

CHAPTER 1: Introduction

1.1 Improving the Design-Use Fit

Involving end users in the development of interactive systems increases the likelihood those systems will be useful and usable [Nielsen 1993; Baecker et al. 1995]. However, user involvement is both time and resource intensive. In some ways, the Internet has magnified these problems by increasing the number, variety, and distribution of potential users and usage situations while reducing the typical cycle time between product releases, and thus the time available for involving users. On the other hand, the Internet presents hitherto unprecedented, and currently underutilized, opportunities for increasing user involvement by enabling cheap, rapid, and large-scale distribution of software for evaluation purposes and providing convenient mechanisms for communicating information about application use and user feedback to development organizations interested in capturing such information.

Unfortunately, one of the main problems facing development organizations today is that there are already more suspected problems, proposed solutions, and novel ideas (emanating from marketers, planners, designers, developers, usability professionals, support personnel, users, and other stakeholders) than there are resources available for addressing such issues (including design, implementation, testing, and usability resources) [Cusumano and Selby 1995]. As a result, development organizations are often more concerned with addressing the following two problems, than in generating *more* ideas about how to improve application designs:

- *Impact assessment*: What is the actual impact of suspected or observed problems on users at-large? What is the expected impact of implementing proposed solutions or novel ideas on users at-large?
- *Effort allocation*: Where should scarce design, implementation, testing, and usability evaluation resources be focused in order to produce the greatest benefit for users at-large?

The two techniques most commonly used to evaluate the fit between application design and use — namely, usability testing and beta testing with user feedback — fail to address these questions adequately, and suffer from a number of limitations that restrict evaluation scale, in the case of usability tests, and data quality, in the case of beta tests.

The goal of this dissertation is to demonstrate technical and methodological solutions to enable usage- and usability-related information of much higher quality than currently available from beta tests to be collected on a much larger scale than currently possible in usability tests. Such data is complementary in that it can be used to address the impact assessment and effort allocation problems in addition to evaluating and improving the fit between application design and use.

The approach described herein involves a development platform for creating software agents that are deployed over the Internet to observe application use and report usage data and user feedback to developers to help improve the fit between design and use. The data can be used to illuminate how applications are used, to uncover mismatches in actual versus expected use, and to increase user involvement in the evolution of interactive systems. This research is aimed at helping developers make more informed

design, impact assessment, and effort allocation decisions, ultimately leading to more cost-effective development of software that is better suited to user needs.

1.2 Limitations of Current Techniques

1.2.1 Usability Testing

Scale emerges as the critical limiting factor in usability tests. Usability tests are typically restricted in terms of size, scope, location, and duration:

- Size is an issue because limitations in data collection and analysis techniques result in the effort of performing evaluations being directly linked to the number of subjects being evaluated.
- Scope is an issue because typically only a small fraction of an application's overall functionality can be exercised during any given evaluation.
- Location is an issue because users must typically be displaced from their normal work environments and placed under more controlled laboratory conditions.
- Finally, duration is an issue because users typically cannot devote extended periods of time to evaluation activities which take them away from their other day-to-day responsibilities.

Perhaps more significantly, however, once problems have been identified in the usability lab, the impact assessment problem remains: What is the actual impact of identified problems on users at-large? Answering this question is essential in tackling the effort allocation problem with respect to resources required to fix identified problems. Furthermore, because usability testing is itself so expensive in terms of user and evaluator effort and time, the effort allocation problem arises in this regard as well: Where should

scarce usability evaluation resources be focused in order to produce the greatest benefit for users at-large?

1.2.2 Beta Testing

Data quality issues emerge as the critical limiting factor in beta tests. When beta testers report usability issues in addition to software bugs, data quality is typically limited due lack of proper incentives, a paradoxical relationship between user performance and subjective reports, lack of necessary knowledge, and lack of detail in reported data.

Incentives are a problem since users are typically more concerned with getting their work done than in paying the price of problem reporting while developers receive most of the benefit. As a result, often only the most obvious or unrecoverable errors are reported.

Perhaps more significantly, there is often a paradoxical relationship between users' performance with respect to a particular application and their subjective ratings of its usability. Numerous usability professionals have observed this phenomenon. Users who perform well in usability evaluations will often volunteer comments in which they report problems with the interface even though these problems apparently did not affect the user's ability to complete tasks. When asked for a justification, these users will typically say something to the effect: "Well, it was easy for me, but I think other people would have been confused." Sometimes these users correctly anticipate problems encountered by other, less seasoned, users, however, this type of feedback is based on

speculation and frequently turns out to be unfounded. On the other hand, users who encounter great difficulties using a particular interface will often *not* volunteer comments, and if pressed, report that the interface is well designed and easy to use. When confronted with the discrepancy between their subjective reports and observed behavior, these users will typically say something to the effect: “Someone with more experience would probably have had a much easier time,” or “I always have more trouble than average with this sort of thing.” As a result, potentially important feedback from beta testers having difficulties will fail to be reported while potentially misleading or unfounded feedback from beta testers having no difficulties will be reported.¹

Nevertheless, beta tests do appear to offer good opportunities for collecting usability-related information. Smilowitz and colleagues showed that beta testers who were asked to record usability problems as they arose in normal use identified almost the same number of significant usability problems as identified in more traditional laboratory tests of the same software [Smilowitz et al. 1994]. A later case study performed by Hartson and associates, using a remote data collection technique, also appeared to support these results [Hartson et al. 1996]. However, while the number of usability problems identified in the lab test and beta test conditions was roughly equal, the number of common problems identified by both was rather small. In summarizing their results, Smilowitz and colleagues offered the following as one possible explanation:

Another reason for this finding may have to do with the individual identifying the problems. In the lab test two observers with experience with

1. These examples were taken from a thread that appeared in a “private” discussion group for usability and user interface design professionals.

the software identified and recorded the problems. In some cases, the users were not aware they were incorrectly using the tool or understanding how the tool worked. If the same is true of the beta testers, some severe problems may have been missed because the testers were not aware they were encountering a problem, and therefore did not record the problem.

Since users are not always aware of developers' expectations about appropriate use, they can behave in ways that blatantly violate developers' expectations without being aware of it. As a result, mismatches in expected versus actual use may go undetected for extended periods, resulting in ongoing usability issues.

Another important limitation identified by Smilowitz and colleagues is that the feedback reported in the beta test condition lacked details regarding the interactions leading up to problems and the frequency of problem occurrences.

In summary, evaluation data reported by beta testers is problematic due to lack of proper incentives, a paradoxical relationship between user performance and subjective reports, lack of necessary knowledge, and lack of details in reported data. Furthermore, without more information regarding the frequency of problem occurrences (or the frequency with which features associated with reported problems are used) the impact assessment and effort allocation problems continue to remain unresolved.

1.2.3 Early Attempts to Exploit the Internet

A number of investigators have begun to investigate Internet-mediated techniques for overcoming some of the limitations inherent in current usability and beta testing methods.

Some have investigated the use of Internet-based video conferencing and remote “application sharing” technologies to support remote usability evaluations [Castillo and Hartson 1997]. Unfortunately, while leveraging the Internet to overcome geographical barriers, these techniques fail to exploit the enormous potential afforded by the Internet to lift current restrictions on evaluation size, scope, and duration. This is because a large amount of data is generated per user, and because observers are typically required to observe and interact with users on a one-on-one basis.

Others have investigated the use of Internet-based user-reporting of “critical incidents” to capture user feedback and limited usage information [Hartson et al. 1996]. In this approach, users are trained to identify “critical incidents” themselves and to press a “report” button to send video data regarding events immediately preceding and following user-identified incidents back to experimenters. While addressing, to a limited degree, the problem of lack of detail in beta tester-reported data, this approach unfortunately suffers from all of the other problems associated with beta tester-reported feedback, including lack of proper incentives, the subjective feedback paradox, and lack of necessary knowledge.

An alternative to such techniques involves automatically capturing information about user and application behavior by monitoring the software components that make up the application and its user interface. This data can then be automatically transported to evaluators who may then analyze it to identify potential problems. A number of application instrumentation and event monitoring techniques have been proposed for this

purpose. However, existing approaches all suffer from some combination of the following problems, resulting in significant limitations on evaluation scalability and data quality:

- *The abstraction problem:* Questions about usage typically occur in terms of concepts at higher levels of abstraction than represented in the data available from software components. This implies the need for “data abstraction” mechanisms to allow low-level data to be related to higher-level concepts such as user interface and application features as well as users' tasks and goals.
- *The selection problem:* The amount of data necessary to answer usage questions is typically a small subset of the much larger set of data that might be captured at any given time. This implies the need for “data selection” mechanisms to allow necessary data to be captured, and unnecessary data filtered, prior to reporting and analysis.
- *The reduction problem:* Much of the analysis that will ultimately be performed to answer usage questions can actually be performed during data collection resulting in greatly reduced data reporting and post-hoc analysis needs. This implies the need for “data reduction” mechanisms to reduce the amount of data that must ultimately be reported and analyzed.
- *The context problem:* Potentially critical information necessary in interpreting the significance of events is often not readily available in event data alone. This implies the need for “context-capture” mechanisms to allow important user interface, application, artifact, and user state information to be used in data abstraction, selection, and reduction.
- *The evolution problem:* Finally, data collection needs typically evolve over time (perhaps due to results of earlier data collection) more rapidly than do applications. This implies the need for “independently evolvable” data collection mechanisms to allow in-context data abstraction, selection, and reduction to evolve over time without impacting application deployment or use.

Furthermore, there are currently no theoretical or methodological guidelines to provide guidance regarding how to collect, analyze, and interpret data and how to incorporate these activities in the development process.

1.3 Research Overview

1.3.1 Goals

As mentioned above, the main goal of this research is to demonstrate technical and methodological solutions to enable larger-scale collection, than currently possible in usability tests, of higher quality data, than currently possible in beta tests, to address the impact assessment and effort allocation problems in addition to capturing information to help improve the fit between application design and use. The underlying motivation is to help developers make more informed design, impact assessment, and effort allocation decisions, ultimately leading to more cost-effective development of software that is better suited to user needs.

1.3.2 Hypotheses

The work presented here is based on the following basic principles:

- That developers have expectations about application use that affect application design, and that designs often embody usage expectations even when developers are not explicitly aware of them.
- That mismatches between expected and actual use indicate potential problems in design or use that may negatively impact usability and utility.
- That making expectations more explicit and observing use to compare users' actions against expectations can help in identifying and resolving mismatches.
- That such mismatch identification and resolution can help bring expectations, and thus designs, into better alignment with actual use.

Based on these principles, we hypothesize that software monitoring techniques can be exploited to aid in the process of mismatch identification and resolution, and thus support developers in improving the design-use fit.

1.3.3 Approach

The approach described in this dissertation involves a development platform for creating software agents that are deployed over the Internet to observe application use and report usage data and user feedback to developers to help improve the fit between application design and use. To this end, the following process is employed:

- Developers design applications and identify usage expectations.
- Developers create agents to monitor application use and capture usage data and user feedback.
- Agents are deployed over the Internet independently of the application to run on users' computers each time the application is run.
- Agents perform in-context data abstraction, selection, and reduction as needed to allow actual use to be compared against expected use on a much larger scale than possible before.
- Agents report data back to developers to inform further evolution of expectations, the application, and agents.

It is also hoped that the use of software monitoring techniques will help make incorporating information about users more palatable to developers. If developer involvement in data collection is increased, the likelihood that results will be taken seriously may also be increased.

1.3.4 Evaluation

This research has been subjected to a number of evaluative activities.

First, a software development project at NYNEX Corporation was observed in which developers observed users completing tasks with prototypes in order to identify mismatches between expected and actual use [Girgensohn et al. 1994]. This experience served to confirm the hypothesis that such an iterative design process could in fact lead to tangible design benefits, and highlighted areas in which automated support might be used to improve the process. An early prototype was constructed as a proof-of-concept that software agents could be used to automatically monitor expectations on developers' behalf and support user-developer communication without requiring developers to be physically present to observe use. This is described in further detail in Chapter 8.

Based on this experience, and with an eye on making the approach more scalable, a second prototype was constructed to explore and clarify the technical issues involved in collecting usage data and user feedback on a potentially large and ongoing basis [Hilbert and Redmiles 1998a & 1998b]. This prototype was informally evaluated within the context of a demonstration scenario performed by Lockheed Corporation, in which independent developers applied the approach to a database query interface in a large-scale governmental logistics and transportation information system. Informal evidence from this experience suggested that the approach was not only suited to capturing design-related information, but that important impact assessment and effort allocation-related information could also be captured. This experience is described in further detail in Chapter 8.

Next, a survey of existing techniques for extracting usability information from user interface events was conducted to identify critical areas in need of improvement. This survey uncovered the abstraction, selection, reduction, context, and evolution problems as key factors limiting the scalability and data quality of existing approaches. Survey results are discussed further in Chapter 4.

Next, participant-observation of a large-scale industrial usage data collection project was undertaken at Microsoft Corporation in order to better understand the issues faced by development organizations wishing to capture usage information on a large-scale. Again the experience served to reinforce the hypothesis that automated software monitoring techniques can be used to capture useful design, impact assessment, and effort allocation information, and also served to highlight a number of ways in which existing techniques could be improved based on the concepts developed in this research. This experience is described further in Chapter 8.

Finally, a third prototype was constructed based on the experience and insights gained from the first two prototypes, the survey of existing techniques, and the Microsoft experience. This prototype served as the basis for demonstrating solutions to the abstraction, selection, reduction, context, and evolution problems within a single data collection architecture. The results of this experience are presented in Chapters 5 and 6.

1.3.5 Contributions

The main contributions of this work include:

- Identification of five key problems limiting the scalability and data quality of existing techniques for extracting usability information from user interface events (Chapter 4).
- A layered architecture that separates data collection code from generic data collection services, which in turn are separated from generic monitoring services, which in turn are separated from details of the underlying component model in which applications are developed (Chapter 5).
- Solutions to the abstraction, selection, reduction, context, and evolution problems demonstrated within a single data collection architecture (Chapters 5 and 6).
- A reference architecture to provide design guidance regarding key components and relationships required to support large-scale data collection (Chapter 6).
- A theory motivating the significance of usage expectations in development and the importance of collecting usage information (Chapter 7).
- Methodological guidance regarding data collection, analysis, interpretation, and process integration (Chapter 7).

1.4 Structure of the Dissertation

The dissertation is organized as follows.

- Chapter 2 provides background to situate this work within the broader context of existing usability evaluation techniques.
- Chapter 3 discusses the specific nature of user interface event data and some important implications on analysis.
- Chapter 4 presents a review of existing techniques for extracting usability information from user interface events, and presents five key problems limiting scalability and data quality.
- Chapter 5 presents more details regarding the approach advocated in this dissertation and a usage scenario to provide the necessary foundation for understanding the following chapter.

- Chapter 6 presents detailed solutions to the problems identified in the review of related work.
- Chapter 7 discusses important theoretical and methodological considerations.
- Chapter 8 provides more details regarding the evaluation activities outlined above.
- Chapter 9 discusses problems faced by practitioners wishing to employ the proposed approach and future directions for researchers wishing to extend this work.
- Chapter 10 presents conclusions, a summary of contributions, and discusses other potential applications of the work.

CHAPTER 2: Background

This chapter serves three main purposes. First, it establishes working definitions of key terms such as “usability,” “usability evaluation,” and “usability data”. Second, it situates observational usability evaluation within the broader context of human-computer interaction (HCI) evaluation approaches, indicating some of the relative strengths and limitations of each. Finally, it identifies user interface events as one of the many types of data commonly collected in observational usability evaluation, indicating some of its strengths and limitations relative to other types. The definitions and frameworks presented here are not new and can be found in standard HCI texts [Nielsen 1993; Preece et al. 1994]. Those well acquainted with usability and usability evaluation may wish to skip directly to Chapter 3 where the specific nature of user interface events and the reasons why analysis is complicated are presented.

2.1 Definitions

“*Usability*” is often thought of as referring to a single attribute of a system or device. However, it is more accurately characterized as referring to a large number of related attributes. Nielsen provides the following definition [Nielsen 1993]:

Usability has multiple components and is traditionally associated with these five usability attributes:

Learnability: The system should be easy to learn so that the user can rapidly start getting some work done with the system.

Efficiency: The system should be efficient to use, so that once the user has learned the system, a high level of productivity is possible.

Memorability: The system should be easy to remember, so that the casual user is able to return to the system after some period of not having used it, without having to learn everything all over again.

Errors: The system should have a low error rate, so that users make few errors during the use of the system, and so that if they do make errors they can easily recover from them. Further, catastrophic errors must not occur.

Satisfaction: The system should be pleasant to use, so that users are subjectively satisfied when using it; they like it.

“Usability evaluation”, can be defined as the act of measuring (or identifying potential issues affecting) usability attributes of a system or device with respect to particular users, performing particular tasks, in particular contexts. The reason that users, tasks, and contexts are part of the definition is that the values of usability attributes may vary depending on the background knowledge and experience of users, the tasks for which the system is used, and the context in which it is used.

“Usability data” is any information that is useful in measuring (or identifying potential issues affecting) the usability attributes of a system under evaluation.

It should be noted that the definition of usability cited above makes no mention of the particular purposes for which the system is designed or used. Thus, a system may be perfectly usable and yet not serve the purposes for which it was designed. Furthermore, a system may not serve any useful purpose at all (save for providing subjective satisfaction)

and still be regarded as perfectly usable. Herein lies the distinction between usability and utility.

Usability and utility are regarded as subcategories of the more general term “usefulness” [Grudin 1992]. Utility is the question of whether the functionality of a system can, in principle, support the needs of users, while usability is the question of how satisfactorily users can make use of that functionality. Thus, system usefulness depends on both usability and utility.

Usability evaluations often identify both usability and utility issues, thus more properly addressing usefulness than strict usability. However, to avoid introducing new terminology, the following discussion simply assumes that usability evaluations and usability data can address questions of utility as well as questions of usability.

2.2 Types of Usability Evaluation

This section presents and contrasts the different types of approaches that have been brought to bear in evaluating usability in the field of human-computer-interaction (HCI).

First, a distinction is commonly drawn between formative and summative evaluation. *Formative* evaluation primarily seeks to provide feedback to designers to inform and evaluate design decisions. *Summative* evaluation primarily involves making judgements about “completed” products, to measure improvement over previous releases

or to compare competing products. The techniques discussed in the following chapters may be applied in both sorts of cases.

Another important distinction is the more specific motivation for evaluating. There are a number of practical motivations for evaluating. For instance, one may wish to gain insight into the behavior of a system and its users in actual usage situations in order to improve usability (formative) and to validate that usability has been improved (summative). One may also wish to gain further insight into users' needs, desires, thought processes, and experiences (also formative and summative). One may wish to compare design alternatives, for example, to determine the most efficient interface layout or the best design representation for some set of domain concepts (formative). One may wish to compute usability metrics so that usability goals can be specified quantitatively and progress measured, or so that competing products can be compared (summative). Finally, one may wish to check for conformance to interface style guidelines and/or standards (summative). There are also academic motivations, such as the desire to discover features of human cognition that affect user performance and comprehension with regard to human-computer interfaces (formative).

There are a number of HCI evaluation approaches for achieving these goals which fall into three basic categories: predictive, observational, and participative.

Predictive evaluation usually involves making predictions about usability attributes based on psychological modeling techniques — e.g., the GOMS model [John and Kieras 1996a & 1996b] or Cognitive Walkthrough [Lewis et al. 1990], or based on

design reviews performed by experts equipped with a knowledge of HCI principles and guidelines and past experience in design and evaluation — e.g., Heuristic Evaluation [Nielsen and Mack 1994]. A key strength of predictive approaches is their ability to produce results based on non-functioning design artifacts without requiring the involvement of actual users.

Observational evaluation involves measuring usability attributes based on observations of users actually interacting with prototypes or fully functioning systems. Observational approaches can range from formal laboratory experiments to less formal field studies. A key strength of observational techniques is that they tend to uncover aspects of actual user behavior and performance that are difficult to capture using other techniques.

Finally, *participative evaluation* involves collecting information regarding usability attributes directly from users based on their subjective reports. Methods for collecting such data range from questionnaires and interviews to more ethnographically inspired approaches involving joint observer/participant interpretation of behavior in context. A key benefit of participative techniques is their ability to capture aspects of users' needs, desires, thought processes, and experiences that are difficult to obtain otherwise.

In practice, actual evaluations often combine techniques from multiple approaches. However, the methods for posing questions and for collecting, analyzing, and interpreting data vary from one category to the next. Table 2-1 provides a high level

summary of the relationship between types of evaluation and typical reasons for evaluating. An upper-case ‘X’ indicates a strong relationship. A lower-case ‘x’ indicates a weaker relationship. An empty cell indicates little or no relationship.

Table 2-1: Types of evaluation and reasons for evaluating

Reasons for Evaluating	Predictive Evaluation	Observational Evaluation	Participative Evaluation
Understanding user behavior and performance		X	x
Understanding user thoughts and experience		x	X
Comparing design alternatives	x	X	X
Computing usability metrics	x	X	X
Certifying conformance with guidelines and standards	X		

The approach described in this dissertation is observational, although it also provides limited support for capturing participative data as well.

2.3 Types of Usability Data

Having situated observational usability evaluation within the broader context of predictive, observational, and participative approaches, user interface events can be isolated as just one of many possible sources of observational data.

Sweeny and colleagues (Sweeny et al. 1993) identify a number of indicators that might be used to measure (or indicate potential issues affecting) usability attributes:

- *On-line behavior/performance*: e.g., task times, percentage of tasks completed, error rates, duration and frequency of on-line help usage, range of functions used.
- *Off-line behavior (non-verbal)*: e.g., eye movements, facial gestures, duration and frequency of off-line documentation usage, off-line problem solving activity.

- *Cognition/understanding*: e.g., verbal protocols, answers to comprehension questions, sorting task scores.
- *Attitude/opinion*: e.g., post-hoc comments, questionnaire and interview comments and ratings.
- *Stress/anxiety*: e.g., galvanic skin response (GSR), heart rate (ECG), event-related brain potentials (ERPs), electroencephalograms (EEG), ratings of anxiety.

Table 2-2 summarizes the relationship between these indicators and various techniques for collecting observational data. This table is by no means comprehensive, and is used only to indicate the rather specialized yet complementary nature of user interface event data in observational evaluation. UI events provide excellent data for quantitatively characterizing on-line behavior, however, the usefulness of UI events in providing data regarding the remaining indicators has not been demonstrated. Some investigators, however, have used UI events to infer features of user knowledge and understanding [Kay and Thomas 1995; Guzdial et al. 1993].

Table 2-2: Data collection techniques and usability indicators

Usability Indicators	UI Event Recording	Audio/Video Recording	Post-hoc Comments	User Interview	Survey/ Questionnaire/ Test scores	Psychophysical recording
On-line behavior/performance	X	X				
Off-line behavior (non-verbal)		X				
Cognition/understanding	X	X	X	X	X	
Attitude/opinion			X	X	X	
Stress						X

The approach described in this dissertation relies on a variant of event recording (in which state information is captured in addition to events) in conjunction with a variant of post-hoc comments (in which comments may be captured in-context, as well as post-hoc).

CHAPTER 3: Nature of UI Events

This chapter discusses the nature and characteristics of HCI events in general, and UI events in specific. The grammatical nature of UI events is discussed including some implications on analysis. The importance of contextual information in interpreting the significance of events is also discussed. Finally, a compositional model of UI events is presented to illustrate how these issues manifest themselves in UI event data analysis. This discussion is used to ground later discussion and to highlight some of the strengths and limitations of current techniques for extracting usability information from UI events.

3.1 Spectrum of HCI Events

Before discussing the specific nature of UI events, this section introduces the broader spectrum of events of interest to researchers and practitioners in HCI. Figure 3-1, adapted from [Sanderson and Fisher 1994], indicates the durations of different types of HCI events.

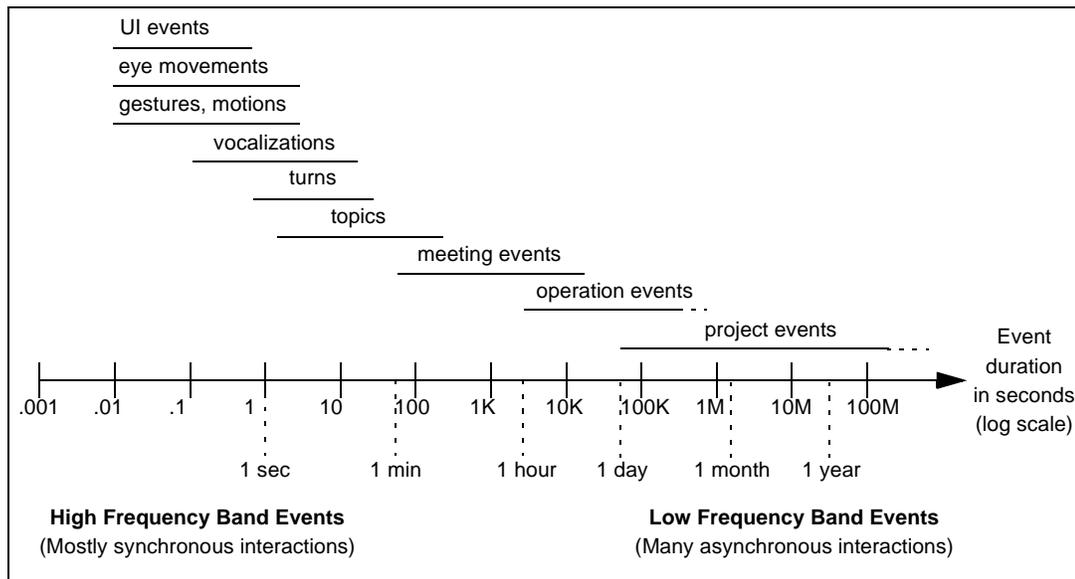


Figure 3-1. A spectrum of HCI events. Adapted from [Sanderson and Fisher 1994].

The horizontal axis is a log scale indicating event durations in seconds. It ranges from durations of less than one second to durations of years. The durations of UI events fall in the range of 10 milliseconds to approximately one second. The range of possible durations for each “type” of event is between one and two orders of magnitude, and the ranges of different types of events overlap one another.

If we assume that events occur serially, then the possible frequencies of events are constrained by the duration of those events. So, by analogy with the continuous domain (e.g. analog signals), each event type will have a characteristic frequency band associated with it [Sanderson and Fisher 1994]. Event types of shorter duration, for example, UI events, can exhibit much higher frequencies when in sequence, and thus might be referred to as high-frequency band event types. Likewise, event types of longer duration, such as

project events, exhibit much lower frequencies when in sequence and thus might be referred to as low-frequency band event types. Evaluations that attempt to address the details of interface design have tended to focus on high-frequency band event types, whereas research on computer supported cooperative work (CSCW) has tended to focus on mid- to low-frequency band event types [Sanderson and Fisher 1994].

