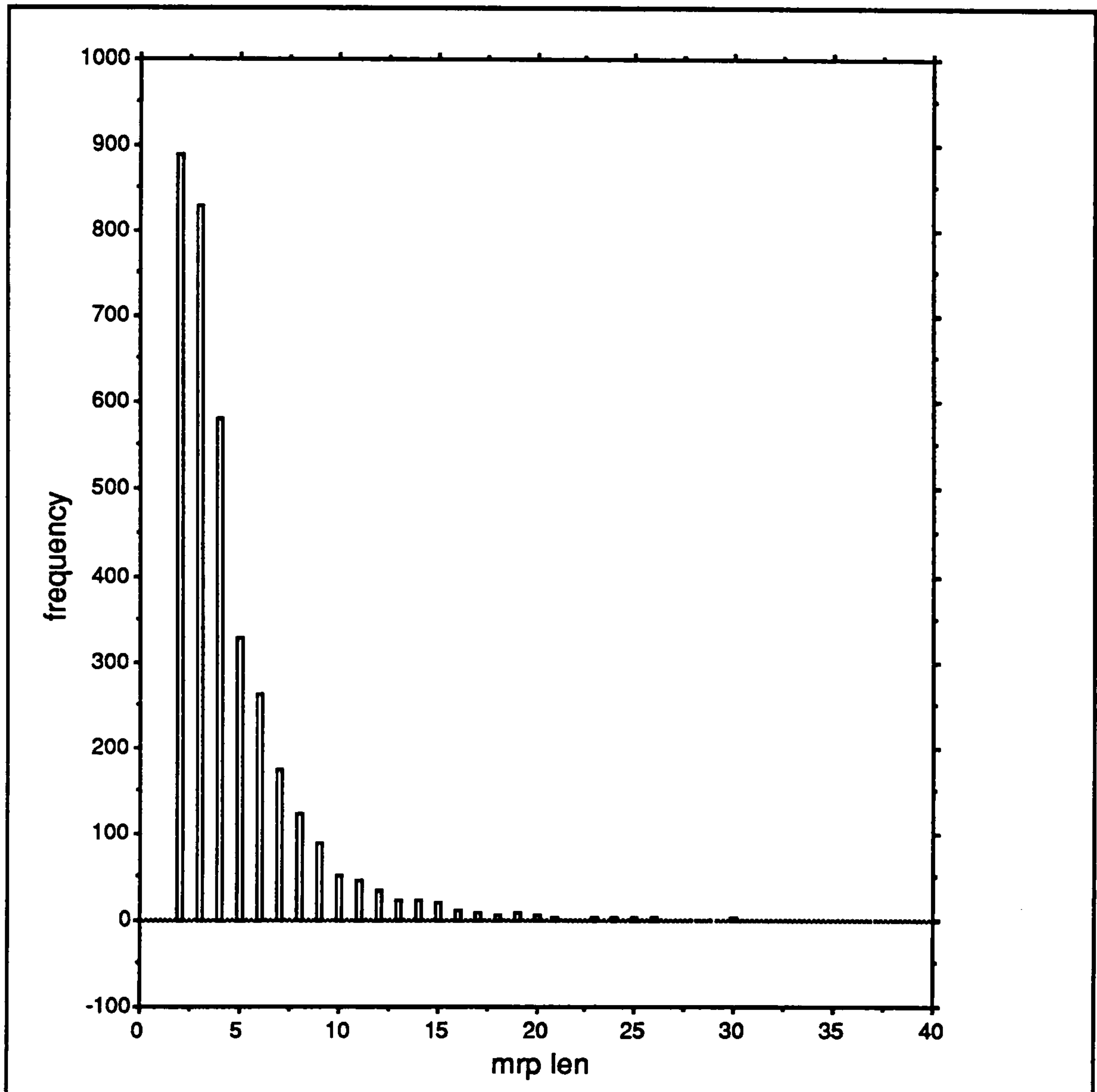

Figure 12. Distribution of frequency of an mrp according to its length.

Chapter VII
CONCLUSIONS

The results demonstrate that the technique was useful for finding specific problems in the GIPSY interface, e.g., the commands found in type 1 mrps. The mrp algorithm found repeating patterns of user actions in the transcripts, and these patterns indicated aspects of the GIPSY user interface which needed attention. Although the mrps did not show the root cause of a problem, much less indicate solutions, the mrps did identify specific, real problems. It is therefore reasonable to expect that this technique would work for other command line-based systems.

The technique provides information about the detailed work-a-day aspects of a system, but does not directly show general problems with that system. However, such detailed aspects are also important, consequently the technique should be part of a larger evaluation methodology.

7.1. ADVANTAGES

The most important benefits provided by this technique are its speed and specificity of results. Mrp analysis allows rapid identification of sections of the raw transcript which potentially show *specific* difficulties users were experiencing with the system. This means that vast amounts of data can be analyzed in a short time. Consider the transcripts analyzed in this research — these represented three calendar months of *actual* use of seventeen users. The seventeen thousand command lines were analyzed in about eight hours, and revealed twelve *specific* GIPSY problems, ten of which were previously unknown.