Some important properties of HCI events that emerge from this characterization include the following:

1. *Synchronous vs. Asynchronous Events*: Sequences composed of high-frequency event types typically occur synchronously. For example, sequences of UI events, gestures, or conversational turns can usually be captured synchronously using a single recording. However, sequences composed of lower frequency event types, such as meeting or project events, may occur asynchronously, aided, for example, by electronic mail, collaborative applications, memos, and letters. This has important implications on the methods used to sample, capture, and analyze data, particularly at lower frequency bands [Sanderson and Fisher 1994].
2. *Composition of Events*: Events within a given frequency band are often composed of events from higher frequency bands. These same events, in turn, combine to form events at lower frequency bands. Sanderson and Fisher offer this example: a conversational turn is typically composed of vocalizations, gestures, and eye movements, and a sequence of conversational turns may combine to form a topic under discussion within a meeting [Sanderson and Fisher 1994]. This compositional structure is also exhibited *within* frequency bands. For instance, human-computer interactions occur and may be analyzed at multiple levels of abstraction, where events at each level are composed of events occurring at lower levels. This is discussed further below.
3. *Inferences Across Frequency Band Boundaries*: Low frequency band events do not

directly reveal their composition from higher frequency events. As a result, recording only low frequency events will typically result in information loss. Likewise, high frequency events do not, in themselves, reveal how they combine to form events at lower frequency bands. As a result, either low frequency band events must be recorded in conjunction with high frequency band events, or there must be some external model (e.g. a grammar) to describe how high frequency events combine to form lower frequency events. This too is discussed further below.

3.2 Grammatical Structure of Events

UI event sequences are often grammatical in structure. Grammars have been used in numerous disciplines to characterize the structure of sequential data. The main feature of grammars that make them useful in this context is their ability to define equivalence classes of patterns in terms of rewrite rules. For instance, the following grammar (expressed as a set of rewrite rules) may be used to represent the ways in which a user can trigger a print job in a given application:

```
PRINT_COMMAND ->  
  "MOUSE_PRESSED PrintToolBarButton" or  
  (PRINT_DIALOG_ACTIVATED then "MOUSE_PRESSED OkButton")
```

```
PRINT_DIALOG_ACTIVATED ->  
  "MOUSE_PRESSED PrintMenuItem" OR  
  "KEY_PRESSED 'Ctrl-P'"
```

Rule 1 simply states that the user can trigger a print job by either pressing the print toolbar button (which triggers the job immediately) or by activating the print dialog and then pressing "OK". Rule 2 specifies that the print dialog may be activated by either selecting the print menu item in the "File" menu, or by entering a keyboard accelerator, "Ctrl-P".

Let us assume that the lexical elements used to construct sentences in this language are:

- A: indicating “Print toolbar button pressed”
- B: indicating “Print menu item selected”
- C: indicating “Print accelerator key entered”
- D: indicating “Print dialog okayed”

Then the following “sentences” constructed from these elements each indicate a series of four consecutive print job activations:

AAAA
CDAAA
ABDBDA
BDCDACD
CDBDCDBD

All occurrences of ‘A’ indicate an immediate print job activation while all occurrences of ‘BD’ or ‘CD’ indicate a print job activated by using the print dialog and then selecting “OK”.

Notice that each of these sequences contains a different number of lexical elements. Some of them have absolutely no lexical elements in common (e.g. AAAA and CDBDCDBD). The lexical elements occupying the first and last positions differ from one sequence to the next. In short, there are a number of salient differences between these sequences at the lexical level. Techniques for automatically detecting sequences, comparing sequences, and characterizing sequences are often sensitive to such differences. Unless the data can somehow be transformed based on the above grammar, the fact that these sequences are semantically equivalent (in the sense that each indicates a

series of four consecutive print job activations) will most likely go unrecognized, and even simple summary statistics such as “# of print jobs per session” may be difficult to compute.

Techniques for extracting usability-related information from UI events must be able to take into consideration the grammatical relationships between lower level events and higher level events of interest.

3.3 Contextual Issues in Interpretation

Another set of issues arises in attempting to interpret the significance of events based only on the information carried within events themselves. To illustrate the problem more generally, consider the analogous problem of interpreting the significance of utterances in a transcript of a natural language conversation. Important contextual cues are often spread across multiple utterances or may be missing from the transcript altogether.

Let us assume we have a transcript of a conversation that took place between individuals A and B at a museum. The task is to identify A’s favorite paintings based on utterances in the transcript.

Example 1: “*The Persistence of Memory*, by Dali, is one of my favorites”.

In this case, everything we need to know in order to determine one of A’s favorite paintings is contained in a single utterance.

Example 2: “*The Persistence of Memory*, by Dali”.

In this case we need access to prior context. ‘A’ is most likely responding to a question posed by ‘B’. Information carried in the question is critical in interpreting the response. For example, the question *could* have been: “Which is your least favorite painting?”.

Example 3: “That is one of my favorites”.

In this case, we need the ability to de-reference an *indexical*. The information carried by the indexical “that” may not be available in any of the utterances in the transcript, but was clearly available to the interlocutors at the time of the utterance. Such contextual information was “there for the asking”, so to speak, and could have been noted had the transcriber been present and chosen to do so at the time of the utterance.

Example 4: “That is another one.”

In this case we would need access to both prior context *and* the ability to de-reference an indexical.

The following examples illustrate analogous situations in the interpretation of user interface events:

Example 1’: “MOUSE_PRESSED PrintToolBarButton”

This event carries with it enough information to indicate the action the user has performed.

Example 2': "MOUSE_PRESSED OkButton"

This event does not on its own indicate what action was performed. As in Example 2 above, this event indicates a response to some prior event, for example, a prior "MOUSE_PRESSED PrintMenuItem" event.

Example 3': "WINDOW_OPENED ErrorDialog"

The information needed to interpret the significance of this event *may* be available in prior events, but a more direct way to interpret its significance would be to query the dialog for its error message. This is similar to de-referencing an indexical, if we think of the error dialog as figuratively "pointing at" an error message that does not actually appear in the event stream.

Example 4': "WINDOW_OPENED ErrorDialog"

Assuming the error message is "Invalid Command", then the information needed to interpret the significance of this event is not only found by de-referencing the indexical (the error message "pointed at" by the dialog) but must be supplemented by information available in prior events. It may also be desirable to query contextual information stored in user interface components to determine the combination of parameters (specified in a dialog, for example) that led to this particular error.

The basic insight here is that sometimes an utterance — or a user interface event — does not carry enough information on its own to allow its significance to be properly interpreted. Sometimes critical contextual information is available elsewhere in the transcript, and sometimes that information is not available in the transcript, but *was* available, “for the asking”, at the time of the utterance, or event, but not afterwards. Therefore, techniques for extracting usability-related information from UI events must take into consideration the fact that context may be spread across multiple events, and that in some cases, important contextual information may need to be explicitly captured *during data collection* if meaningful interpretation is to be performed.

3.4 Composition of Events

Finally, user interactions may be analyzed at multiple levels of abstraction. For instance, one may be interested in analyzing low-level mouse movement and timing information, or one may be more interested in higher-level information regarding the steps taken by users in completing tasks, such as placing an order or composing a business letter. Techniques for extracting usability information from UI events should be capable of addressing events at multiple levels of abstraction.

Figure 3-2 illustrates a multi-level model of events (see [Hilbert et al. 1997] for an earlier treatment). At the lowest level are *physical events*, for example, fingers depressing keys or a hand moving a pointing device such as a mouse. *Input device events*, such as key and mouse interrupts, are generated by hardware in response to physical events. *Window system events* associate input device events with windows and other interface

objects on the screen. Events at this level include button presses, list and menu selections, focus and key events in input fields, and window movements and resizing.

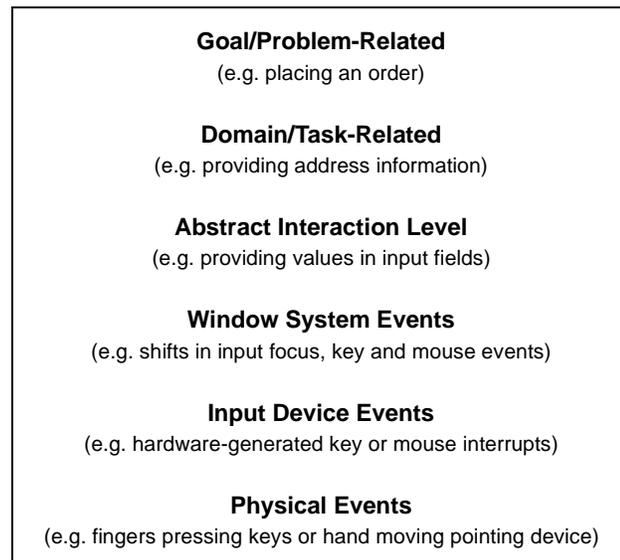


Figure 3-2. A multi-level model of events

Abstract interaction events are not directly generated by the window system, but may be computed based on window system events and other contextual information such as window system state. Abstract interaction events are indicated by recurring, idiomatic patterns of window system events, and indicate higher level concepts such as shifts in users' editing attention, or composite events such as the act of providing values by manipulating application components.

Consider a user editing a field at the top of a form-based interface, then pressing the "Tab" key repeatedly to edit a field at the bottom of the form. In terms of window system events, input focus shifted several times between the first and last fields. In terms

of abstract interaction events, the user's editing attention shifted directly from the top field to the bottom field. Notice that detecting the occurrence of abstract interaction events such as "GOT_EDIT" and "LOST_EDIT" requires the ability to keep track of the last edited component and to notice subsequent editing events in other components.

Another type of abstract interaction event might be associated with the act of providing a new value to an application by manipulating user interface components. In the case of a text field, this would mean that the field had received a number of key events, was no longer receiving key events, and now contains a new value. The patterns of window system events that indicate an abstract interaction event such as "VALUE_PROVIDED" will differ from one type of interface component to another, and from one application to another, but will typically remain fairly stable within a given application. Notice that detecting the occurrence of an abstract interaction event such as "VALUE_PROVIDED" requires the ability to access user interface state such as the component value before and after editing events.

Domain/task-related and goal/problem-related events are at the highest levels. Unlike other levels, these events indicate progress in the user's tasks and goals. Inferring these events based on lower level events can be straightforward when, for example, the user interface provides explicit support for structuring tasks or indicating goals. For instance, Wizards in Microsoft Word™ lead users through a sequence of steps in a predefined task. The user's progress can be recognized in terms of simple window system events such as button presses on the "Next" button. In other cases, inferring task and goal related events may require more complicated composite event detection. For instance, the

goal of placing an order includes the task of providing address information. The task-related event “ADDRESS_PROVIDED” may be recognized in terms of “VALUE_PROVIDED” abstract interaction events occurring within each of the required fields in the address section of the form. Finally, in some cases, it may be impossible to infer events at these levels based only on lower level events.

Techniques for extracting usability-related information from UI events must be sensitive to the fact that user interactions can occur and be analyzed at multiple levels of abstraction.

CHAPTER 4: Related Work

The fundamental goal of this chapter is to present a framework to help HCI researchers and practitioners categorize, compare, and evaluate the relative strengths and limitations of approaches that have been, or might fruitfully be, applied to the problem of exploiting user interface event data in evaluating application usage and usability. Because exhaustive coverage of all existing and potential approaches is impossible, an attempt has been made to identify key characteristics of existing approaches that divide them into more or less natural categories. This allows classes of approaches, not just instances, to be compared. The hope is that illuminating comparison can be conducted at the class level, and that classification of new instances into existing classes will prove to be unproblematic. The chapter concludes with a summary of the state-of-the-art and a list of problems limiting the scalability and data quality of current approaches.

4.1 Extracting Usability Information from User Interface Events

User interface events, or UI events, are typically generated as natural products of the normal operation of window-based user interface systems such as those provided by the Macintosh Operating System, Microsoft Windows, X Window System, and Java Abstract Window Toolkit. Such events indicate user behavior with respect to the components that make up an application's user interface (e.g. mouse movements with respect to application windows, keyboard presses with respect to application input fields, mouse clicks with respect to application buttons, menus, and lists). Because such events can be automatically captured and because they indicate, albeit at low levels of

abstraction, user behavior with respect to an application's user interface, they have long been regarded as a potentially fruitful source of information regarding application usage and usability. However, because such events are typically extremely voluminous and rich in detail, automated support is generally required to extract information at a level of abstraction that is useful to investigators interested in analyzing application usage or usability.

While there are a number of potentially related techniques that have been applied to the problem of analyzing sequential data in other domains, this chapter surveys techniques that have been applied within the domain of HCI. Providing an in-depth treatment of all potentially related techniques would necessarily limit the amount of attention paid to characterizing the approaches that have in fact been brought to bear on the specific problems associated with analyzing UI events. However, this and the previous chapter together attempt to characterize user interface events and analysis techniques in such a way as to make comparison between techniques used in HCI and those used in other domains straightforward.

Ideally, an empirical evaluation of these approaches in practice would help elucidate more precisely the specific types of usability questions for which each approach is most suited, as well as the related costs and benefits of applying each approach. However, because many of the approaches have never been realized beyond the research prototype stage, little empirical work has been performed to evaluate their relative strengths and limitations. This chapter provides a conceptual evaluation by distinguishing classes of approaches and illuminating their underlying nature. As a result, this survey

should be regarded as a guide to the research and not as a guide to selecting an already implemented approach for use in practice.

The high level comparison framework that has emerged as a result of this survey is summarized here:

- *Techniques for synchronization and searching.* These techniques allow user interface events to be synchronized and cross-indexed with other sources of data such as video and coded observations. This allows searches in one medium to locate supplementary information in others. In some ways, this is the most simple (i.e. mechanical) technique for exploiting user interface events in usability evaluations. It is, however, quite powerful.
- *Techniques for transforming event streams.* Transformation involves selecting, abstracting, and recoding events in order to facilitate human analysis (including pattern detection, comparison, and characterization), or to prepare event data as input to automatic techniques for performing these functions. *Selection* allows events and sequences of interest to be separated from the “noise”. *Abstraction* allows events to be related to higher level concepts of interest in analysis. *Recoding* involves generating a new event stream based on the results of selection and abstraction in order to allow selected and abstracted events to be subjected to the same types of manual and automated analysis techniques normally performed on raw event streams.
- *Techniques for performing counts and summary statistics.* Once events have been captured, there are a number of counts and summary statistics that might be computed to summarize user behavior, for example, feature use counts, error frequencies, use of the help system, and so forth. Although most investigators rely on general-purpose spreadsheets and statistical packages to provide such functionality, some investigators have proposed domain-specific “built-in” functions for calculating and reporting this sort of summary information.
- *Techniques for detecting sequences.* These techniques allow investigators to identify occurrences of concrete or abstractly defined “target” sequences within “source” sequences of events that may indicate potential usability issues. In some cases, target sequences may be abstractly defined and are supplied by the developers of the technique. In other cases, target sequences may be more specific to particular applications and are supplied by the users of the technique. Sometimes the purpose is to generate a list of matched source subsequences for further perusal by the investigator. Other times the purpose is to automatically recognize particular

sequences that violate expectations about proper application use. Finally, in some cases, the purpose may be to perform transformation of the source sequence by abstracting and recoding matches of the target sequence into “abstract” events.

- *Techniques for comparing sequences.* These techniques help investigators compare “source” sequences against concrete or abstractly defined “target” sequences indicating the extent to which the sequences match, or *partially* match, one another. Some techniques attempt to detect divergence between an abstract model representing the target sequence and a source sequence. Others attempt to detect divergence between a concrete target sequence produced, for example, by an expert user, and a source sequence produced by some other user. Some produce diagnostic measures of distance to characterize the correspondence between target and source sequences. Others attempt to perform the best possible alignment of events in the target and source sequences and present the results to investigators in visual form. Still others use points of deviation between the target and input sequences to automatically indicate potential usability issues. In all cases, the purpose is to compare actual sequences of events against some model or trace of “ideal” or expected sequences to identify potential usability issues.
- *Techniques for characterizing sequences.* These techniques take “source” sequences as input and attempt to construct an abstract model to summarize, or characterize, interesting sequential features of those sequences. Some techniques compute probability matrices allowing process models with probabilities associated with transitions to be produced. Others construct grammatical models or finite state machines to characterize the grammatical structure of events in the source sequences.
- *Visualization techniques.* These techniques present the results of transformations and analyses in forms allowing humans to exploit their innate visual analysis capabilities to interpret results. These techniques can be particularly useful in linking results of analysis back to features of the interface.
- *Integrated evaluation support.* Integrated support is provided by evaluation environments that facilitate flexible composition of various transformation, analysis, and visualization capabilities. Some environments also provide built-in support for managing domain-specific artifacts such as evaluations, subjects, tasks, data and results of analysis.

Figure 4-1 illustrates how the framework might be visualized as a hierarchy. At the highest level, the surveyed techniques are concerned primarily with: synchronization and searching, transformation, analysis, visualization, and integrated support.

Transformation can be achieved through selection, abstraction, and recoding. Analysis can be performed using counts and summary statistics, sequence detection, sequence comparison, and sequence characterization.

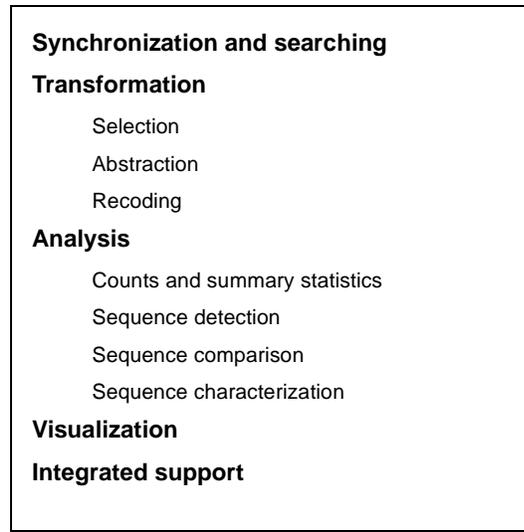


Figure 4-1. Related work comparison framework

The following sections discuss in more detail the features that distinguish each class of approaches and provide examples of some of the approaches in each class. Related work is discussed where appropriate. The relative strengths and limitations of each class are also discussed. Figure 4-2 through Figure 4-9 provide pictorial representations of the key features underlying each class.

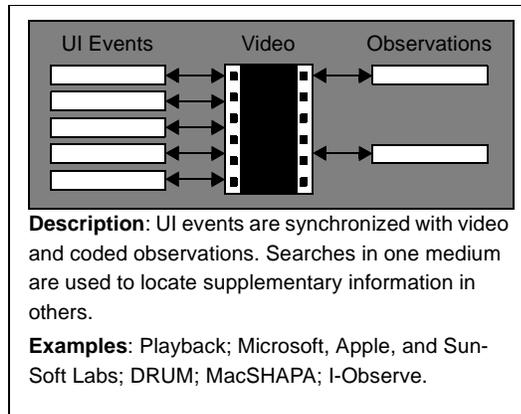


Figure 4-2. Synchronization and Searching

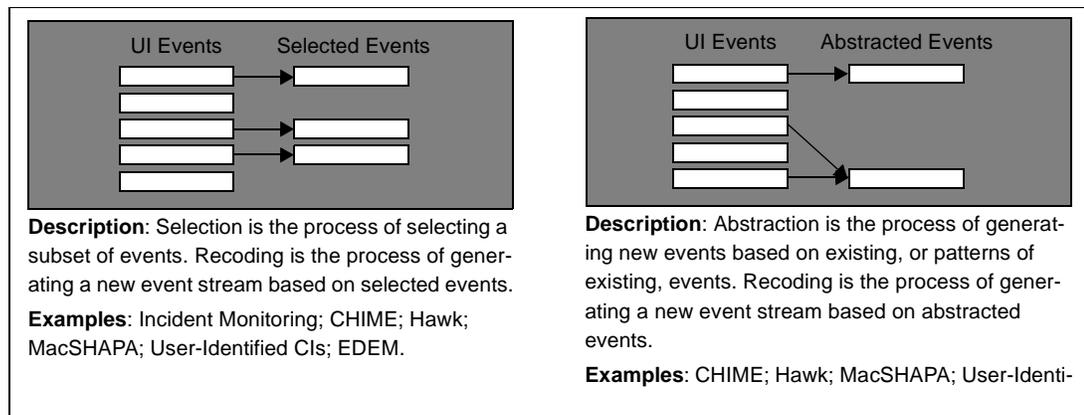


Figure 4-3. Transformation

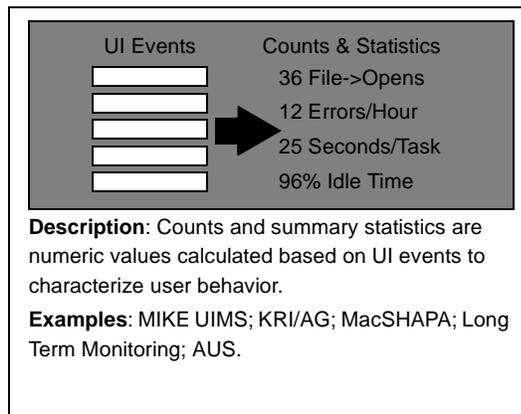


Figure 4-4. Counts and Summary Statistics

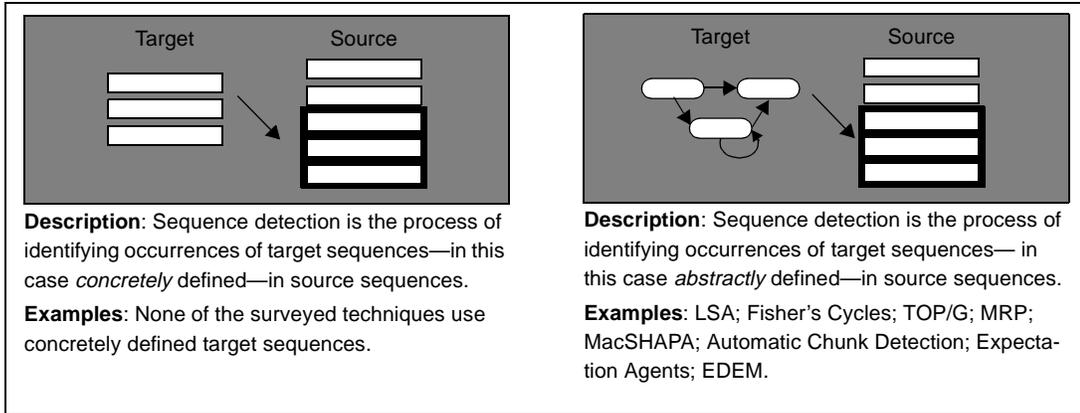


Figure 4-5. Sequence Detection

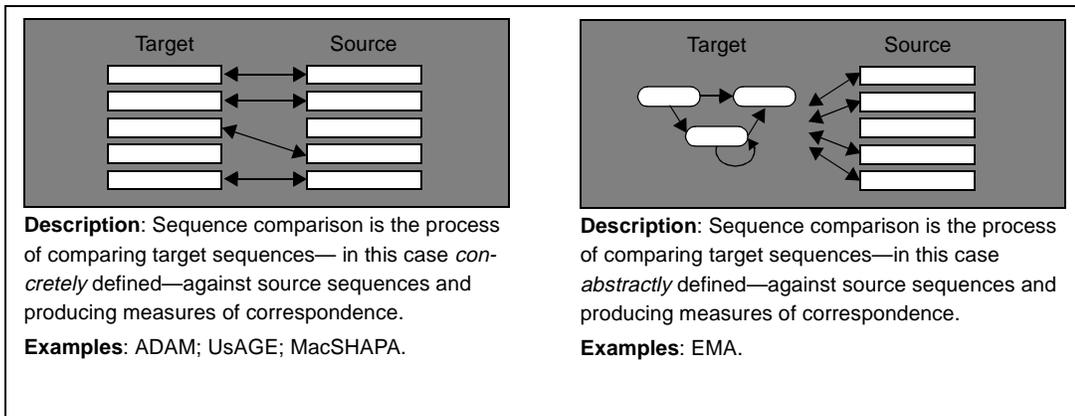


Figure 4-6. Sequence Comparison

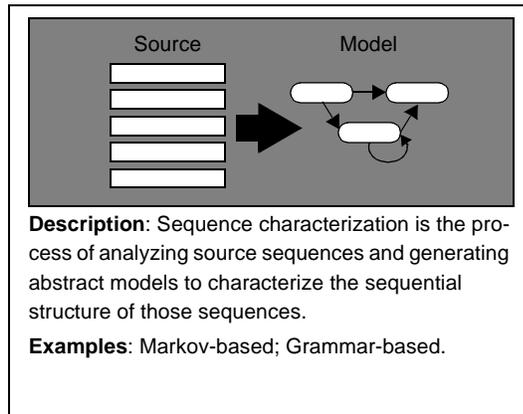


Figure 4-7. Sequence Characterization

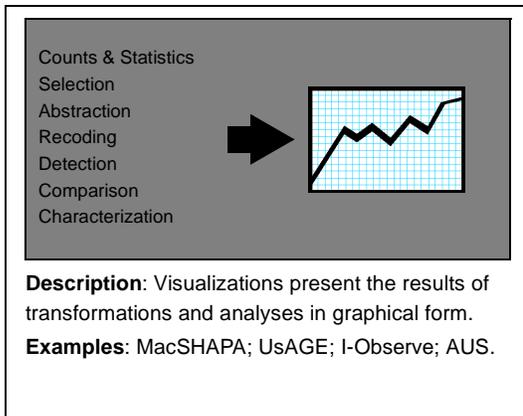


Figure 4-8. Visualization

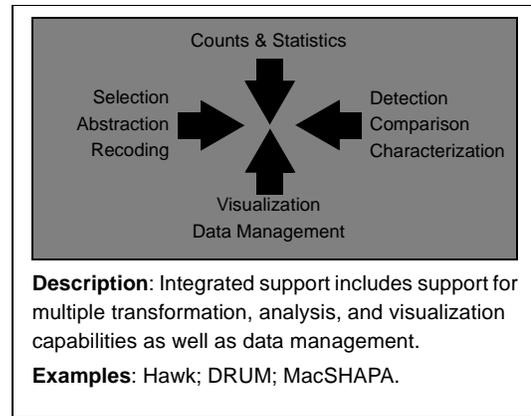


Figure 4-9. Integrated Support

4.2 Synchronization and Searching

4.2.1 Purpose

User interface events provide detailed information regarding user behavior that can be easily captured, searched, counted, and analyzed using automated tools. However, higher level events of interest can be difficult to infer from user interface events, and sometimes critical contextual information is simply missing from the event stream, making proper interpretation challenging at best.

Synchronization and Searching techniques seek to combine the advantages of UI event data with the advantages provided by more semantically rich observational data, such as video recordings and experimenters' observations.

By synchronizing UI events with other sources of data such as video or coded observations, searches in one medium can be used to locate supplementary information in others. Therefore, if an investigator wishes to review all segments of a video in which a user uses the help system or invokes a particular command, it is not necessary to manually

search the entire recording. The investigator can: (a) search through the log of UI events for particular events of interest and use the timestamps associated with those events to automatically cue up the video recording, or (b) search through a log of observations (that were entered by the investigator either during or after the time of the recording) and use the timestamps associated with those observations to cue up the video. Similarly, segments of interest in the video can be used to locate the detailed user interface events associated with those episodes.