These two benefits made the technique useful in preparing the structured interviews. For each of the two users interviewed, approximately one thousand commands in two hundred

sessions were analyzed. The mrp algorithm took only a few seconds per transcript to detect the mrps. In addition, the complete command lines and the sections of raw transcript identified by the mrps enabled users to recall what they had been doing, and reminded them of problems they had with the system. Although for the interview the resulting mrps had to be studied, taking about one and a half hours per mrp list, and users debriefed, taking about one and a half hours each, this still represents a tremendous savings of time if the equivalent observations had been made by videotape!

The speed of the mrp tool in providing access to information such as mrps, complete command lines, and raw data was very useful during the interviews as well. The interviews depended extensively on this information, and when a busy user has graciously volunteered some time, it is important that the user not be kept waiting.

The specificity of the problems identified by this technique can be an invaluable aid to the programmers maintaining the code. For example, it is now known that the SD command does not work when issued from within GIPSY, or that the GIPSY parser fails to interpret correctly the SYSTEM DIRECTORY/SINCE command. In addition, the ability to point to and present raw transcript sections corresponding to problems is also useful when those problems are to be fixed. This information is a tremendous advantage in debugging and is not usually provided by user complaints. Consider the problems with the SYSTEM command — the person "debugging" this command has access not only to the section of transcript where the problem occurred, but also to the commands and system responses that preceded and succeeded it.

It is important to realize that these data represent the experience of users in their natural work context. As such, the information is a source for the discovery of how users actually use the system.

7.2. LIMITATIONS

The central limitation of the technique is the type of information it provides. The mrps focus on specific, detailed problems encountered by users. General aspects of the usability of the system are not directly exposed by this technique. For example, although this technique identified the previously known GIPSY problems of inappropriate error messages and poor feedback, it did not show the well-known deficiency of GIPSY, which is the great difficulty users have in finding out which command to use for the task they wish to perform (see section 6.3). However, the mere fact that GIPSY has more than three hundred and fifty commands should immediately raise concerns about the accessibility of those commands. This type of information is readily discernible from even a short exposure to GIPSY, or casual conversation with users. It is the details of everyday use that are missed in such dialogue, probably because users have adapted to those small inconveniences and thus do not talk about them. It is precisely those details which the mrp technique addresses and has been shown to detect.

One might argue that since users have adapted to those problems, it would not have been cost-effective to fix them. This statement has some validity, yet it ignores the fact that such adaptation has associated costs in terms of increased performance times and lower user satisfaction. Each adaptation is a set of tasks the user has to perform either to avoid some undesirable interface behavior or to effect some desired functionality. The extra time involved in such operations cannot be denied.

A more detailed limitation is that studying the mrps alone does not produce as much insight as studying the complete command lines since the mrps show only the command names, while the complete command lines show the arguments as well. Similarly, the interviews were able to generate more information about the interface. In general, the technique does

not use a lot of other information that could be part of a transcript file. For example, error patterns, help usage, user think and performance times, and system response times could all be recorded on the transcript. In fact, some of the detected mrps showed classes of usage patterns such as repeated invocations of a command on an object, or pipelining the output of one command into the input of another. A broader evaluation tool would encompass these diverse elements.

Another limiting characteristic is the need for analytical acumen. The analysis depends on the evaluator's skill and knowledge of both the evaluation method and the application system being evaluated. Human judgement is still required in deciding which mrps to examine further, in making deductions, and in proposing changes. Also, a sequence of commands may be repeated often, but the conclusion is not necessarily that a macro is needed. The repetition may be a user-adaptation to a different interface problem. Because human judgement is involved, a complete automation of the technique is not possible with current technology.

As presently implemented, this method is not suitable for incorporation into the software development process of commercial software producers. The signal to noise ratio of the algorithm must be improved, and a data collection, management, analysis, and results reporting infrastructure must be developed and set in place with the developers. As with any transcript-based method, the mrp technique as currently available is suitable mainly for the beta test and maintenance phases of the software lifecycle, but could probably be adapted to provide feedback for early design work.

Finally, the technique was applied to a system with a command line interface, ignoring the important new class of direct manipulation interfaces. This was a deliberate decision to

simplify the problem, especially since the viability of the technique was the object of investigation.

It is important to remember that even with these difficulties, several real problems with the GIPSY user interface were uncovered. This shows that the methods described in this dissertation were useful and at the very least merit further research.

7.3. RECOMMENDATIONS

From the previous discussion it can be seen that mrp analysis is good for working at the detailed level of interfaces, but not at a general level. At this point, mrp analysis would be most useful in the summative evaluation, beta-test, and maintenance phases of software development. The information provided by analyzing actual use data from several sites over an extended period of time should serve as excellent feedback to those maintaining the current version.

The rest of this section will outline several recommendations to address the limitations and problems of the technique. The action items involve a study of the problem of large numbers of mrps, a formal investigation of mrps, a proposed architecture for a broader transcript analysis tool, and a discussion of how direct manipulation interfaces might be studied with mrp analysis.

7.3.1. Controlling the Mrp Flood

The mrp tool provided detailed information about interface problems by pointing to transcript sections that were analyzed by both the evaluator and two users. This is valuable information, but considering the graph of Figure 10, a large scale study would inundate the evaluator with mrps. This can be addressed by reducing the number of mrps to study.