4.2.2 Examples

Playback [Neal and Simmons 1983] is an early example of a system employing synchronization and searching capabilities. In Playback, UI events are collected automatically and experimenters enter coded observations and comments, during or after the evaluation session, which are automatically synchronized with the captured events. Instead of using video, Playback allows recorded events to be played back through the application interface to re-trace the user's actions. The evaluator can step through the playback based on events or coded observations as if using an interactive debugger. A handful of simple analyses are built-in that automatically calculate counts and summary statistics. This technique captures less information than video-based techniques since video can also be used to record off-line behavior such as facial gestures, off-line documentation use, and verbalizations. Also, there can be problems associated with replaying user sessions accurately in applications where behavior is affected by events outside of user interactions. For instance, the behavior of some applications may vary depending on the state of networks and persistent data stores.

DRUM, the Diagnostic Recorder for Usability Measurement, is an integrated evaluation environment that supports video-based usability evaluation [Macleod et al. 1993]. DRUM was developed at the National Physical Laboratory as part of the ESPRIT Metrics for Usability Standards in Computing Project (MUSiC). DRUM features a module for recording and synchronizing events, observations, and video (Recording Logger), a module for defining and managing observation coding schemes (Scheme Manager), a module for calculating pre-defined counts and summary statistics (Log Processor), and a module for managing and manipulating evaluation-related information regarding subjects, tasks, recording plans, logs, videos, and results of analysis (Evaluation Manager).

Usability specialists at Microsoft, Apple, and SunSoft all report the use of in-house tools providing synchronization and searching capabilities [Weiler et al. 1993; Hoiem and Sullivan 1994]. The tools used at Microsoft include a tool for logging observations (Observer), a tool for tracking UI events (Tracker), and a tool for synchronizing and reviewing data from the multiple sources (Reviewer). The tools used at Apple and SunSoft are essentially similar. All tools support some level of event filtering as part of the capture process. Apple's filtering appears to be user-definable while Microsoft and SunSoft's filtering appear to be programmed into the capture tools. Scripts and general purpose analysis programs, such as Excel, are used to perform counts and summary statistics after capture. All tools support video annotations to produce "highlights" videos. Microsoft's tools provide an API to allow applications to report application-specific events, or events not readily available in the UI event stream.

I-Observe, the Interface OBServation, Evaluation, Recording, and Visualization Environment developed at Georgia Tech [Badre et al. 1995] also provides synchronization and searching capabilities. I-Observe is a set of loosely integrated tools for collecting, selecting, analyzing, and visualizing event data. Searches are performed by specifying predicates over the fields contained within a single event record. Patterns of events can then be located by stringing together a set of such search specifications into a regular expression. The intervals (identified by begin and end events) matched by the regular expressions can be used to automatically select data for visualization, or to drive the display of corresponding segments of the video tape.

4.2.3 Strengths

The strengths of these techniques lie in their ability to integrate data sources with complementary strengths and weaknesses, and to allow searches in one medium to locate related information in the others. UI events provide detailed performance information that can be searched, counted, and analyzed using automated techniques, however, UI events often leave out higher level contextual information that can be more easily captured using video recordings and coded observations.

4.2.4 Limitations

Techniques relying on synchronizing UI events with video and coded observations require the use of video recording equipment and the presence of observers. The use of video equipment and the presence of observers can make subjects self-conscious and affect performance, and may not be practical or permitted in certain circumstances.

Furthermore, video-based evaluations tend to produce massive amounts of data that can be expensive to analyze. The ratio of the time spent in analysis versus the duration of the sessions being analyzed has been known to reach 10:1 [Sanderson and Fisher 1994; Nielsen 1993; Sweeny 1993]. These matters are all serious limiting factors on evaluation size, scope, location, and duration.²

4.3 Transformation

4.3.1 Purpose

These techniques combine selection, abstraction, and recoding to transform event streams for various purposes, such as facilitating human pattern detection, comparison, and characterization, or to prepare data for input into automatic techniques for performing these functions.

Selection operates by subtracting information from event streams, allowing events and sequences of interest to emerge from the “noise”. Selection involves specifying constraints on event attributes to indicate: (a) events of interest to be separated from other events, or (b) events to be separated from events of interest. For instance, one may elect to disregard all events associated with mouse movements in order to focus analysis on higher level actions such as button presses and menu selections. This can be accomplished by “positively” selecting button press and menu selection events, or by “negatively” selecting, or filtering, mouse movement events.

2. While some of the location limitations can be addressed using remote collaboration technologies, such as software video-conferencing and application sharing technologies, more work must be done in the area of automating evaluation if current restrictions on size, scope, and duration are to be addressed.

Abstraction operates by “synthesizing” new information that can be added to the event stream based on patterns of events in the event stream. For instance, an abstract “USE” event might be synthesized to indicate the use of arbitrary UI components, despite the fact that there is typically no single event in the raw event stream that signifies this abstract event. For instance, mouse events typically signify use of non-textual components while keyboard events typically signify use of textual components. One may also wish to synthesize events to relate the use of particular UI components to higher level concepts such as the use of menus, toolbars, or dialogs to which those components belong.

Recoding involves producing new event streams based on the results of selection and abstraction. This allows the same manual or automated analysis techniques normally applied to raw event streams to be applied to selected and abstracted events, potentially leading to different results. This is discussed in further detail in Chapter 6.

4.3.2 Examples

Chen [Chen 1990] presents an approach to user interface event monitoring that selects events based on the notion of “incidents”. Incidents are defined as only those events that actually trigger some response from the application, and not just the user interface system. This technique was demonstrated by modifying the Xt Intrinsic Toolkit to report events that trigger callback procedures registered by applications. This allows events not handled by the application to be filtered automatically. Investigators may further constrain event reporting by selecting specific incidents of interest to be reported. Widgets in the user interface toolkit were modified to provide a query procedure to return

limited contextual information when an event associated with a given widget triggers a callback.

Hartson and colleagues [Hartson et al. 1996] report an approach to remote collaborative usability evaluation that relies, in part, on users to select events. Users identify potential usability problems that arise during the course of interacting with an application, and report information regarding these “critical incidents” by pressing a “report” button that is supplied in the interface. Traces of the UI events leading up to and following the incident are reported via E-mail along with contextual information that is provided by the user. In this case, selection is achieved by only reporting the n events leading up to, and m events following, user-identified critical incidents (where n and m are parameters that can be set in advance by investigators).

CHIME [Badre and Santos 1991a] is a computer-human interaction monitoring engine developed at Georgia Tech that is similar, in some ways, to Chen’s approach. CHIME allows investigators to specify ahead of time which events to report and which events to filter. An important difference is that CHIME supports a limited notion of abstraction that allows a level of indirection to be built on top of the window system. The basic idea is that abstract “interaction units” (IUs) are defined that translate window system events into platform independent events upon which further monitoring infrastructure is built. The events to be recorded are then specified in terms of these platform independent IUs.³

Hawk [Guzdial 1993] is an environment for selecting, abstracting, and recoding events in log files. Hawk's main functionality is provided by a variant of the AWK programming language [Aho et al. 1988], and an environment for managing data files is provided by HyperCard™. Events appear in the event log one per line, and AWK pattern-action pairs are used to specify what is to be matched in each line of input (the pattern) and what is to be printed as output (the action). This allows fairly flexible selection, abstraction, and recoding to be performed.

MacSHAPA [Sanderson et al. 1994], which is discussed further below, supports selection and recoding in the form of a database query and manipulation language which allows event records to be selected based on attributes, and new streams defined based on the results of queries. Abstraction can be performed manually by entering new records representing abstract events and associating them with existing event records.

4.3.3 Strengths

The main strength of these approaches lies in their explicit support for selection, abstraction, and recoding which are essential steps in preparing UI events for most types of analysis. Chen and CHIME address issues of selection prior to reporting. Hartson and colleagues add to this a technique for accessing contextual information via the user. All these techniques could potentially be used to collect UI events remotely.

-
3. The authors also allude to the possibility of allowing higher level IUs to be hierarchically defined in terms of lower level IUs (using a context-free grammar and pre-conditions) to provide a richer notion of abstraction. However, this appears to never have been implemented [Badre and Santos 1991a & 1991b).

Hawk and MacSHAPA, on the other hand, do not address event collection but provide powerful and flexible environments for transforming and analyzing already captured UI events.

4.3.4 Limitations

The techniques that select, abstract, and recode events while collecting them run the risk of discarding events that could have proven useful in analysis. Techniques that rely exclusively on users to select events are even more likely to miss useful information. The approaches that support flexible selection, abstraction, and recoding do so only after events have been captured, meaning contextual information that may be critical in transformation is not available.

4.4 Counts and Summary Statistics

4.4.1 Purpose

As noted above, one of the key benefits of UI events is how readily details regarding on-line behavior can be captured and manipulated using automated techniques. Most investigators rely on general purpose analysis programs, such as spreadsheets and statistical packages, to compute counts and summary statistics based on collected data (e.g. feature use counts or error frequencies). However, some investigators have proposed systems with built-in facilities for performing and reporting specific calculations. This section provides examples of some of the systems boasting specialized facilities for calculating usability-related metrics.

4.4.2 Examples

The MIKE user interface management system (UIMS) is an early example of a system offering built-in facilities for calculating and reporting usability metrics [Olsen and Halversen 1988]. Because MIKE controls all aspects of input and output activity, and because it has an abstract description of the user interface and application commands, MIKE is in a uniquely good position to monitor UI events and associate them with responsible interface components and application commands. Example metrics include:

- Performance time: How much time is spent completing tasks such as specifying arguments for commands?
- Mouse travel: Is the sum of the distances between mouse clicks unnecessarily high?
- Command frequency: Which commands are used most frequently or not at all?
- Command pair frequency: Which commands are used together frequently? Can they be combined or placed closer to one another?
- Cancel and undo: Which dialogs are frequently canceled? Which commands are frequently undone?
- Physical device swapping: Is the user switching back and forth between keyboard and mouse unnecessarily? Which features are associated with high physical swapping counts?

MIKE logs all UI events and associates them with the interface components triggering them and the application commands triggered by them. Event logs are written to files that are later read by a metric collection and report generation program. This program uses the abstract description of the interface to interpret the log and to generate human readable reports summarizing the metrics.

MacSHAPA [Sanderson et al. 1994] also includes numerous built-in features to support computation and reporting of various counts and summary statistics.

Automatic Usability Software (AUS) [Chang and Dillon 1997] is reported to provide a number of automatically computed metrics such as help system use, use of cancel and undo, mouse travel, and mouse clicks per window.

Finally, ErgoLight Operation Recording Suite (EORS) and Usability Validation Suite (EUVS) [ErgoLight Usability Software 1998] provide a number of built-in counts and summary statistics to characterize user interactions captured either locally in the usability lab, or remotely over the Internet.

4.4.3 Related

A number of commercial tools such as Aqueduct AppScope™ [Aqueduct Software 1998] and Full Circle TalkBack™ [Full Circle Software 1998] have recently become available for capturing data about application crashes over the Internet. These tools typically capture metrics about the operating system and application at the time crashes occur, and sometimes provide users with the ability to provide feedback regarding what they were doing when the crash occurred. These tools sometimes also provide an API by which applications may report data of interest, such as information about application feature use. This information is then sent via E-mail to developers' computers where it is stored in a database and may be plotted using standard database plotting facilities.

4.4.4 Strengths

With the number of possible metrics, counts, and summary statistics that might be computed and that might be useful in usability evaluation, it is nice that some systems provide built-in facilities to perform and report such calculations automatically.

4.4.5 Limitations

With the exception of MacSHAPA, the systems described above do not provide facilities to allow evaluators to modify built-in counts, statistics, and reports or to add new ones of their own. Also, the computation of useful metrics is greatly simplified when the system computing the metrics has a model of the user interface and application commands, as in the case of MIKE, or when the application code must be manually instrumented to report the events to be analyzed, as in the case of Aqueduct AppScope and Full Circle TalkBack. AUS does not address application-specific features and thus is limited in its ability to relate metrics results to application features.

4.5 Sequence Detection

4.5.1 Purpose

These techniques are used to detect occurrences of concrete or abstractly defined “target” sequences within “source” sequences. In some cases target sequences are abstractly defined, and are supplied by the developers of the technique (e.g. Fisher’s cycles, lag sequential analysis, multiple repeating pattern analysis, and automatic chunk detection). In other cases, target sequences may be more specific to a particular application and are supplied by the investigators using the technique (e.g. TOP/G).

Sometimes the purpose is to generate a list of matched source subsequences for perusal by the investigator (e.g. Fisher's cycles, maximal repeating pattern analysis, and automatic chunk detection). Other times the purpose is to recognize sequences of UI events that violate particular expectations about proper UI usage (e.g. TOP/G). Finally, in some cases the purpose may be to perform abstraction and recoding of the source sequence based on matches of the target sequence.

4.5.2 Examples

TOP/G [Hoppe 1988] is a task-oriented parser/generator that parses sequences of commands from a command-line simulation and attempts to infer the higher level tasks that are being performed. Expected tasks are modeled in a notation based on Payne and Green's task-action grammars [Payne and Green 1986] and are represented in a Prolog database as rewrite or production rules. Composite tasks are defined hierarchically in terms of elementary tasks which are in turn decomposed into "triggering rules" that map keystroke level events into elementary tasks. Rules may also be defined to recognize "suboptimal" user behaviors that might be abbreviated by simpler compositions of commands. A later version attempted to use information about the side effects of commands in the environment to recognize when a longer sequence of commands might be replaced by a shorter sequence. In both cases, the generator functionality of TOP/G could be used to produce the shorter, or more "optimal", command sequence.

A number of techniques for detecting abstractly defined patterns in sequential data have been applied by researchers involved in exploratory sequential data analysis

(ESDA). For an in-depth treatment see [Sanderson and Fisher 1994]. These techniques can be subdivided into two basic categories:

Techniques sensitive to sequentially separated patterns of events, for example:

- Fisher's cycles
- Lag sequential analysis (LSA)

Techniques sensitive to strict transitions between events, for example:

- Maximal Repeating Pattern Analysis (MRP)
- Log linear analysis
- Time-series analysis

Fisher's cycles [Fisher 1991] allow investigators to specify beginning and ending events of interest that are then used to automatically identify all occurrences of subsequences beginning and ending with those events (excluding those with further internal occurrences of those events). For example, assume an investigator is faced with a source sequence of events encoded using the letters of the alphabet, such as: ABACDACDBADBCACCCD. Suppose further that the investigator wishes to find out what happened between all occurrences of 'A' (as a starting point) and 'D' (as an ending point), Fisher's cycles produces the following analysis:

Source sequence: ABACDACDBADBCACCCD
 Begin event: A
 End event: D

Output:

ABACDACDBADBCACCCD
 ABACDACDBADBCACCCD
 ABACDACDBADBCACCCD
 ABACDACDBADBCACCCD

Cycle #	Frequency	Cycle
1	2	ACD
2	1	AD
3	1	ACCCD

The investigator could then note that there were clearly no occurrences of B in any A->D cycle. Furthermore, the investigator might use a grammatical technique to recode repetitions of the same event into a single event, thereby revealing that the last cycle (ACCCD) is essentially equivalent to the first two (ACD). This is one way of discovering similar subsequences in “noisy” data.

Lag sequential analysis (LSA) is another popular technique that identifies the frequently with which two events occur at various “removes” from one another [Sackett 1978; Allison and Liker 1982; Faraone and Dorfman 1987; Sanderson and Fisher 1994]. LSA takes one event as a ‘key’ and another event as a ‘target’ and reports how often the target event occurs at various intervals before and after the key event. If ‘A’ were the key and ‘D’ the target in the previous example, LSA would produce the following analysis:

Source sequence: ABACDACDBADBCACCCD
 Key event: A
 Target event: D
 Lag(s): -4 through +4

Output:

Lag	-4	-3	-2	-1	1	2	3	4
Occurrences	0	1	1	1	1	2	0	1

The count of 2 at Lag = +2 corresponds to the two ACD cycles identified by Fischer's cycles above. Assuming the same recoding operation performed above to collapse multiple occurrences of the same event into a single event, this count would increase to 3. The purpose of LSA is to identify correlations between events (that might be causally related to one another) that might otherwise have been missed by techniques more sensitive to the strict transitions between events.

An example of a technique that is more sensitive to strict transitions is the Maximal Repeating Pattern (MRP) analysis technique [Siochi and Hix 1991]. MRP operates under the assumption that repetition of user actions can be an important indicator of potential usability problems. MRP identifies all patterns occurring repeatedly in the input sequence and produces a listing of those patterns sorted by length first followed by frequency of occurrence in the source sequence. MRP applied to the sequence above would produce the following analysis:

Source Sequence: ABACDACDBADBCACCCD

Output:

Pattern #	Frequency	Pattern
1	2	ACD
2	3	AC
3	3	CD
4	2	BA
5	2	DB

MRP is similar in spirit to Fisher's cycles and LSA, however, the investigator does not specify particular events of interest. Notice that the ACCCD subsequence identified in the previous examples is not identified by MRP.

Markov-based techniques can be used to compute the transition probabilities from one or more events to the next event. Statistical tests can be applied to determine whether the probabilities of these transitions is greater than would be expected by chance [Sanderson and Fisher 1994]. Other related techniques include log linear analysis [Gottman and Roy 1990] and formal time-series analysis [Box and Jenkins 1976]. All of these techniques attempt to find strict sequential patterns in the data that occur more frequently than would be expected by chance.

Santos and colleagues have proposed an algorithm for detecting users' "mental chunks" based on pauses and flurries of activity in human computer interaction logs [Santos et al. 1994]. The algorithm is based on an extension of Fitts' law [Fitts 1964] that predicts the expected time between events generated by a user who is actively executing plans, as opposed to engaging in problem solving and planning activity. For each event transition in the log, if the pause in interaction cannot be justified by the predictive model, then the lag is assumed to signify a transition from "plan execution phase" to "plan acquisition phase" [Santos et al. 1994]. The results of the algorithm are used to segment the source sequence into plan execution chunks and chunks most probably associated with problem solving and planning activity. The assumption is that expert users tend to have longer, more regular execution chunks than novice users, so user expertise might be inferred on the basis of the results of the chunking algorithm.

4.5.3 Related

EBBA [Bates 1995] is a distributed debugging system that attempts to match the behavior of a distributed program against partial models of expected behavior. EBBA is similar to the work presented in this dissertation, particularly in its ability to abstract and recode the event stream based on hierarchically defined abstract events.⁴

Amadeus [Selby et al. 1991] and YEAST [Krishnamurthy and Rosenblum 1995] are event-action systems used to detect and take actions based on patterns of events in software processes that are also similar in spirit to the work presented here. Techniques that have been used to specify behavior of concurrent systems, such as the Task Sequencing Language (TSL) as described in [Rosenblum 1991] are also related. Some of the same techniques used for specifying and detecting patterns of events in these approaches may be usefully applied to the problem of specifying and detecting patterns of UI events.

4.5.4 Strengths

The strengths of these approaches lie in their ability to help investigators detect patterns of interest in events, and not just perform analysis on isolated events. The techniques associated with ESDA allow patterns that may not have been anticipated to be discovered. Languages for detecting patterns of interest in UI events based, for example,

4. EBBA is sometimes characterized as a sequence comparison system since the information carried in a partially matched model can be used to help the investigator better understand where the program's behavior has gone wrong (or where a model is inaccurate). However, EBBA does not directly indicate that partial matches have occurred or provide any diagnostic measures of correspondence. Rather, the user must notice that a full match has failed, and then manually inspect the state of the pattern matching mechanism to see which events were matched and which were not.

on extended regular expressions [Sanderson and Fisher 1994], or on more grammatically inspired techniques [Hoppe 1988; Hilbert and Redmiles 1998b] can be used to locate patterns of interest and to transform event streams by recoding abstract patterns of events into “abstract” events.

4.5.5 Limitations

The ESDA techniques described above tend to produce large amounts of output that can be difficult to interpret, and that frequently do not lead to identification of usability problems [Cuomo 1994]. The non-ESDA techniques require investigators to know how to specify the patterns they are looking for, and to define them (sometimes painstakingly) before analysis can be performed.

4.6 Sequence Comparison

4.6.1 Purpose

These techniques compare “source” sequences against concrete or abstractly defined “target” sequences indicating *partial* matches between the two. Some techniques attempt to detect divergence between an abstract model of the target sequence and the source sequence (e.g. EMA and USINE). Others attempt to detect divergence between a concrete target sequence produced, for example, by an expert user and a source sequence produced by some other user (e.g. ADAM and UsAGE). Some produce diagnostic measures of distance to characterize the correspondence between target and source sequences (e.g. ADAM). Others attempt to perform the best possible alignment of events in target and source sequences and present the results visually (e.g. UsAGE and

MacSHAPA). Still others use points of deviation between the target and input sequences to automatically indicate potential errors or “critical incidents” (e.g. EMA and USINE). In all cases, the purpose is to compare actual usage against some model or trace of “ideal” or expected usage to identify potential usability problems.⁵

4.6.2 Examples

ADAM [Finlay and Harrison 1990], is an advanced distributed associative memory that compares fixed length source sequences against a set of target sequences that were used to “train” the memory. Target sequences used to train the memory are associated with “classes” of event patterns by the investigator. When a source sequence is input, the associative memory identifies the class that most closely matches the source sequence, and two diagnostic measures are output: a “confidence” measure that is 100% only when the source sequence is identical to one of the trained target sequences, and a “distance” measure, indicating how far the source pattern is from the next “closest” class. Investigators then use these measure to determine whether a source sequence is different enough from the trained sequences to be judged as a possible “critical incident”. Incidentally, ADAM might also be trained on examples of “expected” critical incidents so that these might be detected directly.

5. TOP/G and the work described in this dissertation are also intended to detect deviations between actual and expected usage to identify potential usability problems. However, these approaches are better characterized as detecting complete matches between source sequences and “negatively defined” target patterns that indicate unexpected, or suboptimal behavior, as opposed to partially matching, or comparing, source sequences against “positively defined” target patterns.

MacSHAPA [Sanderson and Fisher 1994] provides techniques for aligning two sequences of events as optimally as possible based on maximal common subsequences [Hirschberg 1975]. The results are presented visually as cells in adjacent spreadsheet columns with aligned events appearing in the same row, and missing cells indicating events in one sequence that could not be aligned with events in the other (Figure 4-14).

A related technique is applied in UsAGE [Ueling and Wolf 1995] where a source sequence of UI events (related to performance of some task) is aligned as optimally as possible with a target sequence (produced by an “expert” performing the same task), and presented in visual form.

EMA, an automatic analysis mechanism for the evaluation of user interfaces [Balbo 1996], requires that investigators provide a grammar-based model describing all the “legal” paths through a particular user interface. An evaluation program is then used to compare a log of events generated by use of the interface against the model, indicating in the log and the model where the user has taken “illegal”, or unexpected, paths. Other simple patterns, for example, the use of cancel, are also detected and reported. This information can then be used by the evaluator to identify problems in the interface (or problems in the model).

USINE is a similar technique [Lecerof and Paterno 1998], in which investigators use a hierarchical task notation to specify how low level actions can be composed into higher-level tasks, and to specify sequencing constraints on actions and tasks. The tool then compares logs of user actions against the task model. All actions not specified in the

task model or actions and tasks performed “out of order” according to the sequencing constraints specified in the task model are flagged as potential errors. The tool then computes a number of built-in counts and summary statistics regarding errors and other basic metrics (e.g., window resizing and scrollbar usage) and generates simple graphs.

ErgoLight Usability Validation Suite (EUVS) also allows user interactions to be compared against hierarchical representations of user tasks [ErgoLight Usability Software 1998]. It is similar in spirit to EMA and USINE with the added benefit that it provides a number of built-in counts and summary statistics regarding general user interface use in addition to automatically-detected divergences between the task model and user actions.

4.6.3 Related

Software process validation techniques [Cook and Wolf 1997] are related in that they compare actual traces of events generated by a software process against an abstract model of the intended process. A diagnostic measure of distance is computed to indicate the correspondence between the trace and the closest acceptable trace produced by the model. Techniques for performing error correcting parsing are also related. See [Cook and Wolf 1997] for further discussion and pointers to related literature.

4.6.4 Strengths

The strengths of these approaches lie their ability to compare actual traces of events against expected traces, or models of expected traces, in order to identify potential usability problems. This is particularly appealing when expected traces can be specified “by demonstration” as in the case of ADAM and UsAGE.

4.6.5 Limitations

Unfortunately, all of these techniques have significant limitations.

A key limitation of any approach that compares source sequences against concrete target sequences is the underlying assumption that: (a) source and target sequences can be easily segmented for piecemeal comparison, as in the case of ADAM, or (b) that whole interaction sequences produced by different users will actually exhibit reasonable correspondence, as in the case of UsAGE.

Furthermore, the output of all these techniques, (except in the case of perfect matches) requires expert human interpretation to determine whether the sequences are interestingly similar or different. In contrast to techniques that completely match patterns that directly indicate violations of expected patterns (e.g. as in the approach presented in this dissertation), these techniques produce output to the effect, “the source sequence is similar to a target sequence with a correspondence measure of 61%”, leaving it up to investigators to decide on a case to case basis what exactly the correspondence measure means.

A key limitation of any technique comparing sequences against abstract models (e.g. EMA, USINE, ErgoLight EUVS, and the process validation techniques described by Cook and Wolf) is that in order to reliably categorize a source sequence as being a poor match, the model used to perform the comparison must be relatively complete in its ability to describe all possible, or rather, expected paths. This is all but impossible in most non-trivial interfaces. Furthermore, the model must somehow deal with “noise” so that

low level events, such as mouse movements, won't mask otherwise significant correspondence between source sequences and the abstract model. Because these techniques typically have no built-in facilities for performing transformations on input traces, this implies that either the event stream has already been transformed, perhaps by manually instrumenting the application (as with EMA), or complexity must be introduced into the model to avoid sensitivity to noise. In contrast, techniques such as EBBA and the work presented in this dissertation use selection and abstraction to pick out patterns of interest from the noise. The models need not be complete in any sense, and may ignore events that are not of interest.

4.7 Sequence Characterization

4.7.1 Purpose

These techniques take “source” sequences as input and attempt to construct an abstract model to summarize, or characterize, interesting sequential features of those sequences. Some techniques produce a process model with probabilities associated with transitions (e.g., [Guzdial 1993]). Others may be used to construct models that characterize the grammatical structure of events in the input sequences (e.g., [Olson et al. 1993]).