One way to implement this solution would be to limit the amount of data collected, as is eventually done. The maximum amount of data that would be collected can be estimated by first determining how many mrps the evaluator can analyze in the time scheduled. This number is then substituted for the variable y in the equation shown in Figure 10, and that equation solved for x, the number of command lines a user types.

Another way is to develop filtering algorithms or heuristics which can be applied to the mrp list. A simple attempt at heuristics was implemented in the mrp tool, where the evaluator examined only those mrps whose lengths were greater than average. A statistical filter was proposed in the previous chapter. Algorithms which reduce the amount of redundant information, and a re-evaluation of the mrp definition (specifically removing the independent occurrence condition) should be investigated.

This action item should be one of the first performed, since continued use of the technique would be greatly facilitated by any improvements in controlling the number and presentation of mrps.

7.3.2. Formal Investigation

This research studied the feasibility of using mrps as indicators of problems in a user interface. Because the research was preliminary in nature and its context was the practical side of software development, the concern was with the usefulness of the information obtained, rather than the statistical or psychological validity of the mrps.

Since the expectations about the usefulness of mrps in detecting problems in the user interface have been met, formal experiments should now be performed to test the repetition hypothesis and the correlation of mrps to tasks. A likely experimental scenario might consist of having subjects perform a series of benchmark tasks, indicating on the system

when they are starting and finishing tasks, or experiencing difficulty. Users could have a panic button available which they are instructed to press when they run into trouble. The button would place a marker in the transcript, indicating a problem occurred. The transcripts would be analyzed, and the positions in the transcript indicated as interesting by the mrps could be compared to the positions which subjects indicated as tasks, or were having difficulty. The sessions would also be analyzed by standard critical incident analysis and protocol analysis techniques, in order to provide independent comparisons. Another technique which could be used to determine the strength of mrps as problem indicators would be to provide users with intentionally flawed help or documentation. This eliminates the need for explicit user actions (e.g., pressing a panic button) to indicate trouble spots, since the experimenter knows where the problems will occur beforehand.

7.3.3. A Transcript Analyzer

Because of the central limitation of mrp analysis of transcripts, a broader transcript analyzer should be considered. This analyzer would provide the evaluator with a suite of analysis tools, of which the mrp analyzer is but one. Other tools that would be useful include pattern matching tools and statistical tools.

Pattern matching tools would allow the evaluator to search transcripts for usage patterns such as pipelines, consecutive invocations of the same command, or error-help triads. Pipelines and consecutive invocations were described in the previous chapter. An example of an error-help triad is

```
COMMAND_X          (error occurs)
HELP COMMAND_X     (help msg is displayed)
COMMAND_X          (error occurs again)
```

and is interesting because it indicates that the help message is not helping the user.

It is also not productive to have to write a program to scan for such patterns, since this effectively limits the number of patterns a busy evaluator could study. Some sort of pattern description language which allows the specification of variables in the pattern is required. Standard pattern matching algorithms are not sufficient because some of the items to be matched are variables. In the case of the error-help triad, COMMAND_X is a variable, since it is not known which commands are causing problems. A promising mechanism is Prolog, because it uses unification. Thus an evaluator might specify the error-help triad as

(*command.(help *command).*command)

where *command is a variable, the "." indicates concatenation of commands, and the parentheses indicate grouping.

Statistical tools would be similar to those described by Hanson, Kraut, & Farber [1984]. In addition to command usage frequency, information on number of commands per session, error rates, performance times and system response times would be provided. Metrics based on mrps should be investigated as indicators of interface quality. For example, the number of mrps detected per session: a higher than expected number of mrps per session may indicate problems with the interface.

The statistical tools should also facilitate the comparison of results across all users to allow the evaluator to make statements about the prevalence of certain problems among users, or whether a problem is specific to a few users only. This information is needed when developers have to decide which problems should be allocated scarce resources.

Another means of detecting problems in the transcript is a "panic button." This is a key the user presses whenever problems are encountered. As a result, a marker is inserted into the transcript. It is then a simple matter to find problem transcript sections.

Mrp analysis itself should also be expanded to study user performance times and error conditions. Command lines could be tagged with the time at which a user enters a command. This information could be used in mrp analysis to determine whether the mrp was performed mechanically (short time intervals between commands), or thoughtfully (longer than average time intervals between commands). The latter may be indicative of problem solving behavior, e.g. learning how a command works by trial and error.

Command lines could also be tagged with a symbol to indicate that an error occurred as a result of the command. One benefit of such error tagging is the ability to select only those mrps which contained errors. More importantly, the error tag is a concise means of providing contextual information without having to access raw transcripts. Such information has already proved useful (refer to the section on binary search).

Error tagging could be used in other ways. For example, one could calculate an error-use ratio for each command. This is the number of times a given command occurs with an error divided by the total number of times that command is used.