4.7.2 Examples

Guzdial [Guzdial 1993] describes a technique based on Markov Chain analysis that can be used to produce process models with probabilities assigned to transitions to characterize user behavior with interactive applications. First, abstract stages, or states, of

application use are identified. In Guzdial's example, a simple design environment was the object of study. Facilities provided by the design tool supported the following stages in a design process: "initial review", "decomposition", "composition", "debugging", and "final review". Each of the operations in the interface could be mapped to one or another of these abstract design stages. For instance, all debugging related commands (which incidentally all appeared in a single "debugging" menu) could be mapped to the "debugging" stage. The stream of events was then abstracted and recoded to replace low level events with the abstract stages associated with them (presumably dropping events not associated with stages). The observed probability of entering any stage from the stage immediately before it was then computed to yield a transition matrix. The matrix can then be used to create a process diagram with probabilities associated with transitions. For instance, one subject was observed to have transitioned from "debugging" to "composition" more often (52% of all transitions out of "debugging") than to "decomposition" (10%) (Figure 4-15). A steady state vector was then computed to reflect the probability of any event chosen at random belonging to each particular stage. This could then be compared to an expected probability vector computed by simply calculating the percentage of commands associated with each stage. This comparison could then be used to indicate user "preference" for classes of commands.

Olson and colleagues [Olson et al. 1993] describe an approach based on statistical and grammatical techniques for characterizing the sequential structure of verbal interactions between participants in design meetings. Verbalizations were encoded into categories and subjected to statistical techniques, including log linear modeling and lag

sequential analysis, to identify potential dependencies between events. These patterns were then used to suggest rules that might be included in a definite clause grammar used to summarize, or characterize part of the sequential structure of the meeting interactions. These grammar rules were then used to rewrite some of the patterns embedded in the sequence (abstraction and recoding). The sequence was again subjected to statistical techniques and the grammar refined further in an iterative fashion. The result was a set of grammar rules that provided insight into the sequential structure of the meeting interactions.

4.7.3 Related

Software process discovery techniques [Cook and Wolf 1996] are related in that they attempt to automatically generate a process model, in the form of a finite state machine, that accounts for a trace of events produced by a particular software process. It is not clear how well these techniques would perform with data as noisy as UI events. A more promising approach might be to perform selection, abstraction, and recoding on the event stream before submitting it for analysis.

4.7.4 Strengths

The strength of these techniques lies in their ability to provide investigators with insight into the sequential structure of events embedded within sequences.

4.7.5 Limitations

The technique described by Olson and colleagues requires extensive human involvement and can be very time-consuming. On the other hand, the automated

techniques suggested by Cook and Wolf appear to be sensitive to noise, and are less likely to produce models that make sense to investigators [Olson et al. 1994].

Markov-based models, while relying on overly-simplifying assumptions, appear to be more likely than grammar-based techniques to tell investigators something useful about user interactions. Investigators often have an idea of the grammatical structure of interactions that may arise from the use of (at least portions of) a particular interface. Grammars are thus useful in transforming low level UI events into higher level events of interest, or to detect when actual usage patterns violate expected patterns. However, the value of generating grammatical or finite-state-machine-based models after the fact, is more limited. More often than not, a grammar- or finite-state-machine-based model generated on the basis of multiple traces will be vacuous in that it will describe all observed patterns of usage of an interface without indicating which are most common. While this may be useful in defining paths for UI regression testing, investigators interested in locating usability issues will be more interested in seeing and determining the frequency of specific observed patterns than in seeing a grammar to summarize them all.

4.8 Visualization

4.8.1 Purpose

These techniques present the results of transformations and analyses in forms allowing humans to exploit their innate visual analysis capabilities to interpret results.

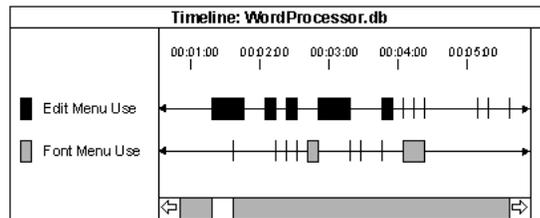


Figure 4-9. Visualizing the results of event selection. Use of “Edit” menu operations is indicated in black. Use of “Font” menu operations is indicated in grey.

Some of these techniques are helpful in linking results of analysis back to features of the interface.

4.8.2 Examples

A number of techniques have been used to visualize data based on UI events. For a survey of such techniques see [Guzdial et al. 1994]. Below are a few examples of techniques that have been used in support of the analysis approaches described above.

4.8.2.1 Transformation

The results of performing selection or abstraction on an event stream can be visualized using a timeline in which bands of colors indicate different selections of events in the event stream. For example, one might use red to highlight the use of “Edit” menu operations and blue to highlight the use of “Font” menu operations in the evaluation of a word processor (Figure 4-9).

OrderForm.db		
UI Events	Abs. Interaction Events	Task-Related. Events
⋮	⋮	⋮
GotFocus(Name)		
Key(Name, 'D')	GotEdit(Name)	AddressSectionStarted()
Key(Name, 'a')		
Key(Name, 'v')		
Key(Name, 'i')		
Key(Name, 'd')		
LostFocus(Name)		
GotFocus(Street)		
	LostEdit(Name)	
Key(Street, '1')	GotEdit(Street)	
	ValueProvided(Name, "David")	NameProvided("David")
Key(Street, '3')		
Key(Street, '8')		
⋮		
⋮	⋮	⋮
LostFocus(ZIP)		
GotFocus(Quantity)		
	LostEdit(ZIP)	
	GotEdit(Quantity)	
Key(Quantity, '1')	ValueProvided(ZIP, "90740")	ZIPProvided("90740")
		AddressSectionCompleted()

Figure 4-10. Visualizing the results of event abstraction. Correspondence between events at different levels of abstraction are indicated by horizontal alignment. Single “Key” events in large cells correspond to “LOST_EDIT”, “GOT_EDIT”, and “VALUE_PROVIDED” abstract interaction events in smaller, horizontally aligned cells.

MacSHAPA [Sanderson and Fisher 1994] visualizes events as occupying cells in a spreadsheet. Event streams are listed vertically (in adjacent columns) and correspondence of events in one stream with events in adjacent streams is indicated by horizontal alignment (across rows). A large cell in one column may correspond to a number of smaller cells in another column to indicate abstraction relationships (Figure 4-10).

4.8.2.2 Counts and summary statistics

There are a number of visualizations that may be used to represent the results of counts and summary statistics, including static 2D and 3D graphs, static 2D effects superimposed on a coordinate space representing the interface, and static and dynamic 2D and 3D effects superimposed on top of an actual visual representation of the interface.

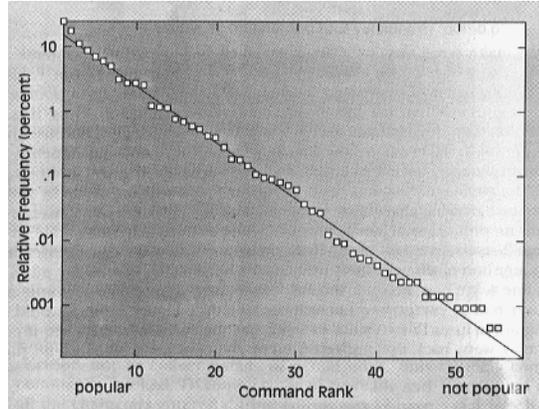


Figure 4-11. Relative command frequencies ordered by “rank”

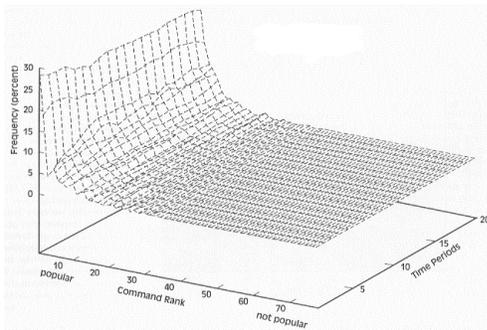


Figure 4-12. Relative command frequencies over time

The following are examples of static 2D and 3D graphs:

- Graph of keystrokes per window [Chang and Dillon 1997].
- Graph of mouse clicks per window [Chang and Dillon 1997].
- Graph of relative command frequencies [Kay and Thomas 1995] (Figure 4-11).
- Graph of relative command frequencies as they vary over time [Kay and Thomas 1995] (Figure 4-12).

The following are examples of static 2D effects superimposed on an abstract coordinate space representing the interface:

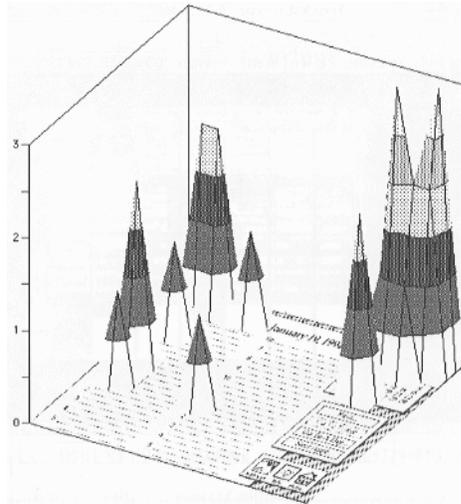


Figure 4-13. Mouse click location and density. A 3D representation of mouse click location and density is superimposed over a graphical representation of the interface.

- Location of mouse clicks [Guzdial et al. 1994; Chang and Dillon 1997].
- Mouse travel patterns between clicks [Buxton et al. 1983; Chang and Dillon 1997].

The following are examples of static and dynamic 2D and 3D effects superimposed on top of a graphical representation of the interface:

- Static highlighting to indicate location and density of mouse clicks [Guzdial et al. 1994].
- Dynamic highlighting of mouse click activity as it varies over time [Guzdial et al. 1994].
- 3D representation of mouse click location and density [Guzdial et al. 1994] (Figure 4-13).

Meeting.db		
	MeetingA	MeetingB
	Amplify	Amplify
	Resolve Issue	Resolve Issue
	Identify Problem	Identify Problem
	Amplify	Digress
	Digress	Amplify
	Recapitulate	Recapitulate
	Identify Issue	Identify Issue
Alignment: Meeting.db		
	MeetingA	MeetingB
Amplify	Amplify	Amplify
Resolve	Resolve Issue	Resolve Issue
Identify	Identify Problem	Identify Problem
Amplify	Amplify	
Digress	Digress	Digress
Amplify	Digress	Digress
Resolve	Recapitulate	Amplify
Identify	Recapitulate	Recapitulate
Resolve	Identify Issue	Identify Issue
Identify		Digress
Resolve	Amplify	Amplify
Identify	Resolve Issue	
Amplify	Identify Issue	Identify Issue
Recapit		Resolve Issue
Resolve	Amplify	Amplify
	Digress	
		Identify Issue

Figure 4-14. Visualizing the results of automated sequence alignment. Horizontal alignment indicates correspondence. Black spaces indicate where alignment was not possible.

4.8.2.3 Sequence detection

The results of selecting subsequences of UI events based on sequence detection techniques can be visualized using the same technique illustrated in (Figure 4-9).

4.8.2.4 Sequence comparison

As described above, MacSHAPA [Sanderson and Fisher 1994] provides facilities for aligning two sequences of events as optimally as possible and presenting the results visually as cells in adjacent spreadsheet columns. Aligned events appear in the same row and missing cells indicate events in one sequence that could not be aligned with events in the other (Figure 4-14).

UsAGE [Ueling and Wolf 1995] provides a similar visualization for comparing sequences based on drawing a connected graph of nodes. The “expert” series of actions is

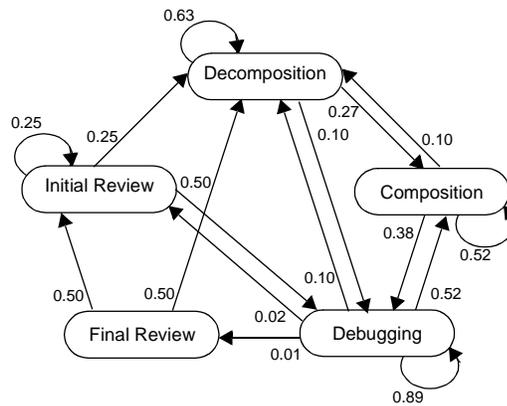


Figure 4-15. A process model characterizing user behavior. Nodes represent process steps and arcs indicate observed probabilities of transitions between process steps.

displayed linearly as a sequence of nodes across the top of the graph. The “novice” series of actions are indicated by drawing directed arcs connecting the nodes to represent the order in which the actions were performed by the novice. Out of sequence actions are indicated by arcs that skip expert nodes in the forward direction, or that point backwards in the graph. Unmatched actions taken by the novice appear as nodes (with a different color) placed below the last matched expert node.

4.8.2.5 Sequence characterization

Guzdial [Guzdial 1993] uses a connected graph visualization of the results of his Markov-based analysis. The result is a process model with nodes representing process steps and arcs indicating the observed probabilities of transitions between process steps (Figure 4-15).

4.8.3 Strengths

The strengths of these techniques lie in their ability to represent the results of analysis in forms allowing humans to exploit their innate visual analysis capabilities to

interpret results. Particularly useful are the techniques that link results of analysis back to features of the interface, such as the techniques superimposing graphical representations of behavior over actual representations of the interface.

4.8.4 Limitations

With the exception of simple graphs (which can typically be generated using standard plotting facilities provided by spreadsheets and statistical analysis packages), most of the visualizations above must be produced by hand. Techniques for accurately superimposing graphical effects over visual representations of the interface can be particularly problematic.

4.9 Integrated Support

4.9.1 Purpose

Integrated support is provided by environments that facilitate flexible composition of various transformation, analysis, and visualization capabilities. Some environments also provide built-in support for managing domain-specific artifacts such as evaluations, subjects, tasks, data and results of analysis.

4.9.2 Examples

MacSHAPA [Sanderson et al. 1994] is perhaps the most comprehensive environment designed to support all manner of exploratory sequential data analysis (ESDA). Features include: data import and export; video and coded observation synchronization and searching capabilities; facilities for performing selection,

abstraction, and recoding; a number of built-in counts and summary statistics; features supporting sequence detection, comparison, and characterization; a general purpose database query and manipulation language; and a number of built-in visualizations and reports.

DRUM [Macleod et al. 1993] provides integrated features for synchronizing events, observations, and video; for defining and managing observation coding schemes; for calculating pre-defined counts and summary statistics; and for managing and manipulating evaluation-related artifacts regarding subjects, tasks, recording plans, logs, videos, and results of analysis.

Hawk [Guzdial 1993] provides flexible support for creating, debugging, and executing scripts that can be used to automatically select, abstract, and recode events in recorded log files. Management facilities are also provided to organize and store event logs and analysis scripts.

Finally, ErgoLight Operation Recording Suite (EORS) and Usability Validation Suite (EUVS) [ErgoLight Usability Software 1998] offer a number of facilities for managing usability evaluations, both local and remote, as well as facilities for merging data from multiple users.

4.9.3 Strengths

The task of extracting usability-related information from UI events typically requires the management of numerous files and media types as well as the creation and

composition of various analysis techniques. Environments supporting the integration of such activities can significantly reduce the burden of data management and integration.

4.9.4 Limitations

All of the above environments possess important limitations.

While MacSHAPA is perhaps the most comprehensive integrated environment for analyzing sequential data, it is not specifically designed for analysis of UI events. As a result, it lacks support for event capture and focuses primarily on analysis techniques that, when applied to UI events, require extensive human intervention and interpretation to extract useful information. MacSHAPA provides many of the basic building blocks required for an “ideal” environment for capturing and analyzing UI events, however, it requires too much human effort to perform selection, abstraction, and recoding that should be performed in a more automated fashion. Furthermore, because the powerful features of MacSHAPA cannot be used *during* event collection, contextual information that might be useful in selection and abstraction is not available.

While providing features for managing and analyzing UI events, coded observations, video data, and evaluation artifacts, DRUM provides no features for selecting, abstracting, and recoding data.

Finally, while Hawk addresses the problem of providing automated support for selection, abstraction, and recoding, like MacSHAPA, it does not address event capture, and as a result, contextual information cannot be used in selection and abstraction.

4.10 Summary of the State of the Art

Synchronization and searching techniques are among the most mature technologies for exploiting user interface event data in support of usage and usability evaluations. Tools supporting these techniques are becoming increasingly common in usability labs. However, these techniques can be costly in terms of equipment, human observers, and data storage and analysis requirements. Furthermore, synchronization and searching techniques generally exploit user interface events as no more than convenient indices into video recordings. In some cases, events may be used as the basis for computing simple counts and summary statistics using spreadsheets or statistical packages. However, such analyses typically require selection and abstraction to be performed by hand prior to analysis to obtain meaningful results.

The other, arguably more sophisticated, analysis techniques such as sequence detection, comparison, and characterization continue to remain denizens of the research lab for the most part. Those that are most compelling tend to require the most human intervention, interpretation, and effort (e.g., exploratory sequential data analysis techniques and the Markov- and Grammar-based sequence characterization techniques). Those that are most automated tend to be least compelling and most unrealistic in their assumptions (e.g., ADAM, UsAGE, and EMA). One of the main problems limiting the success of automated analysis techniques may be their lack of focus on selection, abstraction, and recoding which appear to be preconditions to meaningful analysis.

Nevertheless, few investigators have attempted to address the problem of transformation realistically. Of the over twenty-five approaches surveyed here, only a handful provide mechanisms that allow investigators to transform event streams. Of those, fewer still allow models to be constructed and reused so that transformation can be automated. Of those, nearly none allow transformation to be performed in-context so that important contextual information can be used in selection and abstraction.

There are a number of ways that investigators have attempted to side-step the transformation problem. For instance, building data collection directly into a user interface management system or requiring applications to report events themselves can help ameliorate some of the issues. However, both of these approaches have important limitations.

User interface management systems (UIMS's) typically model the relationships between application features and UI events explicitly, so reasonable data collection and analysis mechanisms can be built directly in, as in the case of the MIKE UIMS [Olsen and Halversen 1988], KRI/AG [Lowgren, J. and Nordqvist 1992], and UsAGE [Ueling and Wolf 1995]. Because UIMS's have dynamic access to most aspects of the user interface, contextual information useful in interpreting the significance of events is also available. However, many developers do not use UIMS's, thus, a more general technique that does not presuppose the use of a UIMS is required.

Techniques that allow applications to report events directly via an event-reporting API provide a useful service, particularly in cases where events of interest cannot be

inferred from UI events. This allows important application-specific events to be reported by applications themselves and provides a more general solution than a UIMS-based approach. However, this places a burden on application developers to capture and transform events of interest, for example, as in [Kay and Thomas 1995; Balbo 1996]. This can be costly, particularly if there is no centralized command dispatch loop, or similar mechanism, that can be tapped as a source of application events. This also complicates evolution since data collection code is typically intermingled with application code. Furthermore, there is much useful usage- and usability-related information not typically processed by applications that can be easily captured by tapping into the user interface event stream, for example, shifts in input focus, mouse movements, and the specific user interface actions used to invoke application features. As a result, an event-reporting API is just part of a more comprehensive solution.

4.10.1 Challenges and Future Directions

More research is needed in the area of transformation *during* data collection to ensure that useful information can be captured in the first place before automated analysis techniques, such as those described above, can yield meaningful results. A reasonable approach should assume no more than a typical event-based user interface system, such as provided by the Macintosh Operating System, Microsoft Windows, X Window System, or Java Abstract Window Toolkit. Developers should not be required to adopt a particular UIMS nor call an API to report every potentially interesting event.

Furthermore, in order to support *large-scale* collection of *high-quality* data, the approach must also address each of the following problems:

- *The abstraction problem:* Questions about usage typically occur in terms of concepts at higher levels of abstraction than represented in user interface event data. Furthermore, questions of usage may occur at multiple levels of abstraction. This implies the need for “data abstraction” mechanisms to allow low-level data to be related to higher-level concepts such as user interface and application features as well as users’ tasks and goals.
- *The selection problem:* The amount of data necessary to answer usage questions is typically a small subset of the much larger set of data that *might* be captured at any given time. Failure to properly select data of interest increases the amount of data that must be reported and decreases the likelihood that automated analysis techniques will identify events and patterns of interest in the “noise”. This implies the need for “data selection” mechanisms to allow necessary data to be captured, and unnecessary data filtered, prior to reporting and analysis.
- *The reduction problem:* Much of the analysis that will ultimately be performed to answer usage questions can actually be performed *during* data collection resulting in greatly reduced data reporting and post-hoc analysis needs. Performing reduction as part of the capture process not only decreases the amount of data that must be reported, but also increases the likelihood that successful analysis will actually be performed. This implies the need for “data reduction” mechanisms to reduce the amount of data that must ultimately be reported and analyzed.
- *The context problem:* Potentially critical information necessary in interpreting the significance of events is often not readily available in event data alone. Such information may be spread across multiple events or missing altogether, but is often available “for the asking” from the user interface, application, artifacts, or user. This implies the need for “context-capture” mechanisms to allow important user interface, application, artifact, and user state information to be used in data abstraction, selection, and reduction.
- *The evolution problem:* Finally, data collection needs typically evolve over time (perhaps due to results of earlier data collection) more rapidly than do applications. Unnecessary coupling of data collection and application code increases the cost of evolution and impact on users. This implies the need for “independently evolvable” data collection mechanisms to allow in-context data abstraction, selection, and reduction to evolve over time without impacting application deployment or use.

CHAPTER 5: Approach

This chapter is divided into three parts. First, a brief example is presented to illustrate how software agents might be used to capture information regarding the use of a simple word processing application. Next, fundamental technical details underlying the implementation of the approach are presented to provide a foundation for understanding the solutions presented in the following chapter. Finally, a usage scenario is presented to illustrate how the approach might be applied in practice.

5.1 Example

The approach presented here involves a development platform for creating software agents that are deployed over the Internet to observe application use and report usage data and user feedback to developers to help improve the fit between application design and use. The following example illustrates how agents might be used to capture information regarding the use of a simple word processing application. The word processing application has a standard menu, toolbar, and dialog-based user interface:

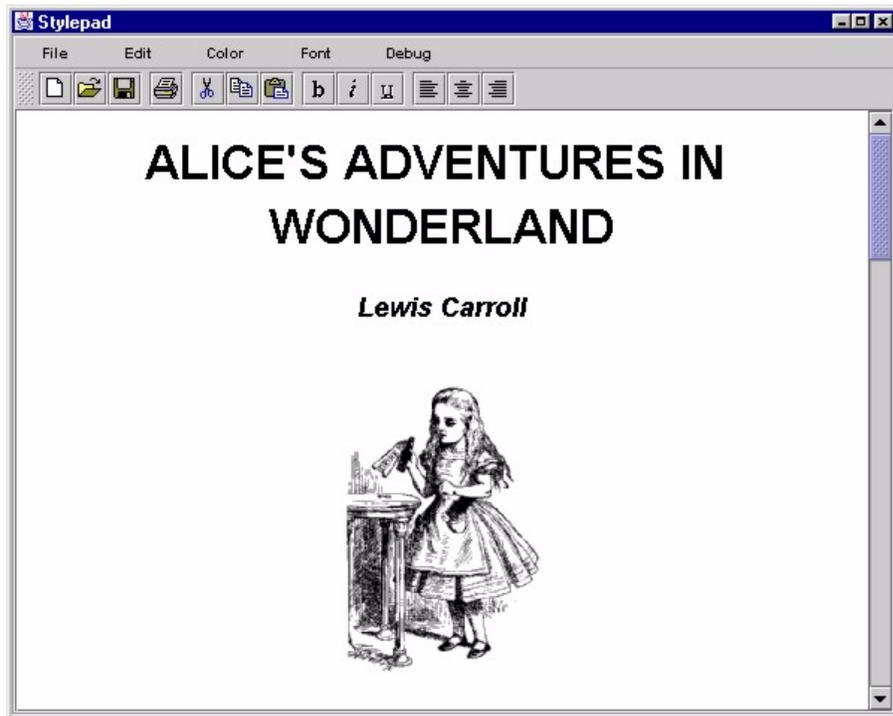


Figure 5-1. A simple word processing application

The approach proposed in this research allows agents to be defined to observe and capture data regarding particular user interactions of interest. Consider the following agent which captures data regarding the use of the “File Menu”:

Begin	Trigger	USE Window/Stylepad/MenuItem/New * OR USE Window/Stylepad/MenuItem/Open * OR USE Window/Stylepad/MenuItem/Save * OR USE Window/Stylepad/MenuItem/Print * OR USE Window/Stylepad/MenuItem/Exit *
	Action	PostEvent("MENU", AgentSource)
	Data	RecordEventData()

Figure 5-2. “File Menu” agent

The technical details underlying agent specifications and behavior are presented in more detail the following section. For the purposes of this example, suffice it to say that this agent: (1) is triggered whenever any of the items in the “File Menu” is used, (2) responds by generating an abstract “MENU” event (with the agent itself as the event source) to indicate that the “File Menu” has been used, and (3) records data regarding the events triggering the agent. The following screenshot illustrates the sort of data captured by this agent, namely, the number of times each File Menu item has been used:

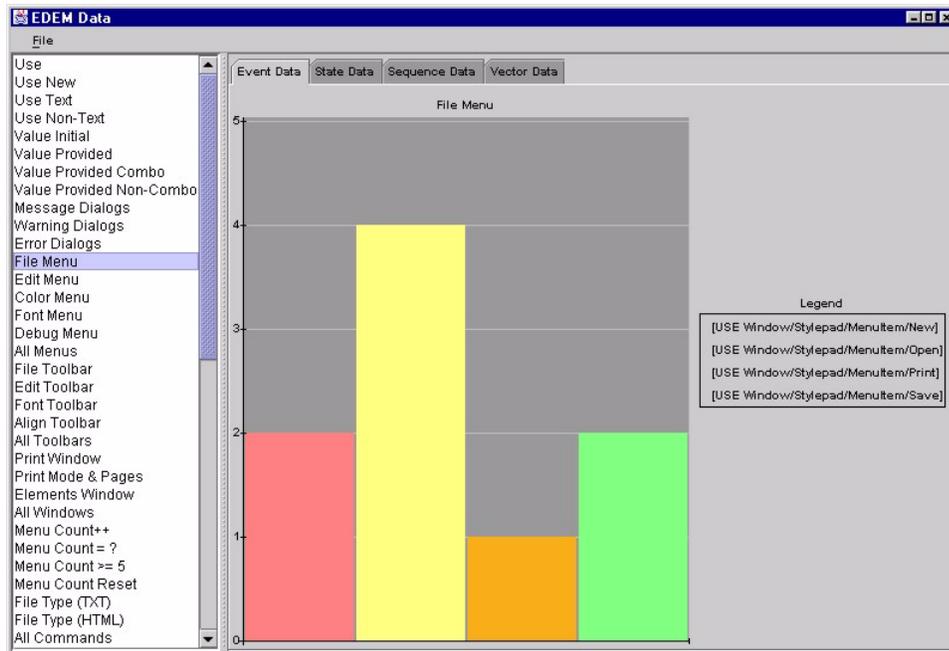


Figure 5-3. “File Menu” data

Similar agents can then be defined to perform analogous actions for each of the remaining menus in the interface. For instance, an “Edit Menu” agent can be defined that: (1) is triggered whenever any of the items in the “Edit Menu” is used, (2) responds by generating an abstract “MENU” event (with the agent itself as the event source) to

indicate that the “Edit Menu” has been used, and (3) records data regarding the events triggering the agent.