The indicators of user problems listed here represent an initial set. More indicators need to be developed and investigated. In addition, this suite of tools deals only with analyzing transcripts and is useful only when there are transcripts to analyze. It should be part of an evaluation environment which enables an interface evaluator to analyze the interface at varying stages of development. This environment should in turn be an integrated part of a User Interface Management System in order to support the data collection, management and analysis activities of an evaluator. This has been argued for by Ehrich [1982], Olsen & Halversen [1988], and Hartson & Hix [1989]. The major benefit of this integration is that it gives explicit recognition to the essential nature of evaluation in the iterative development methodology.

7.3.4. Direct Manipulation Interfaces

The evaluation method was tested on a command line interface. More recent interfaces are of the direct manipulation variety, where commands are not typed in but are effected by manipulation of graphics objects on the screen. The problem with trying to analyze such interfaces is in recording and representing user actions. At the lowest level, these actions are expressed in terms of events and screen locations. Depending on the resolution of the pointing device, the number of screen locations generated could prove astronomical. Capturing data in this form would require large amounts of storage. Apart from this, the data would be extremely difficult, if not impossible, to read. What is needed is a compact form of representing such user actions.

Siochi & Hartson, [1989], have developed such a notation. Although their User Action Notation, or UAN, was created for interface developers who needed a concise way to represent direct manipulation interfaces, user action data can be represented and stored as UAN sequences. To illustrate this point, the UAN will be presented briefly along with a short example of how it can be used to record user actions.

Consider the task of moving a file in the Macintosh™ Finder. This involves the following user actions:

1. Move the cursor to the file icon.
2. Depress and hold down the mouse button. With the button held down, move the mouse.
3. Release the mouse button.

The same information is conveyed by the following UAN sequence, written to show a linewise correspondence with the numbered lines above:

```
~[file_icon]
Mv~[x,y]
M^
```

Macintosh is a trademark of McIntosh Laboratories, Inc.

In this example, the ~ represents cursor movement, and the text inside the square brackets denotes the context of a user action. Thus in the first UAN line above, the cursor is moved to the context of file_icon. The M denotes a device, in this case the mouse, and the v denotes the mouse operation "depress the mouse button." Similarly the ^ denotes "release the mouse button." The [x,y] specifies the context of an arbitrary screen location. This example shows how dragging an object can be represented. Picking an object with a single mouse click would be represented as ~[object]Mv^.

It is the concept of context that will be exploited in using the UAN to record action sequences. Consider the events that are generated when a user clicks the mouse button on a file icon. A mouse down and up event are reported along with the screen locations at which they occur. Instead of recording the screen coordinates, the file icon (which was the context of the action) is recorded:

```
[file_icon]Mv^
```

The UAN sequence which describes how to move a file icon could also serve as the transcript for such an action by substituting x and y with the actual screen location when the mouse up event occurs:

```
~[file_icon]
Mv~[128,256]
M^
```

These are really simplistic cases, but they illustrate the direction in which further study may be undertaken.

Apart from the problem of the form in which data are recorded, there are issues that need to be studied, such as the grain of analysis. In these examples, the analysis is performed at the user action level and would be suitable for studying problems users have with composing

actions to accomplish tasks. The grain of analysis issue is aptly illustrated by Ehrich's question [Ehrich, 1989] about how an evaluator could determine from transcripts that a Macintosh user was having problems deciding which menu item to pick. It would not be possible if the transcripts were recorded at the task level, because only the menu items actually picked would appear on the transcript. However, if the transcripts were recorded at the action level, i.e., the transcripts show the user moving from choice to choice in the menu, then it would be possible to answer the question.

Chapter VIII
FUTURE WORK

In any scientific endeavor, the search for answers generates a cloud of new questions. This chapter presents many issues raised during the course of this research which remain outstanding.

8.1. EXTENSIONS AND INTERFACE IMPROVEMENTS

8.1.1. Time Stamping

The mrps might be able to provide better information if combined with time-stamping data. This would enable a quantitative expression of user effort represented by each mrp, and could provide the evaluator with another means of selecting which mrps to focus attention on. Those mrps with elapsed times greater than expected would be prime candidates for further investigation. The time information can also be used in a cost-benefits analysis to determine which problems in the interface to fix. This is an extremely important benefit in any engineering process, for as Whiteside, Bennett, & Holtzblatt [1987] point out, engineering a piece of software involves the allocation of scarce resources.

8.1.2. Full Programming Language

The current implementation of the mrp tool allows for only two mrp lists: the main list which is generated by the detection algorithm and never changes, and the filter list, which changes as a result of each selection operation performed. The ability to create and maintain an arbitrary number of lists, and specify operations on them, would be a powerful investigative tool. The extent of this capability is still an issue, therefore this proposed work should be delayed until more is known about the usability needs of an evaluator using the mrp method.

8.1.3. Similarity Indicator for Command Lines

The current method involves exact comparisons of commands. It was noted that more information, in the form of the complete command lines, was necessary for analysis. A means of representing those portions of the command lines which differed across instances of an mrp might prove helpful. For example, if the mrp

```
EXSIF
EZPLOT
```

had the following instances,

```
at position 1:
EXSIF foo.dat > foo.out
EZPLOT foo.out

at position 234:
EXSIF fie.dat > fie.out
EZPLOT fie.out
```

a similarity representation might show

```
EXSIF f??.dat > f??.out
EZPLOT f??.out
```

where the question marks indicate differences in the mrp instances. Such a representation might reduce the need to refer to the complete command lines, or to the raw files. This representation technique could be implemented using a string to string correction algorithm, similar to that used by the *diff* program in UNIX.