Once agents have been defined to capture data regarding each of the application’s menus, an agent of the following sort can be defined to capture data regarding the use of all menus:

Begin	Trigger	MENU * *
	Data	RecordEventData()

Figure 5-4. “All Menus” agent

This agent is activated whenever any “MENU” event is observed coming from any event source, and records data regarding the events triggering the agent. The following screenshot illustrates the sort of data captured by this agent, namely, the number of times each application menu has been used:



Figure 5-5. “All Menus” data

Similar agents can then be defined to capture and characterize information regarding the use of all toolbars, dialogs, and application commands, as well as other more specialized data. The following section describes the fundamental technical concepts underlying the implementation of this approach.

5.2 Implementation

5.2.1 Basic Services

The fundamental strategy underlying this work is to exploit already existing information produced by user interface and application components to support usage data collection. To this end, an *event service* — providing generic event and state monitoring

capabilities — was implemented, on top of which an *agent service* — providing generic data abstraction, selection, and reduction services — was implemented.

However, because the means for accessing event and state information can vary depending on the components used to develop applications, the notion of a *default model* was introduced to mediate between monitored components and the event service.⁶ Furthermore, because agents are often reusable, or at least adaptable, across applications, the notion of *default agents* was introduced to allow higher-level generic data collection services to be reused across applications.

The following diagram illustrates the layered relationship between the application, default model, event service, agent service, default agents, and user-defined agents. The shading indicates the degree to which each aspect is believed to be generic and reusable:

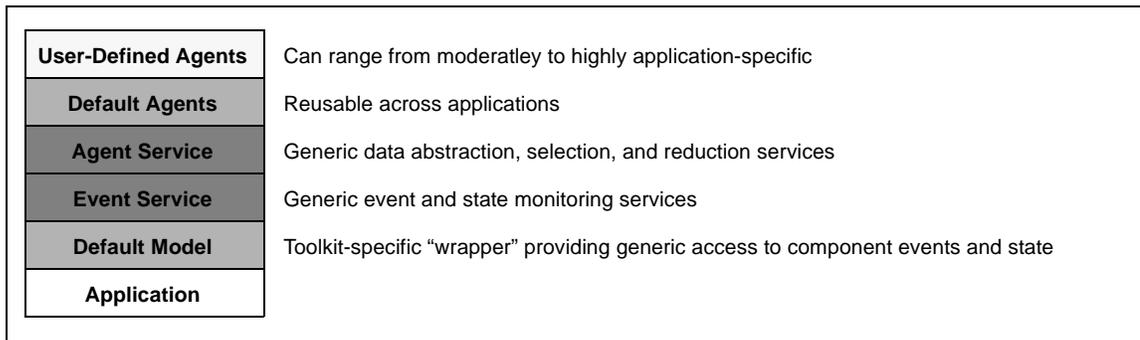


Figure 5-6. Basic services

6. The default model in this case provides generic access to Java’s standard GUI component toolkit. The default model can be adapted as new GUI toolkits are introduced, or alternatively, a more generic model for accessing arbitrary component events and state, based on a software component standard such as JavaBeansTM, might be implemented.

All of the basic monitoring and data collection services needed to perform usage data collection may be invoked via the following API:

Table 5-1: Basic Services API

API Method	Description
void begin()	to start monitoring and data collection
void end()	to end monitoring and data collection and initiate reporting
void setName(Object source, String name)	to uniquely name components of particular interest that could not be uniquely named by the default model

Typically, only begin() and end() will need to be invoked. The agent authoring user interface provided by the agent service may then be used to define application-specific agents to perform in-context data abstraction, selection, and reduction as needed.

5.2.2 Event Service

The event service provides generic component event and state monitoring capabilities.

Components are identified by name. The default model attempts to generate unique names for all user-manipulable components based on component label, type, and position in the application window hierarchy. Components of particular interest that cannot be uniquely named by the default model can be named by the developer using the setName() method as mentioned above.

Event-related operations include publish(), subscribe(), unsubscribe(), and post() in which events are identified using “event specs”. Event specs consist of an event name, source component name, and source component type separated by ‘|’ characters.

Typically one or more of these attributes is “wildcarded” using the ‘*’ character as illustrated below:

Table 5-2: Event Specs

Event Spec	Description
MOUSE_CLICKED Window/Print/OK *	matches all mouse clicks in the “Print” window “OK” button component
MOUSE_CLICKED * javax.swing.JButton	matches all mouse clicks in components of class “javax.swing.JButton”
MOUSE_CLICKED * *	matches all mouse clicks
* Window/Print/OK *	matches all events in the “Print” window “OK” button component
* * javax.swing.JButton	matches all events in components of class “javax.swing.JButton”
* * *	matches all events

State-related operations include getState() and setState() in which components and global variables are identified by name. The getState() method may be used to fetch the current value of the specified component from the default model, and the getState() and setState() methods may be used to fetch and store values in global variables. Below is the complete event service API:

Table 5-3: Event Service API

API Method	Description
void publish(String eventName)	to make events available for subscription
void subscribe(Subscriber subscriber, String eventSpec)	to register interest in events matching event spec
void unsubscribe(Subscriber subscriber, String eventSpec)	to un-register interest in events matching event spec
void post(Object sourceId, String eventName)	to create and dispatch an event to interested parties
String getState(String key)	to retrieve global or component state
void setState(String key, String value)	to store global state

Event dispatching is achieved using a hashtable in which entries correspond to unique event specs for which there are one or more subscribers. Event specs are stored as keys and lists of subscribers are stored as values. For each posted event, subscriber lists for the following event specs are retrieved and subscribers notified in subscription order:

```
EVENT_NAME|SOURCE_NAME|*  
EVENT_NAME|*|SOURCE_CLASS  
*|SOURCE_NAME|*  
*|*|SOURCE_CLASS  
*|*|*
```

Because event dispatch code is executed every time an event occurs, the algorithm is kept simple, and only five hashtable lookups are performed per event, regardless of the number of components, events, or outstanding subscriptions.

Typically, application developers will not directly call the event service API. The agent-authoring interface provided by the agent service provides access to all of this functionality. However, if a developer wishes to generate special-purpose application-specific events or store special-purpose application-specific state from within application code, `publish()` may be called to make events available for agent subscriptions, `post()` may be called to dispatch events to interested agents, and `setState()` may be used to store global state for retrieval by agents.

5.2.3 Agent Service

The agent service provides generic data abstraction, selection, and reduction capabilities and a graphical user interface for authoring agents.

Agents come in two varieties: single-triggered and dual-triggered. The following figure illustrates the structure of an agent with a single trigger:

Begin	Trigger	Disjunct, Conjunct, or Sequence of Event Specs (evaluated for each Event Spec match)
	Trigger Guard	SourceIsNew, SourceInSet, SourceNotInSet, SourceClassInSet, or SourceClassNotInSet (evaluated for event satisfying Trigger)
	Guard	Disjunct or Conjunct of State Check's (evaluated after Trigger & TriggerGuard satisfaction)
	Action	Post Event and one or more State Update's (performed after Trigger, TriggerGuard, & Gaurd satisfaction)
	Data	RecordEventData, RecordEventTransitionData, RecordEventSequenceData, RecordStateData, RecordStateVectorData, RecordStateDataPerEvent, RecordUserData (recorded just before Action performed)

Figure 5-7. A single-triggered agent

The following figure illustrates the structure of an agent with a dual trigger:

Begin	Trigger	Same as above
	Trigger Guard	Same as above
	Guard	Same as above
	Action	Same as above
	Data	RecordEventData, RecordEventTransitionData, RecordEventSequenceData, RecordStateDataPerEvent (Event-related data recorded between Begin and End Trigger, TriggerGuard, & Gaurd satisfaction)
End	Trigger	Same as above
	Trigger Guard	Same as above
	Guard	Same as above
	Action	Same as above
	Data	RecordStateData, RecordStateVectorData, RecordUserData (State- and user-related data recorded just before End Action performed)

Figure 5-8. A dual-triggered agent

Trigger specifications include an event composition operator (Disjunction, Conjunction, or Sequence) and a list of event specs. For instance, a trigger of the form “A or B” fires when an event satisfying either event spec A or event spec B has occurred. “A and B” fires when events satisfying both event specs A and B have occurred. “A then B” fires when events satisfying both event specs A and B have occurred in the specified order.

TriggerGuard specifications include an event source predicate (SourceIsNew, SourceInSet, SourceNotInSet, SourceClassInSet, or SourceClassNotInSet) and an

optional list of source names or classes. Thus, a trigger may be constrained to fire only if the event source has changed since the last time the trigger fired, or if the event source or source class is included, or not included, in a pre-specified set of source components or classes. This might be used to detect when keyboard activity has shifted from one component to the next, or to detect when general “use” activity has shifted from one *group* of components to another.

Guard specifications includes a list of state check specifications which are boolean expressions involving component and global state variables and constants and a comparison predicate (Empty, Contains, Starts w/, Ends w/, ==, <, >, <=, or >=). For instance, an agent may be defined to take action only if the value of a particular component is “empty”, or if the value of a global “mode” variable is “true”, or the value of a global “counter” variable is greater than some pre-specified threshold.

An *Action* specification includes an abstract event specification consisting of an event name and source specification (No Source, Trigger Source, or Agent Source), and a list of state update specifications which are expressions involving component and global state variables and constants and an update operator (Assignment, Increment, or Decrement). For instance, an agent may be defined to associate abstract “MENU” events with every menu item that is invoked in which the agent itself (named after the menu in question) is stored as the event source. Agents may also assign constant values to global “mode” variables and/or increment or decrement global “counter” variables based on event occurrences of interest.

A *Data* specification is composed of *EventData*, *StateData*, and *UserData* specifications:

- An *EventData* specification includes a reduction specification (Events, Transitions, or Sequences) and an optional list of event specs to be recorded in addition to Trigger events. For instance, an agent with a dual trigger might be used to record all “VALUE_PROVIDED” events occurring between the time that a dialog is opened (the Begin Trigger) and the time it is closed (the End Trigger). Furthermore, event data may be recorded in terms of simple counts of individual events, event transitions, or whole sequences of events (segmented by Begin and End Triggers).
- A *StateData* specification includes a reduction specification (Values, Vectors, or Append to Event Data) and a list of component and global state variables. For instance, the values associated with each of the controls in a dialog might be recorded when the dialog is closed. Furthermore, value data may be reported in terms of simple counts of individual values, or vectors of values so that co-occurrence of values can be analyzed. Finally, value data can be appended to event data so that events may be analyzed based on state information, such as the document type being edited when an event occurred or the number of menus opened before a particular menu item was selected.
- A *UserData* specification includes a notification type (None, Non-Intrusive, or Intrusive), a notification header, and a notification message. Non-intrusive notifications involve placing a copy of the notification header in a list of outstanding notifications. The user may then request more information to display the notification message. Finally, the user may provide feedback if desired. Intrusive notification involves immediately posting a user feedback dialog containing the notification message for immediate user attention. These options allow agents to be defined to interact with users when features of interest are used, or when unexpected patterns of behavior are observed so that users may learn more about expected use and potentially provide feedback to help developers refine expectations.

The algorithm governing agent execution can be summarized as follows:

```

IF (Satisfied BeginTrigger)
  IF (IsTrue BeginTriggerGuard & IsTrue BeginGuard)
    IF (Enabled EventData) BeginRecording EventData // record begin time
    IF (IsSingleTriggered Agent)
      IF (Enabled EventData) EndRecording EventData // record event that satisfied this trigger; elapsed time = 0
      IF (Enabled StateData) Record StateData // record state
      IF (Enabled UserData) Record UserData // notify user; user may provide feedback
      IF (Enabled BeginAction) Perform BeginAction // generate abstract event and/or update global state
    ELSE IF (IsDualTriggered Agent)
      IF (Enabled EventData) Record EventData // record event that satisfied this trigger
      IF (Enabled BeginAction) Perform BeginAction // generate abstract event and/or update global state
      Disable BeginTrigger // disable begin trigger (until end trigger satisfied)
      Enable DataTrigger // enable data trigger (until end trigger satisfied)
      Enable EndTrigger // enable end trigger (until end trigger satisfied)
      Pass event to EndTrigger // event that satisfied this trigger may also help satisfy end trigger

IF (Satisfied DataTrigger)
  IF (Enabled EventData) Record EventData // record event that satisfied this trigger

IF (Satisfied EndTrigger)
  IF (IsTrue EndTriggerGuard & IsTrue EndGuard)
    IF (Enabled EventData) EndRecording EventData // record event that satisfied this trigger; elapsed time = end - beg
    IF (Enabled StateData) Record StateData // record state
    IF (Enabled UserData) Record UserData // notify user; user may provide feedback
    IF (Enabled EndAction) Perform EndAction // generate abstract event and/or update global state
  Enable BeginTrigger // reset begin trigger
  Disable DataTrigger // reset data trigger
  Disable EndTrigger // reset end trigger
  
```

Figure 5-9. Agent algorithm

A usage scenario is presented below to illustrate how the proposed approach might be applied in practice. If the concepts presented in this chapter are not clear after reading the scenario, the examples in the following chapter may help clarify matters.

5.3 Usage Scenario

To see how these services might be put to use by developers in practice, consider the following scenario which is adapted from a demonstration performed by Lockheed

Martin C2 Integration Systems within the context of a large-scale, governmental logistics and transportation information system.

A group of engineers are tasked with designing a web-based user interface to provide end users with access to a large store of transportation-related information. The interface in this scenario is modeled after an existing interface (originally developed in HTML and JavaScript) that allows users to request information regarding Department of Defense cargo in transit between points of embarkation and debarkation. For instance, military officers might use the interface to determine the current location of munitions being shipped to U.S. troops in Bosnia.

This is an example of an interface that might be used repeatedly by a number of users in completing their work. It is important that interfaces supporting frequently performed tasks (such as steps in a business process or workflow) are well-suited to users' tasks, and that users are aware of how to most efficiently use them, since inefficiencies and mistakes can add up over time.

After involving users in design, constructing use cases, performing task analyses, doing cognitive walkthroughs, and employing other user-centered design methods, a prototype implementation of the interface is ready for deployment. Figure 5-10 shows the prototype interface.

Figure 5-10. A prototype database query interface

The engineers in this scenario were particularly interested in verifying the expectation that users would not frequently change the “mode of travel” selection in the first section of the form (e.g. “Air”, “Ocean”, “Motor”, “Rail”, or “Any”) after having made subsequent selections, since the “mode of travel” selection affects the choices that are available in subsequent sections. Operating under the expectation that this would not be a common source of problems, the engineers made the design decision to simply reset all selections to their default values whenever the “mode of travel” selection is reselected.

Figure 5-11 is a screenshot of the agent authoring interface provided by the agent service to allow agents to be defined without writing code. In Figure 5-11, the developer has defined an agent that “fires” whenever the user uses one of the controls in the “mode of travel” section of the interface and then uses controls outside of that section. This agent is then used in conjunction with other agents to detect when the user changes the mode of travel after having made subsequent selections. Other agents were also defined to record transitions between sections and whole sequences of section use in order to verify that the

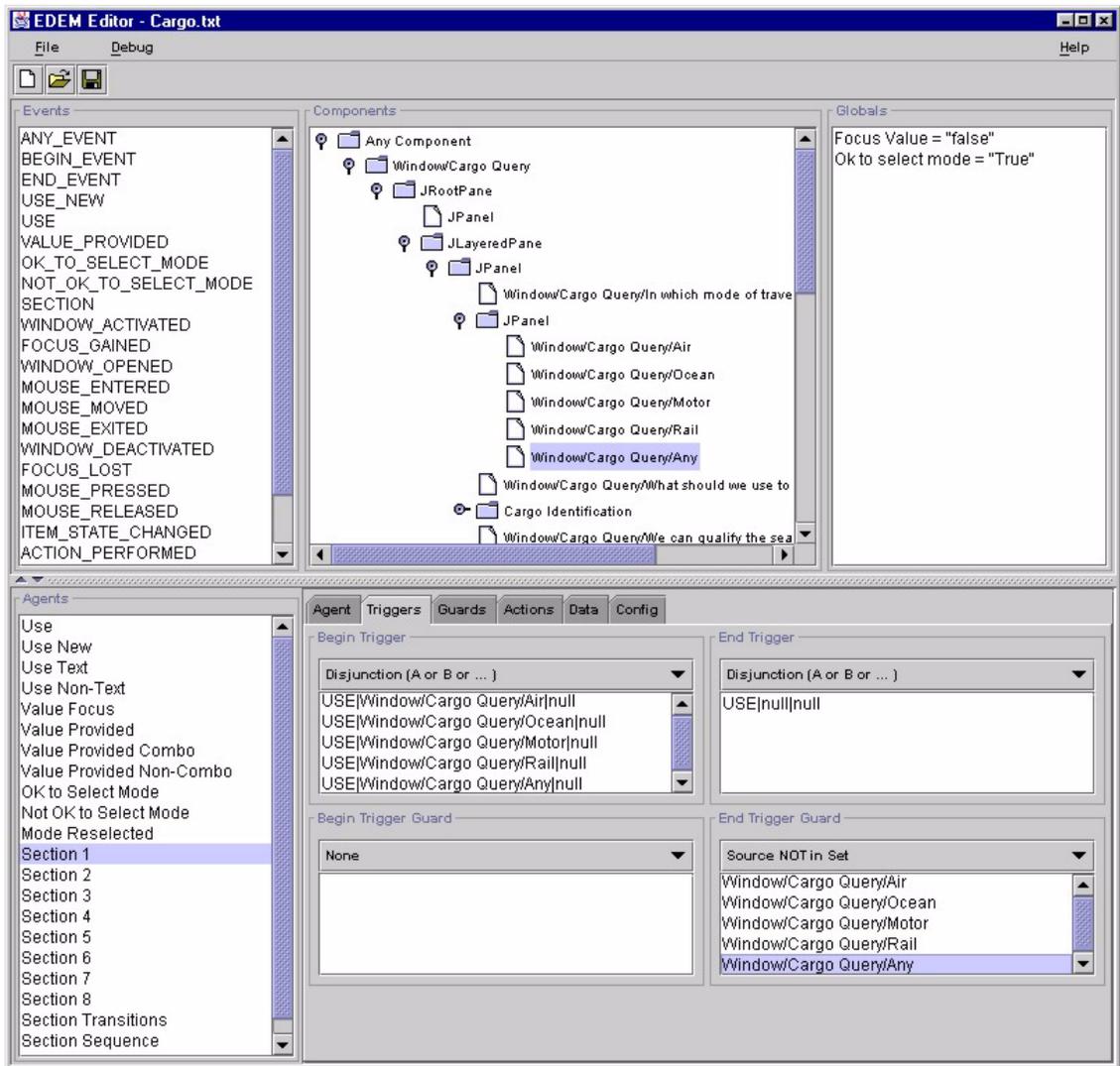


Figure 5-11. Agent authoring interface

layout of the form is well suited to the order in which users actually specify queries. These agents were then downloaded to users' computers (automatically upon application start-up) where they monitored user interactions and reported usage information and user feedback to developers.

In this case, the engineers decided to configure an agent to notify users when it detected behavior in violation of developers' expectations (Figure 5-12). By double-clicking on an agent notification (or by selecting the notification header and pressing the "Feedback" button), users could learn more about the violated expectation and could

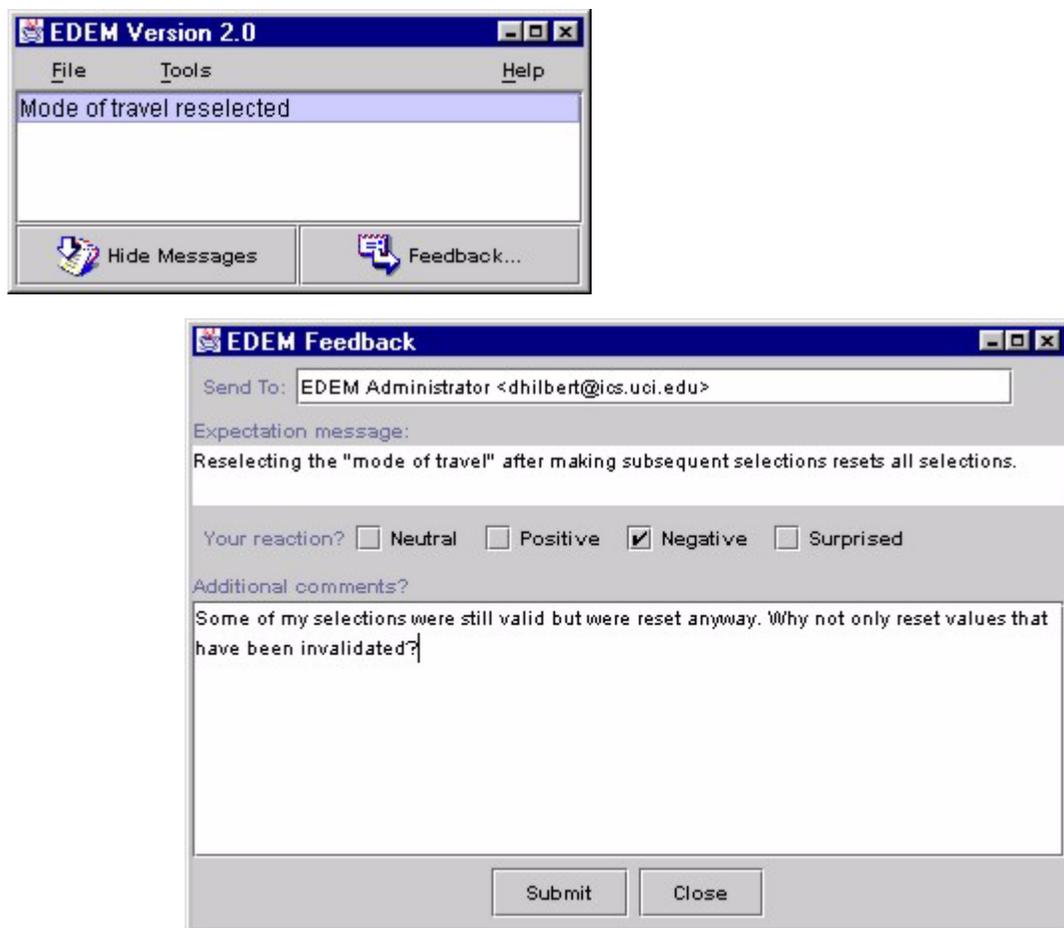


Figure 5-12. Agent notification and user feedback

respond with feedback if desired. Feedback was reported automatically via E-mail along with events associated with violations. Agents also unobtrusively logged and reported all violations along with other usage data via E-mail each time the application was exited. Agent-collected data and feedback was E-mailed to a help desk where it was reviewed by support engineers and entered into a change request tracking system. With the help of other systems, the engineers were able to assist the help desk in providing a new release of the interface to the user community based on the feedback and usage data collected from the field.

It is tempting to think that this example has a clear design flaw that, if corrected, would clearly obviate the need for collection of usage data and user feedback. Namely, one might argue, the application should automatically detect which selections must be reselected and only require users to reselect those values. To illustrate how this objection misses the mark, let us assume that one of the users actually responds to the agent with exactly this suggestion (Figure 5-12). After reviewing the agent-collected feedback, the engineers consider the suggestion, but unsure of whether to implement it (due to its impact on the current design, implementation, and test plans), decide to review the usage data log. The log, which documents over a month of use with over 100 users, indicates that this problem has only occurred twice, and both times with the same user. As a result, the developers decide to put the change request on hold.

The ability to base design and effort allocation decisions on empirical data in this way is one of the key contributions of this approach. Another important contribution is the explicit treatment of usage expectations in the development process. Treating usage

expectations explicitly helps developers think more clearly about the implications of design decisions. Because expectations can be expressed in terms of user interactions, they can be monitored automatically, thereby allowing expectations to be compared against actual use on a potentially large scale. Finally, expectations provide a principled way of focusing data collection so that only information useful in evaluating the design-use fit is captured.

CHAPTER 6: Problems and Solutions

Chapter 4 concluded with the identification of five key problems limiting the scalability and data quality of existing techniques for extracting usage and usability information from user interface events. This chapter begins with a brief review of the problems and then presents examples to illustrate how each of the problems can be addressed based on the constructs presented in the previous chapter. For the final problem, a reference architecture is presented to illustrate how independent evolution is achieved. Finally, relationships and interdependencies between the problems and solutions are discussed.

6.1 Problems

Abstraction: Questions about usage typically occur in terms of concepts at higher levels of abstraction than represented in user interface event data. Furthermore, questions of usage may occur at multiple levels of abstraction. This is primarily a data quality issue. Analysis of non-abstracted data is likely to produce results that are difficult to relate to concerns about application design and use, and patterns that might otherwise be obvious in abstracted data may go unnoticed in un-abstracted data.

Selection: The amount of data necessary to answer usage questions is typically a small subset of the much larger set of data that *might* be captured at any given time. Furthermore, selection is necessary to group data and define “contexts” in which to perform reduction. This is both a scalability and data quality issue. Failure to properly

select data of interest increases the amount of data that must be reported, and patterns that might otherwise be obvious in selected data may go unnoticed in un-selected data.

Reduction: Much of the analysis that will ultimately be performed to answer usage questions can actually be performed *during* data collection resulting in greatly reduced data reporting and post-hoc analysis needs. This is both a scalability and data quality issue. When analysis is left as a final step, it can be difficult to perform (due to forgotten or faulty assumptions in data collection) or may fail to be performed at all (due to missing context and/or lack of automated tools). Therefore, performing reduction during data collection not only decreases the amount of data that must be reported, but also increases the likelihood that successful analysis will in fact be performed.

Context: Potentially critical information necessary in understanding usage is often not available in event data alone. Such information may be spread across multiple events or missing altogether, but is often available “for the asking” from the user interface, application, artifacts, or user. This is primarily a data quality issue. Since user interface, application, artifact, and user state information can be critical in understanding usage, such information should be available when abstracting, selecting, and reducing data.

Evolution: Data collection needs typically evolve over time (perhaps due to results of earlier data collection) more rapidly than do applications. This is primarily a scalability issue. Unnecessary coupling of data collection and application code increases the cost of evolution and impact on users.

The following sections demonstrate how each of these problems can be addressed using the constructs presented in the previous chapter.

6.2 Abstraction

6.2.1 Abstract Interaction Events

At the most basic level, analyzing application use presupposes the ability to identify when users are pro-actively using the user interface and when values are being provided. However, identifying such abstract interaction events as “USE” and “VALUE_PROVIDED” can be more difficult than one would expect.

First, not all user interface events reliably indicate pro-active user interface use. For instance, mouse movements, while potentially interesting in their own right, often do not indicate pro-active use. Furthermore, events indicating use often differ from one component type to the next. For instance, key press events typically indicate textual component use while mouse press events typically indicate non-textual component use. Finally, events indicating pro-active use may occur multiple times within a single “episode” of use. For instance, multiple key press events are typically associated with a single use of a text field component. The following examples illustrate how these issues are addressed using the constructs presented in the previous chapter.

The following agent generates a “USE” abstract interaction event each time a “KEY_PRESSED” event is observed in a textual component:

Begin	Trigger	KEY_PRESSED * javax.swing.JTextField OR KEY_PRESSED * javax.swing.JTextArea OR KEY_PRESSED * javax.swing.JTextPane
	Action	PostEvent("USE", TriggerSource)

Figure 6-1. “Use Text” agent (abstract interaction events)

The following agent generates a “USE” abstract interaction event each time a “MOUSE_PRESSED” event is observed in a non-textual component:

Begin	Trigger	MOUSE_PRESSED * *
	Trigger Guard	SourceClassNotInSet(javax.swing.JTextField, javax.swing.JTextArea, javax.swing.JTextPane)
	Action	PostEvent("USE", TriggerSource)

Figure 6-2. “Use Non-Text” agent (abstract interaction events)

The following agent generates a “USE_NEW” abstract interaction event each time the source of “USE” events has changed:

Begin	Trigger	USE * *
	Trigger Guard	SourceIsNew()
	Action	PostEvent("USE_NEW", TriggerSource)

Figure 6-3. “Use New” agent (abstract interaction events)

However, not all “USE” events can be associated with users providing values. For instance, many component types are used primarily to trigger actions as opposed to providing values, such as menu items, toolbar buttons, push buttons and scrollbars. Furthermore, user interface components that *are* used to provide values are sometimes used without new values being provided. The following examples illustrates how these issues can be addressed using the constructs presented in the previous chapter.

The following agent stores an initial “Focus Value” whenever a “FOCUS_GAINED” event is detected:

Begin	Trigger	FOCUS_GAINED * *
	Action	UpdateState("Focus Value", ValueOf(TriggerSource))

Figure 6-4. “Value Initial” agent (abstract interaction events)

The following agent generates a “VALUE_PROVIDED” abstract interaction event whenever a “FOCUS_LOST” event is detected and the current component value is not equal to the previously stored “Focus Value”.

Begin	Trigger	FOCUS_LOST * *
	Guard	CheckState("Focus Value", "!=", ValueOf(TriggerSource))
	Action	PostEvent(VALUE_PROVIDED, TriggerSource)

Figure 6-5. “Value Provided” agent (abstract interaction events)

The purpose of these agents is to generate abstract events corresponding to component use and the provision of values so that other data collection code can refer to

these higher-level events as opposed to the lower-level events associated them. This simplifies data collection by factoring code that would be common across multiple agents, and localizes the impact of introducing new user interface components with different event and state semantics.

6.2.2 Relating Data to User Interface Features

It is also important to relate usage data to user interface features such as menus, toolbars, dialogs, and windows. This allows data to be related to user interface design considerations.

Agents can be defined to relate the use of menu items to their parent menus. The following agent generates “MENU” events relating the use of menu items to the “File Menu”:

Begin	Trigger	USE Window/Stylepad/MenuItem/New * OR USE Window/Stylepad/MenuItem/Open * OR USE Window/Stylepad/MenuItem/Save * OR USE Window/Stylepad/MenuItem/Print * OR USE Window/Stylepad/MenuItem/Exit *
	Action	PostEvent("MENU", AgentSource)

Figure 6-6. “File Menu” agent (relating data to user interface features)

The following agent generates “MENU” events relating the use of menu items to the “Edit Menu”:

Begin	Trigger	USE Window/Stylepad/MenuItem/Cut * OR USE Window/Stylepad/MenuItem/Copy * OR USE Window/Stylepad/MenuItem/Paste * OR USE Window/Stylepad/MenuItem/Undo * OR USE Window/Stylepad/MenuItem/Redo *
	Action	PostEvent(“MENU”, AgentSource)

Figure 6-7. “Edit Menu” agent (relating data to user interface features)

Agents can also be defined to relate the use of toolbar buttons to their parent toolbars. The following agent generates “TOOL” events relating the use of toolbar buttons to the “File Toolbar”:

Begin	Trigger	USE Window/Stylepad/ImageIcon/new.gif * OR USE Window/Stylepad/ImageIcon/open.gif * OR USE Window/Stylepad/ImageIcon/save.gif * OR USE Window/Stylepad/ImageIcon/print.gif *
	Action	PostEvent(“TOOL”, AgentSource)

Figure 6-8. “File Toolbar” agent (relating data to user interface features)

The following agent generates “TOOL” events relating the use of toolbar buttons to the “Edit Toolbar”:

Begin	Trigger	USE Window/Stylepad/ImageIcon/cut.gif * OR USE Window/Stylepad/ImageIcon/copy.gif * OR USE Window/Stylepad/ImageIcon/paste.gif *
	Action	PostEvent(“TOOL”, AgentSource)

Figure 6-9. “Edit Toolbar” agent (relating data to user interface features)

Agents can also be defined to indicate when application dialogs or windows are opened and closed. The following agent generates “OPEN” and “CLOSE” events to indicate when the “Print Window” is opened and closed:

Begin	Trigger	WINDOW_ACTIVATED Window/Print *
	Action	PostEvent("OPEN", AgentSource)
End	Trigger	WINDOW_CLOSING Window/Print * OR USE Window/Print/OK * OR USE Window/Print/Cancel *
	Action	PostEvent("CLOSE", AgentSource)

Figure 6-10. “Print Window” agent (relating data to user interface features)

The purpose of these agents is to allow the use of components in menus, toolbars, and windows to be related to their parent menus, toolbars, and windows. The abstract events generated by these agents can be used by other agents to characterize the use of whole menus, toolbars, and windows, as opposed to their constituents, as illustrated below.

6.2.3 Relating Data to Application Features

It is also important to relate usage data to application features such as application commands. This allows data to be related to application design considerations.

Agents can be defined to relate the various ways in which application commands may be invoked via the user interface to the commands themselves. The following agent

generates “CMD” events relating the various ways in which the “File->New” command can be invoked to the “File->New” command:

Begin	Trigger	USE Window/Stylepad/MenuItem/New * OR USE Window/Stylepad/ImageIcon/new.gif *
	Action	PostEvent("CMD", AgentSource)

Figure 6-11. “File->New” agent (relating data to application features)

The following agent generates “CMD” events relating the various ways in which the “File->Print” command can be invoked (and canceled) to the “File->Print” command:

Begin	Trigger	USE Window/Stylepad/ImageIcon/print.gif * OR USE Window/Print/OK * OR USE Window/Print/Cancel *
	Action	PostEvent("CMD", AgentSource)

Figure 6-12. “File->Print” agent (relating data to application features)

The purpose of these agents is to relate the use of commands to the different ways in which those commands may be invoked via the user interface. The abstract events generated by these agents can be used by other agents to characterize general command use, as opposed to specific command invocation methods, as is illustrated below.

6.2.4 Relating Data to Users’ Tasks and Goals

It is also important to relate usage data to aspects of users’ tasks and goals.

Users’ progress in completing tasks, particularly in form-based user interfaces, can sometimes be analyzed in terms of progress in completing “sections” of the interface.

The following agent detects when the first section in a database query interface has been completed:

Begin	Trigger	USE Window/Cargo Query/Air * OR USE Window/Cargo Query/Ocean * OR USE Window/Cargo Query/Motor * OR USE Window/Cargo Query/Rail * OR USE Window/Cargo Query/Any *
End	Trigger	USE * *
	Trigger Guard	SourceNotInSet(Window/Cargo Query/Air Window/Cargo Query/Ocean Window/Cargo Query/Motor Window/Cargo Query/Rail Window/Cargo Query/Any)
	Action	PostEvent("SECTION", AgentSource)

Figure 6-13. "Section 1" agent (relating data to users' tasks and goals)

The following agent detects when the second section in a database query interface has been completed:

Begin	Trigger	ITEM_STATE_CHANGED Cargo Identification *
End	Trigger	USE * *
	Trigger Guard	SourceNotInSet(Cargo Identification)
	Action	PostEvent("SECTION", AgentSource)

Figure 6-14. "Section 2" agent (relating data to users' tasks and goals)

The purpose of these agents is to relate user interface events to the steps required to complete tasks in the user interface. The abstract events generated by these agents can be used by other agents to characterize users' progress in completing tasks as is illustrated below.

6.3 Selection

6.3.1 Selecting Events

Event selection allows event data of interest to be captured, and unnecessary data filtered, prior to reporting and analysis. It also allows data to be grouped in various ways for abstraction and reduction purposes.

Once agents have been defined, as illustrated above, they can be configured to select and group event data in different ways for reduction purposes. For instance, the menu, toolbar, and command agents presented above may be used to capture and characterize the use of specific menu items, toolbar buttons, and commands by simply adding a simple Data specification.

The following agent can be used to capture and characterize use of the “File Menu”:

Begin	Trigger	USE Window/Stylepad/MenuItem/New * OR USE Window/Stylepad/MenuItem/Open * OR USE Window/Stylepad/MenuItem/Save * OR USE Window/Stylepad/MenuItem/Print * OR USE Window/Stylepad/MenuItem/Exit *
	Action	PostEvent("MENU", AgentSource)
	Data	RecordEventData()

Figure 6-15. “File Menu” agent (selecting events)

The following agent can be used to capture and characterize use of the “File Toolbar”:

Begin	Trigger	USE Window/Stylepad/ImageIcon/new.gif * OR USE Window/Stylepad/ImageIcon/open.gif * OR USE Window/Stylepad/ImageIcon/save.gif * OR USE Window/Stylepad/ImageIcon/print.gif *
	Action	PostEvent("TOOL", AgentSource)
	Data	RecordEventData()

Figure 6-16. “File Toolbar” agent (selecting events)

The following agent can be used to capture and characterize use of the “File->Print” command:

Begin	Trigger	USE Window/Stylepad/ImageIcon/print.gif * OR USE Window/Print/OK * OR USE Window/Print/Cancel *
	Action	PostEvent("CMD", AgentSource)
	Data	RecordEventData()

Figure 6-17. “File->Print” agent (selecting events)

However, there are other ways that events might be selected and grouped for reduction purposes.

For instance, the following agent can be used to capture and characterize the use of whole menus, as opposed to individual menu items:

Begin	Trigger	MENU * *
	Data	RecordEventData()

Figure 6-18. “All Menu” agent (selecting events)

The following agent can be used to capture and characterize the use of whole toolbars, as opposed to individual toolbar buttons:

Begin	Trigger	TOOL * *
	Data	RecordEventData()

Figure 6-19. “All Toolbars” agent (selecting events)

The following agent can be used to capture and characterize the use of all commands, as opposed to the specific methods used to invoke individual commands:

Begin	Trigger	CMD * *
	Data	RecordEventData()

Figure 6-20. “All Commands” agent (selecting events)

6.3.2 Selecting Event Contexts

Event selection can also be used to specify “contexts” in which to capture and reduce events.

The following agent uses Begin and End Triggers to define a context in which to capture “VALUE_PROVIDED” events (namely, while the “Print Window” is open):

Begin	Trigger	WINDOW_ACTIVATED Window/Print *
	Action	PostEvent("OPEN", AgentSource)
	Data	RecordEventData(VALUE_PROVIDED * *)
End	Trigger	WINDOW_CLOSING Window/Print * OR USE Window/Print/OK * OR USE Window/Print/Cancel *
	Action	PostEvent("CLOSE", AgentSource)

Figure 6-21. “Print Window” agent (selecting event contexts)

6.3.3 Selecting State

State selection allows state data of interest to be captured in addition to event data. However, since state information is always available, state selection not only involves selecting *what* state data to capture, but *when* to capture it. Thus, state selection depends on event selection. Examples illustrating state selection are presented in Section 6.5.

6.4 Reduction

6.4.1 Reducing Event Data

Event data can be reduced in a number of ways. Three simple pre-defined reduction algorithms were implemented as part of this research. The “Events” algorithm produces counts of individual event occurrences. The “Transitions” algorithm produces counts of transitions between events. The “Sequences” algorithm produces counts of whole event sequences between Begin and End Triggers.

The following agent invokes the “Events” reduction algorithm to characterize the relative frequency with which sections in a database query interface are completed:

Begin	Trigger	SECTION * *
	Data	RecordEventData(SECTION * *)

Figure 6-22. “Section Events” agent (reducing event data)

Rather than recording each event occurrence separately and performing reduction after data collection, the “Events” reduction algorithm stores data in a hashtable format in which keys indicate unique observed events and values indicate the number of times each event was observed. The following figure illustrates the effect of applying this algorithm:

User, Session, Agent, Type, Name, Value dhillbert,925836566850,Section Events,Agent,Time,64240 dhillbert,925836566850,Section Events,Agent,Total,35 dhillbert,925836566850,Section Events,Event,[SECTION Section 1],11 dhillbert,925836566850,Section Events,Event,[SECTION Section 2],6 dhillbert,925836566850,Section Events,Event,[SECTION Section 7],2 dhillbert,925836566850,Section Events,Event,[SECTION Section 8],5
--

Figure 6-23. “Section Events” data (reducing event data)

Each data record consists of a user identifier, a session identifier, the name of the agent responsible for generating the data, the type of data (agent, event, transition, sequence, value, or vector), and finally a name and value pair indicating the name of the data element and its value.

The following figure illustrates the effect of visualizing this data in graphical form:



Figure 6-24. “Section Events” data visualization (reducing event data)

The following agent invokes the “Transitions” reduction algorithm. Begin and End Triggers are used to indicate when queries begin and end. This way, transitions between the last section completed in one query and the first section completed in the next query are not recorded as transitions.

Begin	Trigger	SECTION * *
	Data	RecordEventTransitionData(SECTION * *)
End	Trigger	USE Window/Cargo Query/Reset * OR USE Window/Cargo Query/Submit *

Figure 6-25. “Section Transitions” agent (reducing event data)

The “Transitions” reduction algorithm stores data in a hashtable format in which keys indicate unique observed event transitions and values indicate the number of times each transition was observed. The following figure illustrates the effect of applying this algorithm:

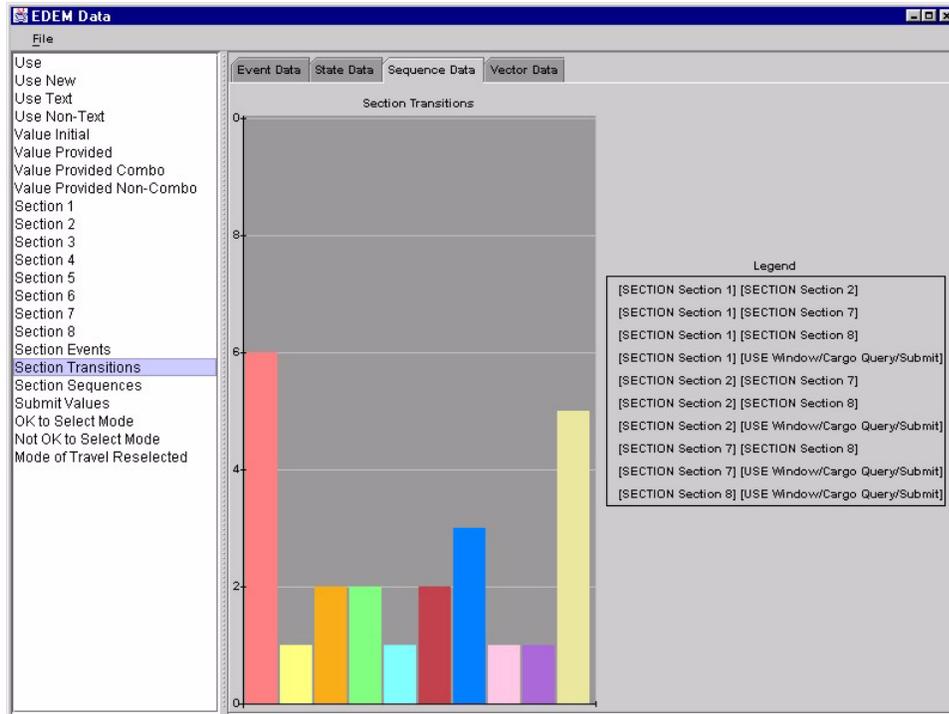


Figure 6-26. “Section Transitions” data (reducing event data)

It should be noted that the “Transitions” algorithm is basically equivalent to a first order Markov analysis or lag sequential analysis with lag=0. Introducing the notion of “lags” between events of interest would allow arbitrary lag sequential analyses to be performed.

The following agent invokes the “Sequences” reduction algorithm. Begin and End Triggers are used to segment sequences.

Begin	Trigger	SECTION * *
	Data	RecordEventSequenceData(SECTION * *)
End	Trigger	USE Window/Cargo Query/Reset * OR USE Window/Cargo Query/Submit *

Figure 6-27. “Section Sequences” agent (reducing event data)

The “Sequences” reduction algorithm stores data in a hashtable format in which keys indicate unique observed event sequences and values indicate the number of times each sequence was observed. The following figure illustrates the effect of applying this algorithm:

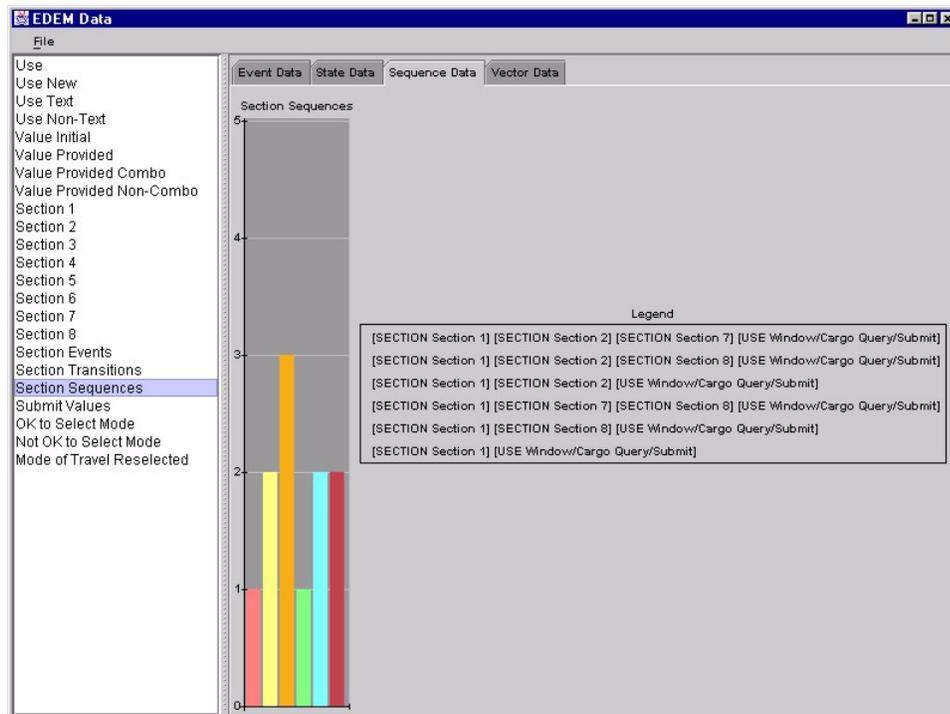


Figure 6-28. “Section Sequences” data (reducing event data)

It should be noted that the “Sequences” algorithm is roughly equivalent to Fisher’s Cycles analysis. The only difference is that in this case, sequence detection begins when a begin event is detected and doesn’t restart until after an end event is detected. In Fisher’s Cycles, sequence detection restarts each time a begin event is detected until an end event is detected.

6.4.2 Reducing State Data

Like event data, state data can be reduced in a number of ways. Three simple pre-defined reduction algorithms were implemented as part of this research. The “Values” algorithm calculates counts of individual values. The “Vectors” algorithm calculates counts of vectors of values. The “Append to Event Data” algorithm causes state information to be appended to event data which can then be reduced using the “Events” data reduction algorithm.

The following agent invokes the “Values” data reduction algorithm:

Begin	Trigger	USE Window/Cargo Query/Submit *
	Data	RecordStateData(Window/Cargo Query/Air Window/Cargo Query/Ocean Window/Cargo Query/Motor Window/Cargo Query/Rail Window/Cargo Query/Any Cargo Identification Cargo Qualification Cargo Status Cargo Location From Month, From Day, From Year, From Hour, From Minute To Month, To Day, To Year, To Hour, To Minute, List by Location, Summarize by Location, Summarize All)

Figure 6-29. “Submit Values” agent (reducing state data)

The “Values” reduction algorithm stores data in a hashtable format in which keys indicate unique observed values and hashtable values indicate the number of times each value was observed. The following figure illustrates the effect of applying this algorithm:

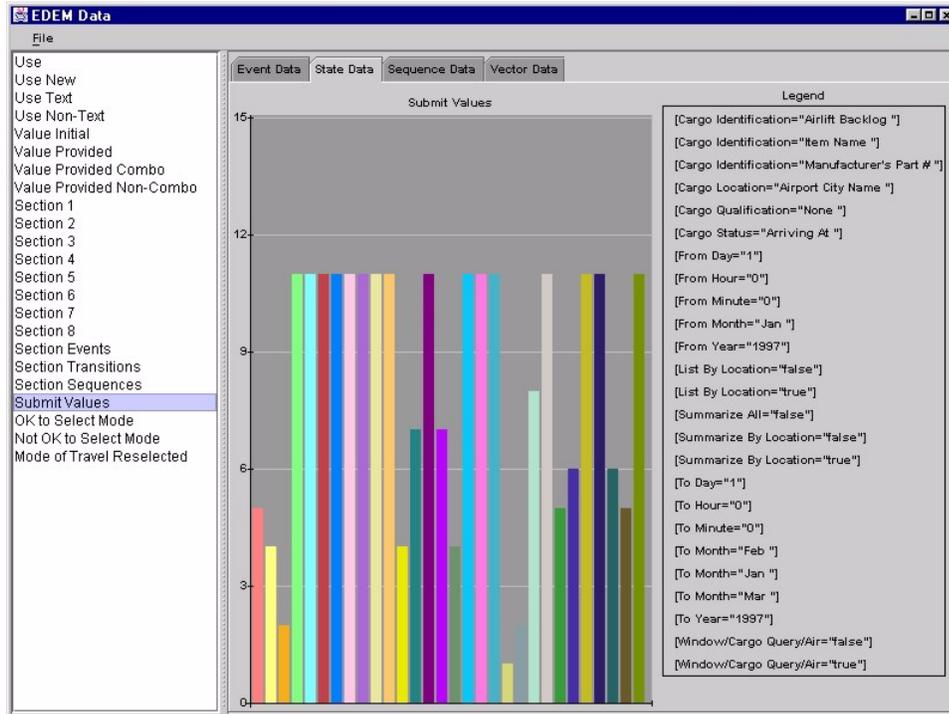


Figure 6-30. “Submit Values” data (reducing state data)

The following agent invokes the “Vectors” reduction algorithm:

Begin	Trigger	USE Window/Print/OK *
	Data	RecordStateVectorData("Print to File", "All", "Current Page", "Pages:")

Figure 6-31. “Print Mode & Pages” agent (reducing state data)

The “Vectors” reduction algorithm stores data in a hashtable format in which keys indicate unique observed vectors of values and hashtable values indicate the number of

times each vector of values was observed. This allows co-occurrence of values to be analyzed. The following figure illustrates the effect of applying this algorithm:

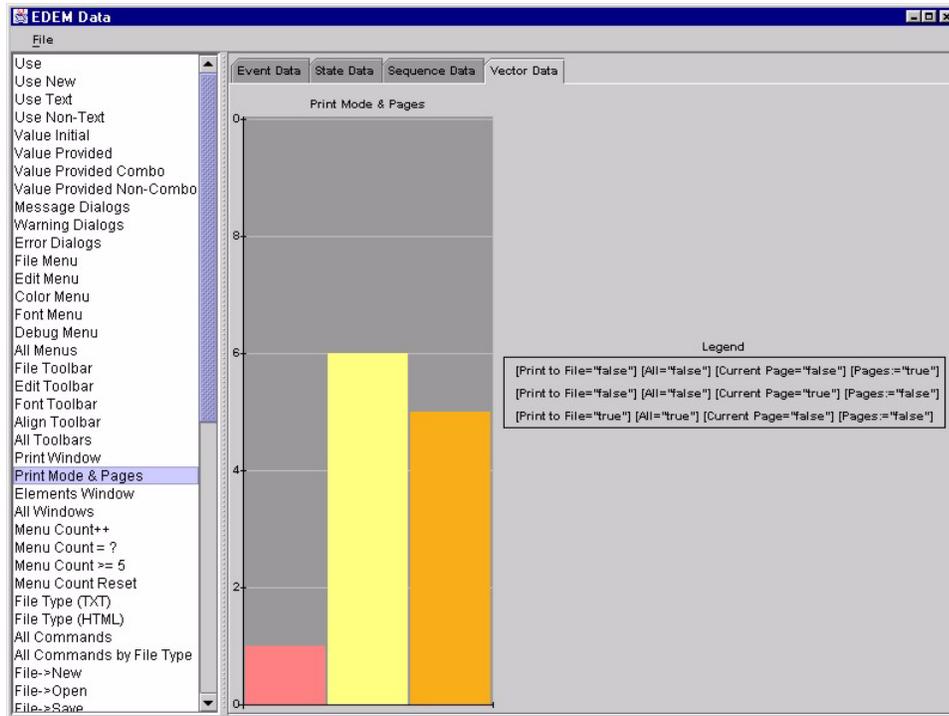


Figure 6-32. “Print Mode & Pages” data (reducing state data)

Finally, the following agent invokes the “Append to Event Data” algorithm causing state data to be reduced along with event data:

Begin	Trigger	CMD * *
	Data	RecordEventData() RecordStateDataPerEvent("File Type")

Figure 6-33. “All Commands” agent (reducing state data)

The “Append to Event Data” option allows state data to be appended to each event being reduced using the “Events” data reduction algorithm. This way, event data can be

analyzed based on state information captured at the time of event occurrences. The following figure illustrates the effect of appending state data to event data:

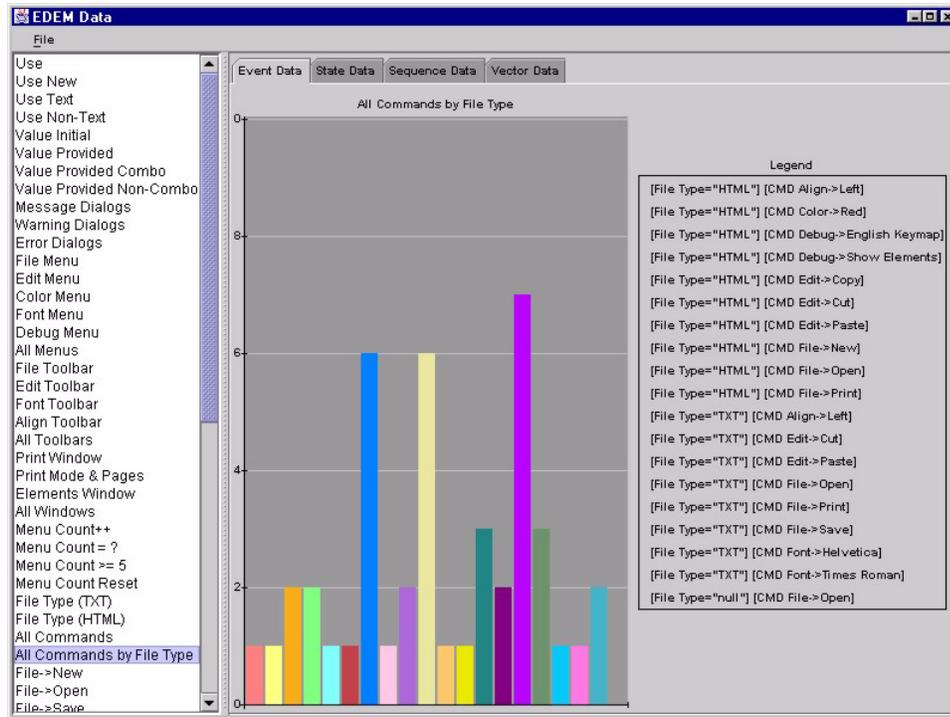


Figure 6-34. "All Commands by File Type" data (reducing state data)

6.5 Context

6.5.1 Incorporating User Interface State

Some of the examples above have already illustrated the capture of user interface state information.