8.1.4 Mrps with Variables

This dissertation studied repeating patterns using exact matches, i.e., two instances of the same mrp had identical command sequences. Ehrich [1989] has suggested that sequences with inexact matches may also be interesting to investigate. Consider the case where a user

is having difficulty with a task which requires a sequence of commands. The user may repeat a sequence of commands, each time varying one element of that sequence:

```
(first attempt)
CMD_X
CMD_Y
CMD_Z

(second attempt)
CMD_X
CMD_A
CMD_Z
```

These sequences are really an mrp with a variable element. The problem would be how to detect such mrps.

8.2. SUPPORT FOR VIDEOTAPE ANALYSIS

Another interesting avenue to explore would be the use of mrp detection as an aid to videotape analysis. A major problem with videotape is having to review the tape manually for critical incidents. Since the mrp method points to potential problem sections in the transcript, it could also be used to indicate similar sections of videotape. This is readily accomplished by inserting the frame numbers generated by the videotape machine into the transcript file. When mrps are detected, the evaluator will not only be able to refer to the pertinent sections of the raw transcript, but will be able to randomly access the videotape by using the recorded frame number. Unfortunately, videotape is inherently sequential, while the investigative task requires random access. Consequently new technologies such as CD/ROM should be investigated.

8.3 BEHAVIOR CHARACTERIZATION

It may be possible to use mrps to characterize user behavior as represented in a transcript. For example, differences between expert users and novices could be studied by analyzing

the number, distribution, and types of mrps detected in their respective transcripts. One would assume that novices are engaged in what is really problem-solving behavior, whereas experts are engaged in practiced, almost mechanical behavior. Thus it would be reasonable to expect that experts would have more mrps than novices.

8.3. OTHER APPLICATIONS OF MRPS

Repeating patterns are useful in other disciplines. The algorithm described here can be used whenever the data from which patterns are to be detected can be represented as a string of characters. For example, files can be represented as a string of bits or bytes. Data compression algorithms work by eliminating the information theoretic redundancy inherent in the files. By using the mrp algorithm to detect repeating bit or byte patterns in an arbitrary file, and then assigning codes to those patterns, one could substitute the occurrences of the repeating patterns with the shorter codes, thus compressing the file.

Some form of writing analysis may also be possible using the mrp algorithm. Text files could be analyzed for repeating patterns, thus providing authors with an indication of how often they repeat words or phrases. It would also be interesting to investigate whether or not mrps can be used to characterize writing. For example, what types of mrps are found in essays which are considered as good writing by literary experts? Does Shakespeare have a characteristic mrp "signature" which can be used to verify (or disprove) authorship? What mrps characterize well-written computer programs and badly written ones?

Another interesting domain is that of cryptanalysis. The ability to detect, by means of an algorithm, repeating patterns in a ciphertext may be useful to code breakers. Denning, [1982] describes the Kasiski method which "analyzes repetitions in the ciphertext to determine the exact period" as part of a larger code breaking method.

Chapter IX
SUMMARY

This research reported on the development of a new technique for evaluating interfaces by analyzing user session transcripts. The technique involved the detection of repeated user actions in those transcripts, and is based on the hypothesis that repetition of user actions is an indicator of potential interface problems. The concept of maximal recurring patterns, or mrps, was developed as a means of defining the repetition.

An $O(n^2)$ algorithm was developed to detect mrps and was implemented as a program running on the UNIX operating system. This mrp tool enabled the evaluator to detect mrps, manage lists of mrps, select mrps based on their length or frequency, and view the raw transcripts pointed to by those mrps.

The technique was tested on GIPSY, an image processing system in use at several sites throughout the country. The data were collected from actual users over three months and the analysis involved both an independent study of the mrp lists and structured interviews of two users. The technique was shown to provide useful information about the GIPSY interface by revealing twelve specific GIPSY problems.

The advantages of the technique are its specificity and speed. Large amounts of data can be scanned to provide pointers to transcript sections which potentially show specific problems users were having. The technique's limitation is that the information it provides is at a detailed level, and does not directly indicate general problems. The current implementation also produces a lot of mrps, some of which are redundant or seem insignificant. Despite these limitations, the success of this technique in providing useful information about the GIPSY interface shows that the technique merits further research. In this regard, several specific recommendations were made.

REFERENCES

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Series in Computer Science and Information Processing. Harrison, Ed. Addison-Wesley, Reading, Massachusetts.

Aho, A., Weinberger, P., & Kernighan, B. 1978. AWK — a Pattern Scanning and Processing Language. *Soft. Prac. and Experience*. (Jul.).

Brooks, F. P., Jr. 1988. Grasping Reality Through Illusion - Interactive Graphics Serving Science. In Proceedings of *CHI'88 Conference on Human Factors in Computing Systems* (Washington, D. C., May 15-19). ACM, New York, pp. 1-11.