A common technique is to capture selected user interface state information when application dialogs or windows are closed:

Begin	Trigger	WINDOW_ACTIVATED Window/Print *
	Action	PostEvent("OPEN", AgentSource)
End	Trigger	WINDOW_CLOSING Window/Print * OR USE Window/Print/OK * OR USE Window/Print/Cancel *
	Action	PostEvent("CLOSE", AgentSource)
	Data	RecordStateData("Printer Name", "Printer Status", "Printer Type", "Print to File", "All", "Current Page", "Pages:", "Pages", "Number of Copies", "Collate")

Figure 6-35. "Print Window" agent (user interface state)

There are also other ways that user interface state might be incorporated in analysis. The following agent, which observes the use of a phone service provisioning form, checks to see if the ZIP field in the address section of the form is empty whenever the City or State fields are edited:

Begin	Trigger	VALUE_CHANGED Window/Phone/Control/City * OR VALUE_CHANGED Window/Phone/Control/State *
	Guard	CheckState(Window/Phone/Control/ZIP, "=", "")
	Data	RecordUserData("Enter ZIP to Complete City/State", "The City and State can typically be completed automatically based on the ZIP.")

Figure 6-36. "Enter ZIP to complete City/State" agent (user interface state)

6.5.2 Incorporating Arbitrary State

In addition to checking and capturing user interface state, agents may also update, check, and capture arbitrary global state.

The following agent updates a global variable indicating when it is okay to make a selection in the “mode of travel” section in a database query interface:

Begin	Trigger	BEGIN_EVENT * * OR USE Window/Cargo Query/Reset * OR USE Window/Cargo Query/Submit *
	Action	UpdateState("Ok to select mode", True)

Figure 6-37. “OK to Select Mode of Travel” agent (arbitrary state)

The following agent updates the same global variable indicating when it is *not* okay to select the “mode of travel”:

Begin	Trigger	USE * *
	Trigger Guard	SourceNotInSet(Window/Cargo Query/Air Window/Cargo Query/Ocean Window/Cargo Query/Motor Window/Cargo Query/Rail Window/Cargo Query/Any Window/Cargo Query/Reset Window/Cargo Query/Submit)
	Action	UpdateState("Ok to select mode", False)

Figure 6-38. “Not OK to Select Mode of Travel” agent (arbitrary state)

The following agent then checks the “Ok to select mode” global variable whenever the user makes a selection in the “mode of travel” section of the database query interface:

Begin	Trigger	USE Window/Cargo Query/Air * OR USE Window/Cargo Query/Ocean * OR USE Window/Cargo Query/Motor * OR USE Window/Cargo Query/Rail * OR USE Window/Cargo Query/Any *
	Guard	CheckState("Ok to select mode", "!=" , True)
	Action	UpdateState("Ok to select mode", True)

Figure 6-39. “Mode of Travel Reselected” agent (arbitrary state)

Here is another example of how arbitrary state might be incorporated in analysis. The following agent increments the value of a global counter variable each time a new menu is opened:

Begin	Trigger	ITEM_STATE_CHANGED * javax.swing.JMenu
	Trigger Guard	SourceIsNew()
	Action	UpdateState("Menu Count", ValueOf("Menu Count") + 1)

Figure 6-40. “Menu Count Increment” agent (arbitrary state)

The following agent resets the same global variable each time a non-menu component is used:

Begin	Trigger	USE * *
	Trigger Guard	SourceClassNotInSet(javax.swing.JMenu)
	Action	UpdateState("Menu Count", 0)

Figure 6-41. "Menu Count Reset" agent (arbitrary state)

The following agent is then defined to check the value of the global variable each time a menu item is selected, and to record the event if the count of previously opened menus is greater than five:

Begin	Trigger	USE * javax.swing.JMenuItem
	Guard	CheckState("Menu Count", ">", 5)
	Data	RecordEventData()

Figure 6-42. "Menu Count > 5" agent (arbitrary state)

6.5.3 Incorporating Application State

It sometimes makes sense to incorporate application state information in analysis. In some cases, application state can be inferred based on user interface state. For instance, the following agents update a global state variable indicating the current "File Type" being edited in a word processing application based on the file name stored in the application's title bar:

Begin	Trigger	USE * *
	Guard	CheckState("Window/Styelpad", "Ends w/", ".txt")
	Action	UpdateState("File Type", "TXT")

Begin	Trigger	USE * *
	Guard	CheckState("Window/Styelpad", "Ends w/", ".html") OR CheckState("Window/Styelpad", "Ends w/", ".htm")
	Action	UpdateState("File Type", "HTML")

Figure 6-43. "File Type" agents (application state)

In other cases, application state may be more difficult to infer. In such cases, the Event Service API may be used by the application to store application state in global state variables that may then be accessed by agents in the same way that all global variables are accessed.

6.5.4 Incorporating Artifact State

In some cases, data about the structure and/or content of application artifacts may also be useful in analysis. In most cases, applications must provide such information. For instance, a word processing application might calculate statistics about the contents of documents and then use the Event Service API to store such data in global variables and generate events to notify agents of when to capture the data stored in those variables.

6.5.5 Incorporating User State

In some cases, it may also be helpful to involve users in data collection. For instance, the following agent posts a user notification to inform users of the effects of

reselecting the “mode of travel” in a database query interface, allowing users to provide feedback if desired:

Begin	Trigger	USE Window/Cargo Query/Air * OR USE Window/Cargo Query/Ocean * OR USE Window/Cargo Query/Motor * OR USE Window/Cargo Query/Rail * OR USE Window/Cargo Query/Any *
	Guard	CheckState("Ok to select mode", "!=" , True)
	Action	UpdateState("Ok to select mode", True)
	Data	RecordUserData("Mode of Travel Reselected", "Reselecting the 'mode of travel' after making subsequent selections resets all selections.")

Figure 6-44. “Mode of Travel Reselected” agent (user state)

Such incorporation of user state must be implemented carefully so as to avoid distracting and potentially annoying users. This is discussed further in Chapter 9.

6.6 Evolution

The following reference architecture illustrates the components typically required to perform large-scale usage data collection:

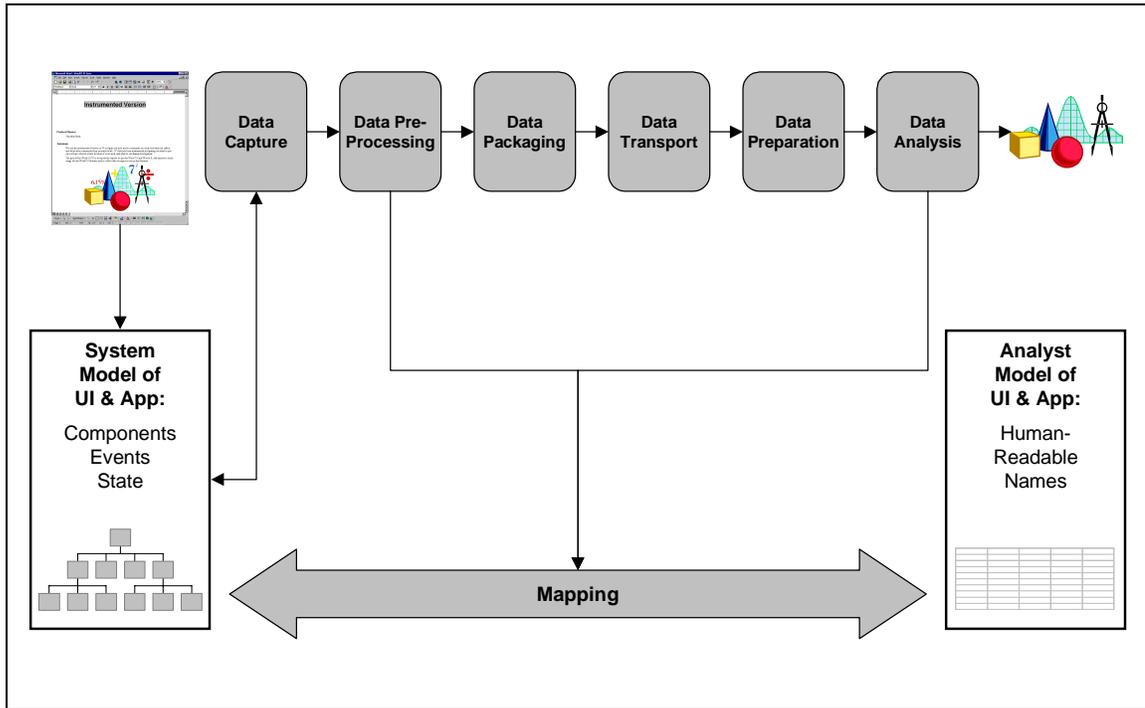


Figure 6-45. Data collection reference architecture

Data Capture mechanisms are required to allow information about user interactions and/or application behavior to be accessed for data collection purposes (e.g. code that taps into the user interface event queue or the application command dispatch routine).

Data Pre-Processing mechanisms are required to select data to be reported from the mass of data that might potentially be reported (e.g. instrumentation code inserted into application code or separate event “filtering” code outside of the application).

Data Packaging mechanisms are required to make selected data persistent in preparation for transport (e.g. code to write data to disk files or to package it into E-Mail messages — perhaps employing compression if necessary).

Data Transport mechanisms are required to transfer captured data to a location where aggregation and analysis can be performed (e.g. code that copies data to removable media to be mailed or code to support automatic transport via E-Mail or the Word Wide Web).

Data Preparation mechanisms are required to transform captured data into a format that is ready for aggregation and analysis (e.g. code that uncompresses captured data if necessary and writes it to a database importable format).

Data Analysis mechanisms are required to aggregate, analyze, visualize, and report the results of data collection (e.g. database, spreadsheet, and statistical packages — this is where most abstraction, selection, and reduction of data is typically performed).

And finally, *Mappings* are required to map between the implementation-dependent IDs associated with captured data and conceptual features of the UI and/or application being studied. This can be particularly important in pre-processing and analysis (e.g., header files or database tables that map between implementation-dependent IDs and human-readable names associated with user interface and application features).

The following figure illustrates how the reference architecture is instantiated when instrumentation code is inserted directly into application code:

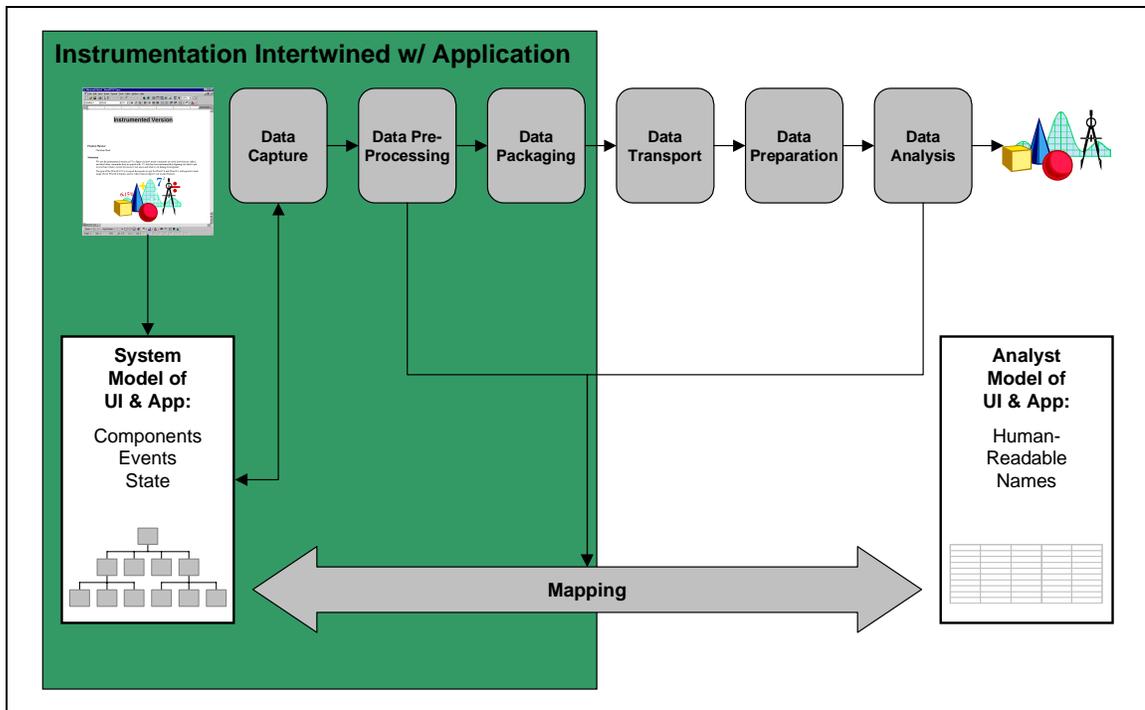


Figure 6-46. Instrumentation-based data collection architecture

The problem with this approach is that in order to modify data collection code, the application must be modified, re-compiled, re-tested, and re-deployed.

The following diagram illustrates an improved instantiation of the reference architecture based on an event monitoring approach:

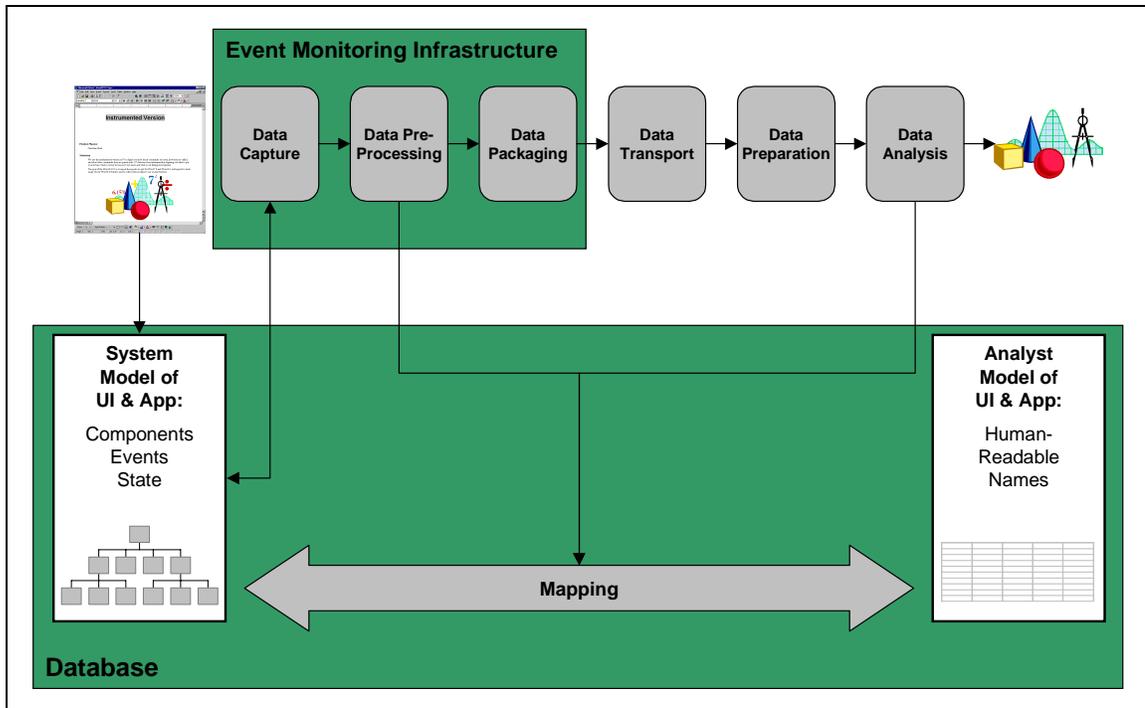


Figure 6-47. Event monitoring-based data collection architecture

This approach is preferable in that it separates data collection code from application code. Updates to data collection no longer directly affect application code. However, unless the event monitoring infrastructure can be modified and re-deployed without affecting application use, then the evolution problem has still not been solved from the point of view of users.

Finally, the following diagram illustrates the proposed instantiation of the reference architecture based on an event monitoring approach coupled with “pluggable”

pre-processing code capable of performing in-context data abstraction, selection, and reduction:

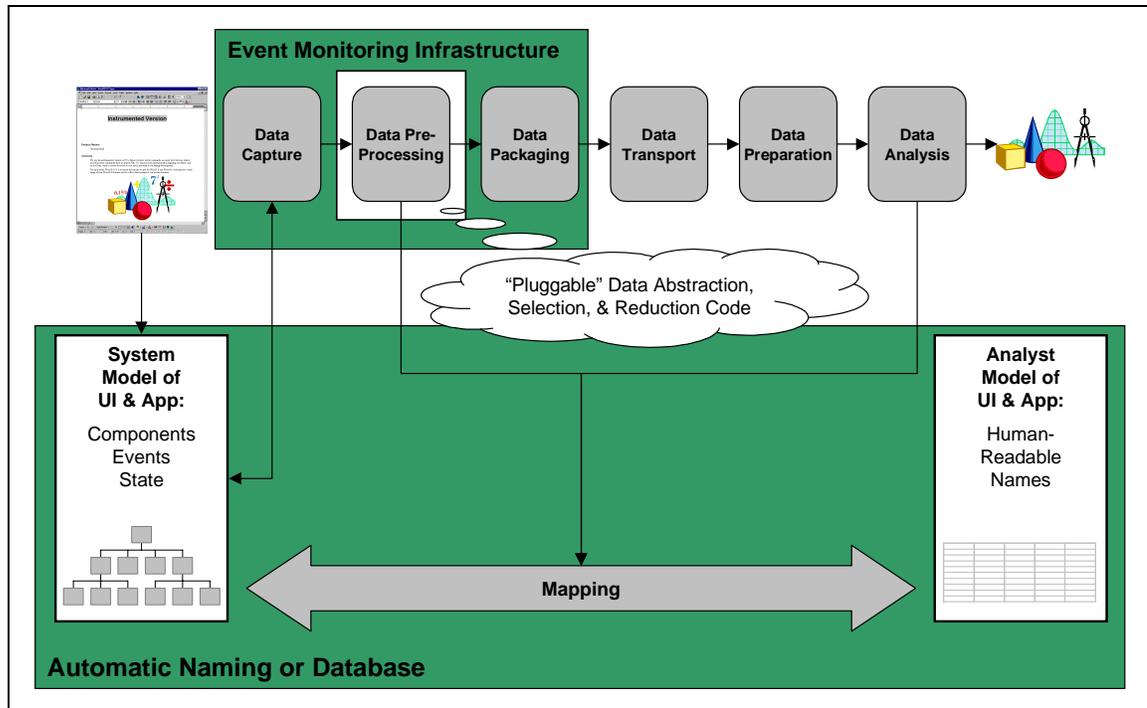


Figure 6-48. Proposed data collection architecture

In this approach, data collection can be modified over time without impacting application deployment or use by simply re-deploying “pluggable” pre-processing code. This approach involves moving work that was typically performed in the analysis phase into automated pre-processing code that is executed during data collection. This allows data abstraction, selection, and reduction to be performed in-context, resulting in significantly improved data quality and reduced data reporting and post-hoc analysis needs.

The approach described in this dissertation instantiates the reference architecture in this way. Agents are “pluggable” pre-processing modules that can be evolved and updated independently from application code. Once agents have been defined, they are serialized and stored in ASCII format in a file that is associated with a Universal Resource Locator (URL) on a development computer. The URL is passed as a command-line argument to the application of interest. When the application of interest is run, the URL is automatically downloaded and the latest agents are instantiated on the user’s computer. A standard HTTP server is used to field requests for agent specifications and a standard E-mail protocol is used to send agent reports back to development computers. Since data collection code can be added and deleted incrementally, investment in data collection is incremental, and as new issues or concerns arise (e.g., new suspected problems identified in the usability lab), new data collection code can be deployed to observe how frequently those issues actually arise in practice.

6.7 Interrelationships and Dependencies

While this chapter has attempted to address each of the identified problems relatively independently, the problems and their solutions are in fact interrelated in important ways.

Abstraction: Events must be selected in order to be abstracted, and in some cases, as with the “Value Provided” agent, access to context is also necessary. Thus, abstraction depends on selection and access to context.

Selection: While selection is a prerequisite for performing abstraction, the results of abstraction are often selected for capture. Selection also depends on access to context, not only by virtue of depending on abstraction, but also because contextual information, and not just event data, is often selected for capture. Thus, selection depends on abstraction and access to context.

Reduction: Reduction is performed on the results of abstraction and selection. Thus, reduction is dependent on abstraction, selection, and access to context (by virtue of depending on abstraction and selection).

Context: As indicated above, abstraction, selection, and reduction all depend on access to context.

Evolution: Finally, there is a tension between evolution and the other issues since the other issues require data collection code to be tightly integrated with user interface and application event and state information. However, this can be addressed by encapsulating data collection decisions in code that is independent of application code and that can be updated independently as described above.

The following diagram illustrates the interrelationships between solutions to the abstraction, selection, reduction, context, and evolution problems:

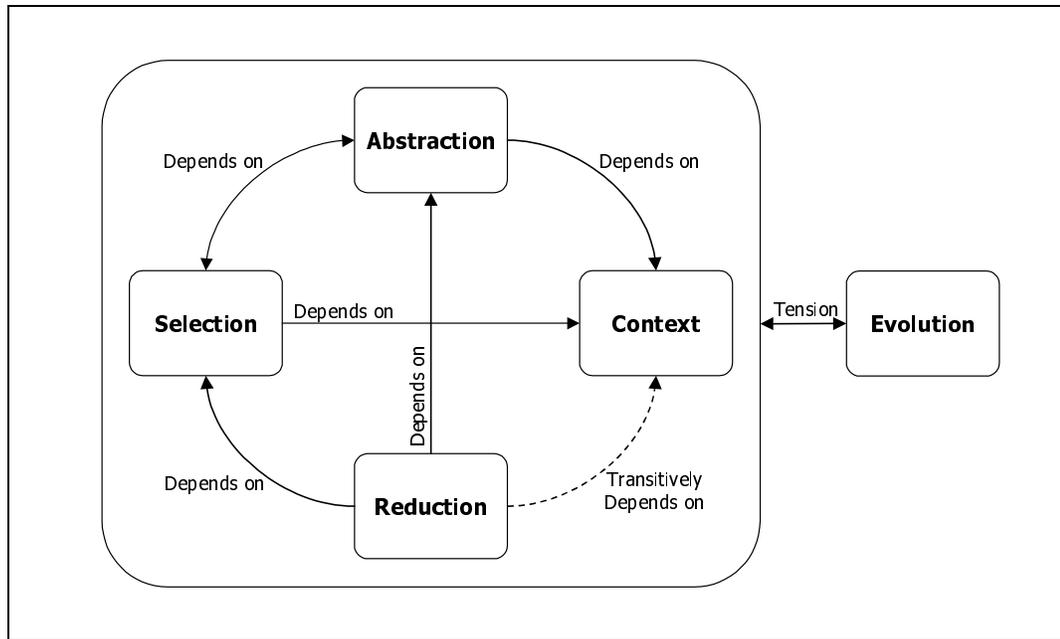


Figure 6-49. Interrelationships between the problems and solutions

CHAPTER 7: Methodological Considerations

The previous two chapters have focused primarily on technical issues. This chapter focuses on theoretical and methodological issues involved in data collection, analysis, and interpretation, and discusses how these tasks might be incorporated into existing development practices.

7.1 Theory of Expectations

As discussed in the introduction, the work presented here is founded on the following basic principles:

- That developers have expectations about application use that affect application design, and that designs often embody usage expectations even when developers are not explicitly aware of them.
- That mismatches between expected and actual use indicate potential problems in design or use that may negatively impact usability and utility.
- That making expectations more explicit and observing use to compare users' actions against expectations can help in identifying and resolving mismatches.
- That such mismatch identification and resolution can help bring expectations, and thus designs, into better alignment with actual use.

This section discusses, in more detail, a simple “theory of expectations” that provides the foundation for this work.

When developers design systems, they typically rely on a number of expectations, or assumptions, about how those systems will be used. We call these *usage expectations* [Girgensohn et al. 1994]. Developers' expectations are based on their knowledge of requirements, knowledge of the specific tasks and work environments of users, knowledge of the application domain, and past experience in developing and using applications themselves. Some expectations are explicitly represented, for example, those that are specified in requirements and in use cases. Others are implicit, including assumptions about usage that are encoded in user interface layout and application structure.

For instance, implicit in the layout of most data entry forms is the expectation that users will complete them from top to bottom with only minor variation. In laying out menus and toolbars, it is usually expected that frequently used or important features can be easily recognized and accessed, and that features placed on the toolbar will be more frequently used than those deeply nested in menus. Such expectations are typically not represented explicitly, and as a result, fail to be tested adequately.

Detecting and resolving mismatches between developers' expectations and actual use is important in improving the fit between design and use. Once mismatches are detected, they may be resolved in one of two ways. Developers may adjust their expectations to better match actual use, thus refining system requirements and eventually making the system more usable and/or useful. For instance, features that were expected to be used rarely, but are used often in practice can be made easier to access and more efficient. Alternatively, users can learn about developers' expectations, thus learning how

to use the existing system more effectively. For instance, learning that they are not expected to type full URLs in Netscape Navigator™ can lead users to omit characters such as “http://” in standard URLs, not to mention “www.” and “.com” in commercial URLs such as “http://www.amazon.com/”.

Thus, it is important to identify, and make explicit, usage expectations that importantly affect, or are embodied in, application designs. This can help developers think more clearly about the implications of design decisions, and may, in itself, promote improved design. Usage data collection techniques should then be directed at capturing information that is helpful in detecting mismatches between expected and actual use, and mismatches may then be used as opportunities to adjust the design based on usage-related information, or to adjust usage based on design-related information.

The following sections discuss methodological issues involved in collecting, analyzing, and interpreting usage information.