Card, S. K., Moran, T. P., & Newell, A. 1980. The Keystroke-Level Model for User Performance Time with Interactive Systems. *Commun. ACM. 23*, 7, pp. 396-410.

Card, S. K., Moran, T. P., & Newell, A. 1983. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum, Assoc., New Jersey.

Carroll, J. M. 1982. Learning, Using and Designing Command Paradigms. *Hum. Learn. 1*, pp. 31-62.

Carroll, J. M. & Rosson, M. B. 1985. "Usability Specification as a Tool in Iterative Development." *Advances in Human Computer Interaction*. Hartson, Ed. Ablex, Norwood, New Jersey, pp. 1-28.

Cohill, A. M. & Ehrich, R. W. 1983. Automated Tools for the Study of Human/Computer Interaction. In Proceedings of *Human Factors Society 27th Annual Meeting* (Norfolk, Va., Oct. 10-14). Human Factors Society, California, pp. 897-900.

Denning, D. E. R. 1982. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts.

Ehrich, R. W. 1982. *The DMS Multiprocess Execution Environment*. Technical Report CSIE-82-6, Virginia Tech Computer Science Dept.

Ehrich, R. W. 1989. Personal communication.

Garland, E. & Ehrich, R. W. 1987. *A GIPSY Primer*. Spatial Data Analysis Laboratory, Virginia Tech, Blacksburg, VA.

Good, M. 1985. The Use of Logging Data in the Design of a New Text Editor. In Proceedings of *CHI'85, Conference on Human Factors in Computing Systems* (San Francisco, California, April 14-18). ACM, New York, pp. 93-97.

Good, M. 1988. Comment made at a special interest group session on Field Techniques in the Design of a Usable Product, at CHI'88.

Gould, J. D. & Lewis, C. 1985. Designing for Usability: Key Principles and What Designers Think. *Commun. ACM. 28*, pp. 300-311.

Hanson, S. J., Kraut, R. E., & Farber, J. M. 1984. Interface Design and Multivariate Analysis of UNIX Command Use. *ACM Trans. Office Info. Sys. 2*, 1, pp. 42-57.

Hartson, H. R. & Hix, D. 1989. Toward Empirically Derived Methodologies and Tools for Human-Computer Interface Development. *IJMMS.* , (to appear).

Kieras, D. & Polson, P. G. 1985. An Approach to the Formal Analysis of User Complexity. *IJMMS. 22*, pp. 365-394.

Mackay, W. E., Guindon, R., Mantei, M. M., Suchman, L., & Tatar, D. G. 1988. Video: Data for Studying Human-Computer Interaction. In Proceedings of *CHI'88 Conference on Human Factors in Computing Systems* (Washington, D. C., May 15-19). ACM, New York, pp. 133-137.

Neal, A. S. & Simons, R. M. 1983. Playback: A Method for Evaluating the Usability of Software and its Documentation. In Proceedings of *CHI'83 Conference on Human Factors in Computing Systems* (Boston, Mass., Dec. 12-15). North-Holland, Amsterdam, pp. 78-82.

Olsen, D. R. & Halversen, B. W. 1988. Interface Usage Measurements in a User Interface Management System. In Proceedings of *ACM SIGGRAPH Symposium on User Interface Software* (Banff, Alberta, Canada, Oct. 17-19). ACM Press, pp. 102-108.

Payne, S. J. & Green, T. R. G. 1986. "Task-Action Grammars: A Model of the Mental Representation of Task Languages." *Human-Computer Interaction.* Lawrence Erlbaum Associates, Inc., pp. 93-133.

Reisner, P. 1981. Formal Grammar and Human Factors Design of an Interactive Graphics System. *IEEE Trans. Soft. Eng. SE-7*, pp. 229-240.

SAS Institute. 1979. *SAS User's Guide.* Helwig & Council, Ed. SAS Institute, Raleigh, N.C.

Shneiderman, B. 1983. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Comput. 16*, 8, pp. 57-69.

Siochi, A. C. & Hartson, H. R. 1989. Task-oriented Representation of Asynchronous User Interfaces. In Proceedings of *CHI'89 Conference on Human Factors in Computing Systems* (Austin, Texas, April 30 - May 4). ACM, New York, (to appear).

Snyder, H. 1987. Comment made in a class on non-parametric statistics.

Weiner, P. 1973. Linear Pattern Matching Algorithms. In Proceedings of *IEEE 14th Annual Symposium on Switching and Automata Theory* , pp. 1-11.

Whiteside, J., Bennett, J., & Holtzblatt, K. 1987. *Usability Engineering: Our Experience and Evolution*. Technical Report DEC-TR 547 (to appear as a chapter in Handbook of Human-Computer Interaction, M. Helander ed., North-Holland), Digital.

Williges, R. C. 1987. The Use of Models in Human-Computer Interface Design. *Ergonomics*. *30*, 3, pp. 491-502.

Williges, R. C., Williges, B. H., & Elkerton, J. 1987. "Software Interface Design." *Handbook of Human Factors*. Salvendy, Ed. Wiley, New York, pp. 1416-1449.

APPENDIX

The mrp analysis tool was written in C, on a VAX 11/785 running ULTRIX 2.2. The source code is stored in a UNIX tar file, and was successfully transferred to, and compiled on a Macintosh II running A/UX, Apple Computer's implementation of UNIX. The normalizer routines were written in AWK and C-shell, on the VAX. They have not been tested on A/UX.

The rest of this appendix contains the MAKE files used to maintain the source code of the mrp analysis tool, and brief instructions for using the mrp analysis tool.

MAKEFILES

MAKE is a UNIX utility used to maintain the link dependencies of source files. Changes to source files are tracked. When the command "make" is issued, only those source files which changed are recompiled, and the executable image is rebuilt by linking the new object files. For more information about the MAKE utility, see any UNIX manual.

The source code is kept in two subdirectories, newpt and symtabs. Newpt contains the code for the position tree and mrp routines. Symtabs contains the code for the symbol table routines. Each of these subdirectories contains a makefile that describes the link dependencies among the source files. The top level directory contains the makefile that describes how to link the various object files. There is no need to work with the lower level makefiles. The top level makefile allows a developer to bring the executable image up to date, or create a tar file.

Top level directory:

```
Makefile    README      STATUS
newpt/      symtabs/    ui*
```

Makefile:

```
P          = lpr -p
MAIN       = ui

OBJECTS  =  newpt/lib/ui.o \
            newpt/lib/pt.o \
            newpt/lib/dl.o \
            newpt/lib/newMRP.o \
            newpt/lib/lines.o \
            newpt/lib/mem.o \
            newpt/lib/myAtoi.o \
            newpt/lib/myStr.o \
            newpt/lib/timer.o \
            newpt/lib/list.o \
            symtabs/lib/btree.o \
            symtabs/lib/val.o \
            symtabs/lib/getST.o

$(MAIN)    : Onewpt  Osymtabs;
           cc -g $(OBJECTS) -o $(MAIN)

Osymtabs:
           (cd symtabs; make all )

Onewpt:
           (cd newpt; make all )

index:     $(CFILES)
           ctags -x $(CFILES) > index

print:     Makefile Mnewpt Msymtabs
#          $P $?
           echo `date` >> print
           echo $? >> print
           touch print

Mnewpt:
           (cd newpt; make print)

Msymtabs:
           (cd symtabs; make print )

clean      : ;
           (cd newpt; make clean)
           (cd symtabs; make clean)
           /bin/rm  -f  $(MAIN)  core

install    : ;
           (cd newpt; mkdir lib)
```

```
            (cd symtabs; mkdir lib)
            make

tar         : ;
            make clean
            tar cvf ../0MRPS.tar .

test        : ;
            make $(MAIN)
            $(MAIN) dumper
```

Subdirectory newpt:

```
Makefile    lib/    src/
all         test/
```

Makefile:

```
#CFLAGS    = -c -O
CFLAGS     = -c -g

CFILES     = src/ui.c src/pt.c src/dl.c src/newMRP.c src/lines.c src/myAtoi.c
src/myStr.c src/mem.c src/list.c src/timer.c
IFILES     = src/pt.h src/dl.h  src/misc.h
XFILES     = src/ui.c.xfc src/pt.c.xfc src/dl.c.xfc src/newMRP.c.xfc src/list.c.xfc

lib/ui.o    : src/ui.c \
                    src/misc.h \
                    src/pt.h \
                    src/ui.c.xfc \
                    src/dl.h
            cc $(CFLAGS) src/ui.c
            mv ui.o lib/ui.o

lib/pt.o    : src/pt.c \
                    src/misc.h \
                    src/pt.c.xfc \
                    src/pt.h
            cc -DTRACKMEM $(CFLAGS) src/pt.c
#           cc $(CFLAGS) src/pt.c
            mv pt.o lib/pt.o

lib/dl.o    : src/dl.c \
                    src/dl.h \
                    src/dl.c.xfc \
                    src/misc.h
            cc $(CFLAGS) src/dl.c
            mv dl.o lib/dl.o
```

```
lib/newMRP.o : src/newMRP.c \
                        src/misc.h \
                        src/dl.h \
                        src/newMRP.c.xfc \
                        src/pt.h
            cc $(CFLAGS) src/newMRP.c
            mv newMRP.o lib/newMRP.o

lib/lines.o  : src/lines.c
            cc $(CFLAGS) src/lines.c
            mv lines.o lib/lines.o

lib/myAtoi.o  : src/myAtoi.c
            cc $(CFLAGS) src/myAtoi.c
            mv  myAtoi.o lib/myAtoi.o

lib/myStr.o   : src/myStr.c
            cc $(CFLAGS) src/myStr.c
            mv  myStr.o lib/myStr.o

lib/mem.o  : src/mem.c
            cc $(CFLAGS) -DTRACKMEM src/mem.c
#           cc $(CFLAGS) -DNOFREE src/mem.c
#           cc $(CFLAGS) src/mem.c
            mv mem.o lib/mem.o

lib/timer.o     : src/timer.c
            cc $(CFLAGS) src/timer.c
            mv  timer.o lib/timer.o

lib/list.o   : src/list.c \
                        src/list.c.xfc
            cc $(CFLAGS) src/list.c
            mv  list.o lib/list.o

all: lib/ui.o lib/pt.o lib/dl.o lib/newMRP.o lib/lines.o lib/myAtoi.o lib/myStr.o lib/mem.o
lib/timer.o  lib/list.o
            touch all

clean       :  ;
            (cd lib; /bin/rm -f *)

print       : Makefile $(XFILES) $(IFILES) $(CFILES)
#           print $?
            echo `date` >> print
            echo $? >> print
            touch print
```

Subdirectory symtabs:

```
EXPLAIN    lib/    src/
Makefile   include/        test/
```

Makefile:

```
test   = $(SYMTABS)/test
#CFLAGS = -c -O
CFLAGS = -c -g

CFILES = src/btree.c src/getST.c src/val.c
IFILES = include/btree.h

test/btmain :  test/btmain.o  lib/btree.o  ../newpt/lib/mem.o  lib/val.o
               cc  -O  test/btmain.o  lib/btree.o  lib/val.o  ../newpt/lib/mem.o  -o  test/btmain

test/tst    :  test/tst.o  \
               lib/getST.o  \
               lib/btree.o  \
               ../newpt/lib/mem.o   \
               lib/val.o
               cc  -O  test/tst.o  \
                 lib/getST.o  \
                 lib/btree.o  \
                  ../newpt/lib/mem.o  \
                 lib/val.o  -o  test/tst

test/tst.o   :  test/tst.c
               cc  $(CFLAGS)  test/tst.c
               mv  tst.o  test/tst.o

test/btmain.o  :     test/btmain.c  \
               $(SYMTABS)/include/btree.h
               cc  $(CFLAGS)  test/btmain.c
               mv  btmain.o  test/btmain.o

lib/btree.o  :  src/btree.c  \
               include/btree.h
               cc  $(CFLAGS)  src/btree.c
               mv  btree.o  lib/btree.o


lib/getST.o  :  src/getST.c
               cc  $(CFLAGS)  src/getST.c
               mv  getST.o  lib/getST.o

lib/val.o    :  src/val.c  \
               include/btree.h
```

```
                cc $(CFLAGS) src/val.c
                mv val.o lib/val.o

all           : lib/btree.o lib/getST.o lib/val.o
              touch all

print         : Makefile $(IFILES) $(CFILES)
#             print $?
              echo `date` >> print
              echo $? >> print
              touch print

test          : test/btmain  test/tst;
                echo "btmain; cat tst.data | tst"

clean         : ;
                /bin/rm -f  lib/btree.o lib/getST.o \
                lib/val.o
                make cleantest

cleantest     : ;
                /bin/rm -f test/btmain.o test/btmain test/tst test/tst.o
```

MRP TOOL INSTRUCTIONS

Before using the Mrp tool, a transcript file must first be normalized. The normalized file should have a filename extension of ".cmd," the file containing the complete command lines should have an extension of ".inp," and the raw transcript an extension of ".raw." Once the mrp tool (named "ui" by the makefile routines) has been compiled, it should be installed such that it can be run from any directory. On UNIX systems, this is accomplished by installing it in a directory which is in the command search path.

To run the mrp tool, type "ui." The tool has a simple command line interface, and the commands are listed in Table 1. Help is available by typing the command "help." To extract the mrps of a normalized file named "user1.cmd," first make the position tree, then extract the mrps. Do this by entering the command "mkpt user1," followed by the command "mkmrp." Do not enter the complete filename "user1.cmd" as the tool expects only the name portion and not the extension of the filename. Note that all command lines are

terminated by a carriage return, all commands and keywords must be entered completely (no abbreviations), and each argument on the command line must be separated by at least one space. Typing a blank command line will repeat the previous command.

Some dots will appear on the terminal screen indicating the progress of the extraction. When the ">" prompt appears, the extraction is done. What has happened is that a doubly linked list of mrps has been generated. The commands "show," "show next," and "show prev" are used to examine this list. However, at this point it is helpful to issue the "info" command to obtain information about the number and distribution of the mrps detected. If there are more mrps in the list than there is time to analyze, it is possible to generate another list of mrps containing only selected mrps. This is done using the "filter" command. It is possible to switch from the filter list to the main list by using the "go" command. Thus the command line "go main" makes the main list the current list, and issuing the "show" commands will cause this list to be traversed. Each subsequent invocation of the "filter" command works on the current list, and replaces the previous filter list. Thus if the current list is the filter list, successive filters can be applied to effectively produce boolean "AND" queries. The main list is never replaced.

All the information that appears on the screen can be saved or appended to a file. Thus if a particularly interesting mrp is found, the mrp, the complete command lines, or the sections of raw transcript can be saved to a file.

A simple macro capability is also available. A file containing mrp tool commands can be executed by issuing the command "readfrom filen," where filen is the name of the commands file. To re-establish input from the terminal, be sure that filen ends with the command "readfrom."

Finally, local operating system commands are accessible from within the mrp tool. Any text which follows a "! " is sent to the operating system for exection. This is analogous to the shell escape mechanisms provided in many UNIX programs, e.g., *vi*.

The vita has been removed from the scanned document