7.2 Data Collection

It is interesting to note that user interface design guidelines often make reference to, or assume knowledge of, the frequency and sequence of user actions. Because of this, and because the impact assessment and effort allocation problems are so important (even with respect to the task of collecting usage data itself) it makes sense to capture generic frequency and sequence data at a minimum. Such data can then help focus more detailed data collection and provide a backdrop against which to assess the relative importance of identified issues. This is why the examples in the previous chapter have focused primarily

on relatively generic data collection agents. In any large-scale usage data collection project, the following types of data should be considered:

Data about the user. At a minimum, there should be some way of identifying users so that data can be tabulated per user. If more information is available, for example, some characterization of the user's skill level, company, or occupation, this can be used to enrich analysis further. In some cases, user feedback may also be collected as part of the data collection process allowing subjective information to complement objective data. Finally, if users allow themselves to be identified, for example, by E-mail address, the results of data collection might be used to select particular users for follow-up communication.

Data about sessions. At a minimum, there should be a way of identifying sessions in which data was collected so that data can be tabulated per session. If more information is available, for instance, when the session occurred, how long it lasted, or information regarding the computing environment in which the session occurred, this can be used to enrich analysis further.

Data about user and application actions. This is perhaps the most common and generic type of data. This includes data about command use, menu use, toolbar use, dialog use, and so forth. The previous chapter presented a number of examples of agents for capturing this sort of information. In the case of dialog- and form-based interactions, it may also be helpful to capture information about transitions and sequences of interactions and values provided so that the layout and default values in dialogs and forms can be

tuned to better fit actual use. In applications where actions are performed automatically on behalf of users, it is also useful to collect data regarding how often users accept or reject the results of such automated actions. Finally, it is also useful to capture information regarding errors and the use of help, particularly if such information can be associated with application features.⁷ As mentioned above, it should be possible to link data of this type to user and session information so that actions can be analyzed in terms of the number of users performing them and the number of sessions in which they occurred.

Data about relatively persistent information (e.g., user preferences, customizations, and configurations). For instance, data about which features are enabled or disabled, which custom dictionary entries are added, and which import and export filters are installed can be very useful — particularly in providing sensible defaults — despite the fact that the actual user interactions associated with setting these “values” do not occur frequently.

Chapter 6 presents a number of example agents that can be used to capture such information. Some are relatively generic and can be applied with little (or no) modification across multiple applications. Others are more specifically targeted at detailed expectations regarding particular user interfaces. Chapter 6 also illustrates how data might be represented and linked to users and sessions.

7. There is some evidence to suggest that there may be a number of other generic “usability problem indicators” that might be used to identify potential usability problems automatically — such as multiple successive uses of the same user interface object (e.g. a popup menu or dialog) or multiple perusals of the same container object (e.g., a list or menu) without a selection being made [Swallow et al. 1997]. However, more work is needed in this area.

Note that the detail and specificity of the data that might be collected is limited only by the imagination and availability of resources for performing data collection coupled with considerations regarding the privacy, security, and general willingness of users to be involved in data collection. This section outlines some of the fundamental data collection requirements that should be considered before extra effort is expended on collecting special-purpose application-specific data.

7.3 Data Analysis

When collecting usage data, it is important to consider how results will be aggregated and analyzed. As mentioned above, results are often analyzed in terms of user or application actions, where actions can refer to low level actions in the user interface, higher level abstract interactions, use of user interface and application features, application warnings and errors, or even specific violations of expected use. Results may be presented in the following forms:

- Number and % of users that do X
- Number and % of actions that are of type X
- Time and % of time spent doing X

In this way, the relative importance of observed events can be determined with reference to the number of users affected, the relative frequency of occurrence with respect to other comparable events, and the amount of user time affected. These numbers can be further qualified based on state information, for example:

- Number and % of users that do X in state Y
- Number and % of actions of type X in state Y
- Time and % of time spent doing X in state Y

Introducing the notion of state allows usage data to be compared across multiple “modes of use”. For instance, usage statistics for a general purpose word processing program may differ significantly depending on whether it is being used as a text editor, E-mail editor, or Web page editor. This sort of data can be important to developers wishing to strategically optimize features associated with particular types of use, for example, E-mail editing. These numbers can be further qualified based on counts and timing constraints, for example:

- Number and % of users that do X over Y times
- Number and % of users that do X for over Y length of time

In this way, events that occur frequently or for extended periods of time in small groups of users may also be identified, even if these events do not figure prominently in aggregate counts. Finally, numbers may also be reported in terms of application concepts such as sessions, documents, and so forth, for example:

- Number and % of sessions in which X occurred...
- Number and % of documents in which X occurred...

Finally, it may also be useful to analyze data about relatively persistent information such as user preferences, customizations, configuration settings, and so forth, for example:

- Number and % of sessions or documents in which X was enabled/disabled
- Number and % of users who added X to their custom dictionary
- Number and % of users who installed X as part of their configuration

Such information can be particularly useful in tuning defaults and in providing a good starting points for application features that must “learn” about user behavior to provide personalized functionality. Furthermore, such information may also be helpful in deciding what functionality is core functionality that should be shipped with the product versus functionality that might be separated and sold separately.

7.4 Data Interpretation

The most difficult problem is determining what exactly the data means and how to act on it. This will depend on developers’ expectations, goals, priorities, and resource considerations. Assume, for instance, that a study has been conducted to characterize the relative use of application features in a word processing program. For any given feature, a high or low usage measure can indicate a number of contradictory courses of action. For instance, a *low usage measure* may indicate the following courses of action:

- No change — this level of usage is what was expected
- Increased effort — this feature is not used enough because it needs improvement
- Reduced effort — this feature is not worth improving because it is not used enough
- Drop — this feature is so poorly designed and unused it's not worth keeping

A high usage measure may indicate the following:

- No change — this level of usage is what was expected
- Increased effort — this feature is worth improving since it is used so much
- Reduced effort — this feature is used so much because it needs no improvement
- Add elsewhere— this feature is so well designed and used it's worth adding to other parts of the application

Furthermore, while increasing the use of application features is often an unstated goal, developers may wish to drive usage measures in either direction. However, the reason for a low or high measure, and the correct course of action to reverse the trend may not be evident from the data alone. In cases where developers wish to *increase* use, there are a number of possible reasons for the lower than desired measure, and a number of possible corrective actions:

- *Discoverability* (i.e., users are not aware of the feature's existence) — perhaps developers need to improve the accessibility and/or delivery of the feature, or make it default.

- *Understandability* (i.e., users are not aware of the feature’s purpose) — perhaps developers need to improve user knowledge and/or delivery of the feature, or make it default.
- *Usability* (i.e., users are not able to use the feature easily) — perhaps developers need to improve the design and/or implementation of the feature to increase learnability, efficiency, memorability, error handling, and/or satisfaction.
- *Utility* (i.e., users are not able to use the feature fruitfully) — perhaps developers need to either drop the feature, or adapt it to better fit user needs.

In cases where developers wish to *decrease* use, there are a number of possible reasons for the higher than desired measure, and a number of possible corrective actions:

- There are *better existing alternatives* for accomplishing the same effect (a.k.a. “usage kludge”) — perhaps developers need to improve the discoverability, understandability, usability, and/or utility of alternative ways of accomplishing the same effect.
- There are *better potential alternatives* for accomplishing the same effect (a.k.a. “design kludge”) — perhaps developers need to improve the application design to include better alternatives for accomplishing the same effect.

Finally, the interpretation of results can vary significantly depending on the nature of the feature in question. For instance, some features, such as Wizards and templates, help automate processes that generate persistent results. If the results can be saved, copied, and reused, then it may not be significant that a Wizard or template was used once and then never used again. For features that do not produce such results, the same usage scenario could indicate problems with the feature.

7.5 Process Integration

In addition to deciding what data to collect, how to analyze it, and how to interpret and act on results, it is important to devise an appropriate strategy for incorporating data collection into the broader development process.

To begin with, data collection infrastructure will need to be purchased and/or developed in-house. At present, tools exist that support the capture and transport of limited usage data and user feedback [ErgoLight Usability Software 1998; Aqueduct Software 1998; Full Circle Software 1998]). However, no scalable tools currently exist for capturing arbitrary user interaction information on a large and ongoing basis. Thus, some design, development, and testing effort will likely need to be expended developing and maintaining data collection infrastructure. However, it should be noted that this work will differ from product development in at least two important ways. First, it will not directly or immediately improve the product itself, a consideration for organizations in which development resources are highly focused on achieving just these purposes. Second, it may be possible to amortize such work across more than one product, assuming multiple products exist. As a result, “special” or “shared” resources may need to be allocated for this purpose.

A “data collection manager” should be identified and charged with the responsibility of overseeing infrastructure development as well as the data collection and analysis process itself. The data collection manager acts as a central point-of-contact for other stakeholders wishing to capture specific usage information for particular purposes,

for example, marketers wishing to better understand the current user population, planners wishing to gain ideas for the next version, and development and usability professionals wishing to base design, impact assessment, and effort allocation decisions on empirical data about use. Since usage data will typically impact the “next version” of a product, and not the current version, the data collection manager will most likely need to be in a product planning role, as opposed to being a member of the “front-line” design and development team. However, as stated above, this person will require access to design, development, and testing resources.

In addition to the design and development resources needed to support data collection infrastructure, product designers and developers will also need to become involved. The goal is for data collection to become an integral part of the feature design, development, and justification process. First, designers should think clearly about the assumptions surrounding design decisions. Existing data should be used, when available, to help make such decisions. Designers should also consider what data might be captured to help measure the impact of new features or feature updates and include this as part of the design specification. If necessary data cannot be easily gleaned from already available user interaction data, then the design spec should include plans for generating necessary information and communicating it to the data collection infrastructure. This may be done in collaboration with the data collection manager and associated staff, if necessary. Once data collection has become part of the feature design and development process, data collection functionality — in addition to application functionality — will need to be tested as part of the normal testing process.

However, as long as appropriate data is being generated, fine-tuning of data collection code (e.g., specific decisions regarding data abstraction, selection, and reduction) and automation of the post-hoc aggregation and analysis process can be postponed until a beta version of the product has been completed. Once beta testers have access to the application, data collection code may be updated over the Internet by having users manually download updates or, better yet, by automatically downloading updates, either on a polling schedule, or each time the application or computer is re-started.

In addition to the uses cited above, such data may be used to help manage the beta testing process itself. For instance, data may be used to assess testing coverage of all critical and updated features. Furthermore, if users allow themselves to be identified, data may even be used to target questionnaires or follow-up communication with users who make extensive use of features of interest to developers, designers, and usability professionals.

Extending this approach beyond beta testing populations will introduce new social challenges, however, the proposed strategies for performing data abstraction, selection, reduction, context-capture, and evolution are capable of being adapted to larger user populations over more extended periods of time. Assuming data collection can be turned on and off automatically, development organizations may chose to implement a “snapshot”-type approach, in which data is collected intermittently, to allow data collection to be performed over extended periods of time with large numbers of users.

CHAPTER 8: Evaluation

This research has been evaluated in a number of contexts outside of the research lab, including work performed in cooperation between NYNEX Corporation and the University of Colorado at Boulder, Lockheed Martin Corporation and the University of California at Irvine, and Microsoft Corporation and the University of California at Irvine.

8.1 NYNEX Corporation: The Bridget Project

8.1.1 Overview

This work was performed in cooperation between members of the Intelligent Interfaces Group at NYNEX Corporation and the Human-Computer Communication Group at the University of Colorado at Boulder. This effort included participation in, and observation of, a development project in which developers observed users completing tasks with prototypes in order to identify mismatches between expected and actual use. This experience served to confirm the hypothesis that such an iterative design process could in fact lead to tangible design benefits, and highlighted areas in which automated support might be used to improve the process. An early prototype was constructed as a proof-of-concept that software agents could be used to automatically monitor expectations on developers' behalf and support user-developer communication without requiring developers to be physically present to observe use [Girgensohn et al. 1994].

8.1.2 Description

The Bridget system is a form-based phone service provisioning system developed by the Intelligent Interfaces group at NYNEX. It was developed with the explicit goal of providing a simpler, more usable interface for customer sales representatives who create and manage phone service accounts for small businesses [Atwood et al. 1993]. The Bridget development process was participatory and iterative in nature. Intended users, their managers, and other stakeholders all contributed knowledge about existing tasks and systems involved in phone service provisioning as part of the requirements gathering and design process. Observations of users in their existing work environment and discussions of various tasks also took place. Once a prototype of Bridget had been developed, users were asked to perform tasks with the prototype while developers observed and noted discrepancies between expected and actual use. Developers then discussed observed mismatches with users after each task. Users also interacted with Bridget on their own and voluntarily reported feedback to developers. Information gathered in this way was then incorporated into design changes resulting in revised prototypes that were iteratively refined in the same manner.

8.1.3 Results

There were two major results of this experience. First, the participatory development process outlined above led to features that might not have been introduced otherwise. For instance, observation of users interacting with early prototypes of Bridget indicated they often overlooked filling-in some of the required fields. This became apparent to users only after Bridget was unable to successfully submit a service order to

the database system, thereby requiring users to go back and complete missing fields. Also, due to the complexity of some of the service orders, many of them were actually submitted with incomplete or incorrect information and were not corrected until some time later. This could lead to delays in establishing phone service. In response to the observed problems, a built-in “checklist” was added to Bridget that used color to indicate parts of the form that still needed work. Since the checklist was embedded in the user interface, it successfully drew user’s attention without requiring extra screen real-estate or training, and proved to be very successful.

A second result was that a number of areas in which automated support might be used to improve the development process were identified. First, it was observed that much time was spent traveling between development and user sites to perform observation, and that observation itself was extremely time intensive. It was hypothesized that software agents could be developed to observe usage on developers’ behalf and initiate remote communication between developers and users when discrepancies between expected and actual use were observed. Thus, more users could be observed in parallel over more extended periods of time without requiring developers to be present at all times. A prototype agent-based system was developed at the University of Colorado at Boulder to demonstrate how such support might be implemented [Girgensohn et al. 1994]. This early prototype focused primarily on identifying sequences of user actions that did not match developers’ expectations about appropriate use. Once a mismatch was detected, developers’ expectations could be communicated to users, and users could respond with feedback if desired. User feedback could then be reported along with

contextual information about the events leading up to mismatches. The idea of capturing generic usage data in addition to reacting to specific mismatches was not considered to be of primary importance, and was thus not addressed in this early prototype.

8.2 Lockheed Martin Corporation: The GTN Scenario

8.2.1 Overview

This work was performed in cooperation between members of the Lockheed Martin C2 Integration Systems Team and the Software Group at the University of California at Irvine. Based on the NYNEX experience, and with an eye on making the approach more scalable, a second prototype was constructed at the University of California at Irvine to explore and clarify the technical issues involved in collecting usage data and user feedback on a potentially large and ongoing basis. This prototype was informally evaluated within the context of a demonstration scenario performed by Lockheed Martin C2 Integration Systems, in which the prototype was applied to a database query interface in a large-scale governmental logistics and transportation information system. Informal evidence from this experience suggested that the approach was not only suited to capturing design-related information, but that important impact assessment and effort allocation-related information could also be captured [Hilbert and Redmiles 1998a].

8.2.2 Description

In this case, independent developers at Lockheed Martin Corporation integrated a prototype developed at the University of California at Irvine into a large-scale

governmental logistics and transportation information system as part of a government-sponsored demonstration scenario. The purpose of this exercise was to demonstrate the integration of a number of government-funded research tools into a single software development scenario.⁸

The scenario itself was developed by Lockheed personnel with input from researchers supplying the research tools being integrated. The scenario involved the development of a web-based user interface to provide end users with access to a large store of transportation-related information. The engineers in the scenario were interested in verifying expectations regarding the order in which users would use the interface to specify queries. The developers used the prototype to define agents to detect when users had violated expectations regarding appropriate query specification sequence, and to notify users of the mismatch. Agent-collected data and feedback was then E-mailed to a hypothetical help desk where it was reviewed by support engineers and entered into a change request tracking system. With the help of other systems, the engineers were able to assist the help desk in providing a new release of the interface to the user community based on usage information collected from the field. Please refer to the usage scenario in Chapter 5 for more details.

8. The scenario was based on the Global Transportation Network (GTN) Project. The GTN integrates and distributes transportation-related information and acts as the central clearinghouse of transportation information for the U.S. Department of Defense. The system is intended to become the U.S. Transportation Command's primary command and control system and a fully integrated component of the Department of Defense's command and control infrastructure.

8.2.3 Results

There were two major results of this experience. First, the experience suggested that “outside” developers could successfully apply the approach, and that the effort and expertise required to author agents was not extensive, but that significant data could nonetheless be captured. The most difficult part was indicating to the demonstration team how the approach might be applied in this particular context. There were also some initial difficulties in understanding how to specify event patterns of interest. However, once these initial obstacles were overcome, the documentation was reported to have been “very helpful” and the user interface for authoring agents “simple to use”. The approach was quickly integrated by Lockheed personnel into the demonstration with only minor code insertions, and agents were easily authored and extended by Lockheed personnel to perform actions involving coordination with other systems.

A second important result is that the information collected in the demonstration scenario proved to be useful in supporting impact assessment and effort allocation decisions in addition to design decisions. The demonstration team fashioned the scenario to include a user providing feedback regarding a suggested design change. However, unsure of whether to implement the change (due to its impact on the current design, implementation, and test plans), the engineers used the usage data log to determine the impact of the suggested change on the user community at large. They decided to put the change request on hold based on how infrequently the problem had occurred in practice. This outcome had not been anticipated, since, up to this point, this research had primarily focused on capturing design-related information.⁹

8.3 Microsoft Corporation: The “Instrumented Version”

8.3.1 Overview

This work was performed in cooperation between members of Product Planning and Program Management at Microsoft Corporation and the Software Group at the University of California at Irvine. This effort included participation in, and observation of, a large-scale industrial usage data collection project performed at Microsoft Corporation in order to better understand the issues faced by development organizations wishing to capture usage information on a large-scale. This experience served to reinforce the hypothesis that automated software monitoring techniques can be used to capture useful design, impact assessment, and effort allocation information, and highlighted a number of ways in which existing techniques, as exemplified by the Microsoft approach, could be improved based on the concepts developed in this research.

8.3.2 Description

Due to a non-disclosure agreement limiting what can be published regarding this experience, it must be described in abstract terms without direct reference to the specific product that was the subject of the usage study. The application in question features over 1000 application “commands” accessible through menus and toolbars and over 300 user interface dialogs. The author of this dissertation managed the instrumentation effort in which basic usage data was collected regarding the behavior of 500 to 1000 volunteer users using the instrumented version of the application over a two month period.

9. It should be noted that while the integration and use of the prototype was in fact performed, the actual scenario itself was fabricated by Lockheed personnel to link the use of a number of research prototypes in a common story-line.

Because this was not the first time data would be collected regarding the use of this product, infrastructure already existed to capture usage information. The infrastructure essentially consisted in instrumentation code inserted directly into application code that captured data of interest and wrote it to binary disk files that were then copied to floppies and mailed to Microsoft by users after a pre-specified period of use. The sheer amount of instrumentation code already embedded in the application, the limited time available for updating instrumentation to capture information regarding new application features, and the requirement to compare the latest data against prior data collection results all conspired to make a complete overhaul of the data collection infrastructure, based upon this research, impossible. Thus, the existing data collection infrastructure was updated allowing the difficulties and limitations inherent in such an approach to be observed first-hand and compared against the ideas developed in this research.

8.3.3 Results

The results of this experience were instructive in a number of ways. First and foremost, it served to confirm the hypothesis that software monitoring techniques could indeed be used to capture strategic information useful in supporting design, impact assessment, and effort allocation decisions. Furthermore, it was an existence proof that there *are* in fact situations in which the costs associated with maintaining data collection infrastructure and performing data analysis are perceived to be outweighed by the resulting benefits, even in an extremely competitive development organization in which time-to-market is of utmost importance. The experience highlighted a number of areas in

which existing techniques, as exemplified by the Microsoft approach, might be improved based on the concepts developed in this research. The experience also provided a number of insights that have informed and refined this research.

8.3.3.1 How Practice can be Informed by this Research

Microsoft's data collection process and infrastructure suffer from a number of important limitations well-known to those within the organization who have worked on the "instrumented version". Some of the problems are due to the fact that the data collection process itself has often been implemented as an "afterthought" with the help of summer interns who have little control over the overall approach and who typically are no longer available when data has been collected and is ready for analysis. However, without underplaying the importance of these procedural problems, this subsection focuses instead on limitations inherent in the underlying technical approach.

First, because the approach relies on intrusive instrumentation of application code, evolution is a critical problem. In order to modify data collection in any way — for instance, to adjust what data is collected (i.e., selection) — the application itself must be modified, impacting the build and test processes and making independent evolution impossible. Furthermore, there is no way to flexibly map between lower level events and higher level events of interest (i.e., abstraction). Also, because the approach primarily relies on instrumentation code that has been inserted into an internal application command dispatch loop, linking command use to the user interface mechanisms used to invoke commands can be problematic at times. Next, there is no way to reduce data in context, resulting in every user or application action being recorded as a separate data

record. Finally, the approach does not allow users to provide feedback to augment automatically captured data.

There are now plans underway to implement a new data collection approach based on a user interface event monitoring model, however, support for flexible data abstraction, selection, reduction and context-capture mechanisms that can be evolved over time independently of the application and data collection infrastructure are not currently part of the plan.¹⁰

8.3.3.2 How Practice has Informed this Research

Despite the substantial observed limitations outlined above, this experience also resulted in a number of insights that have informed and refined this research. Most importantly, it helped motivate a shift from “micro” expectations regarding the behavior of single users within single sessions to “macro” expectations regarding the behavior of multiple users over multiple sessions. In the beginning, this research focused primarily on expectations regarding specific sequences of actions performed by a single user within a single session. This sort of analysis, by itself, can be challenging due to the difficulties involved in determining what exactly users are attempting to do and in anticipating all the important areas in which such mismatches *might* occur. Furthermore, once mismatches are identified, whether or not developers should take action and adjust the design is not clear. In some cases, when mismatches lead to intolerable results, developer action may be warranted in the absence of more information. However, in general, it is important to

10. The author presented several of the ideas developed here to the product planning and program management teams who expressed enthusiasm about adding such “advanced features” incrementally after the initial event monitoring infrastructure had been implemented and evaluated.

have a “big picture” view of how the application is used in order to know whether the frequency with which identified problems are observed warrants developer action.

There are two related issues at work here: the “denominator problem” and the “baseline problem”. For instance, how should developers react to the fact that the “print current page” option was used 10,000 times? The number of occurrences of any event must be compared against the number of times the event *might* have occurred. This is the denominator problem. 10,000 uses of the “print current page” option out of 11,000 uses of the print dialog paints a very different picture from 10,000 uses of the “print current page” option out of 2,000,000 uses of the print dialog. The first scenario suggests the option might reasonably be made default while the second does not. A related issue is the need for a more general baseline against which to compare specific results of data collection. For instance, if there are design issues associated with features that are much more frequently used than printing, then perhaps *those* issues should be given priority over changes to the print dialog. This is why generic usage information should be captured to provide developers with a better sense of the “big picture”.

In practice, automated usage data collection techniques are much stronger in terms of capturing indicators of the “big picture” than in identifying subtle, nuanced, and unexpected usability issues. Luckily, these strengths and weaknesses are nicely complemented by the strengths and weaknesses inherent in usability testing techniques, in which subtle usability problems may be identified through careful human observation, but in which there is little sense of the big picture. In fact, it was reported by one Microsoft usability professional that the usability team is often approached by design and

development team members with questions such as “how often do users do X?” or “how often does Y happen?”. This is obviously useful information, however, it is not information that can be collected in the usability lab. Furthermore, as has been pointed out, such information can be useful in assessing the results of, and focusing the efforts of, usability evaluations themselves.

There were also a number of realizations regarding issues involved in collecting useful usage data. For instance, some applications provide automated features that are invoked by the application on behalf of users. The invocation of such features cannot always be detected by strict user interface monitoring alone. In such cases, the application should report invocation events directly to the data collection infrastructure, and data collection code should attempt to identify user reactions to such automated feature use, such as immediate user undo's.

Another important insight was that interactions with particular interface elements, such as dialogs and secondary windows, might be analyzed separately from other interactions, and that event data may be “segmented” based on when a dialog or window is opened and then closed. This suggested a reasonable way of separating sequential information that might be useful in optimizing dialog and window layout from more general sequential information that would be costly to capture and much less likely to produce significant results in terms of potential design impact.

Another important realization was that relatively persistent information, for example, regarding user preferences and customizations, should be captured and analyzed

in different ways than other generic event and state information. Analysis of such information should address the number of sessions or user time spent under various “settings” conditions as opposed to simply counting the events associated with changing settings. For instance, it is more important to know that a user ultimately spend 95% of their time with feature X enabled than to know the number of times the user enabled and disabled the feature. Furthermore, some types of persistent information evolve over time, such as entries in a custom dictionary, and should be captured later rather than sooner in “snapshot” form as opposed to capturing the “add”, “delete”, and “modify” events associated with constructing that information, and then attempting to re-construct it later.

Finally, appending state information to event data was observed to be useful in comparing event data across multiple “modes” of use. The application under study could actually be used for a variety of purposes. Thus, capturing information about the purpose the application was serving at the time of each event occurrence was helpful in understanding different usage patterns associated with different uses of the application. Such data might be used to adjust the user interface depending on the type of use to which the application is being put, or to focus effort on features associated with particular types of use that the organization wishes to promote.

CHAPTER 9: Challenges and Future Directions

This chapter discusses some of the key practical challenges faced by organizations wishing to implement large-scale usage data collection projects, as well as important research questions and future directions.

9.1 Maintenance

Perhaps the most significant technical challenge faced by organizations wishing to capture usage data on a large and ongoing basis is maintenance of data collection code. Maintenance of data collection infrastructure is similar to standard software maintenance, and therefore is not of primary concern. However, the code that represents data selection, abstraction, and reduction decisions is more problematic, particularly if separated from application code as advocated in this dissertation. Such separation is beneficial in allowing independent evolution of application and data collection, however, it exacerbates, to a certain degree, the problem of maintaining dependencies between the two. If both are developed and maintained concurrently by a single team of developers, maintenance issues are ameliorated to an extent. However, the goal is to de-couple the process of developing the application (and data generation code) from the process of developing the data selection, abstraction, and reduction code so that they may be performed independently by potentially independent stakeholders. This means that special attention must be placed on the problem of maintaining dependencies between application and data collection code in such a way that both can evolve without unduly impacting the other.

The solution to the evolution problem presented in this dissertation primarily addresses the issue of minimizing the impact of data collection code changes on application code and users. However, a related issue arises in minimizing the impact of application code changes on data collection code. There are a number of “mappings” that must be maintained, including mappings between implementation-dependent IDs used to identify user interface and application components and human-readable names for use in data selection, abstraction, reduction, and post-hoc analysis, as well as mappings between lower-level events and higher-level events of interest.

Perhaps the most basic and problematic mappings are those which link component IDs to human-readable names for use in data selection, abstraction, reduction, and post-hoc analysis. The approach taken in this research has been to generate names automatically, and to require developers to explicitly name components of particular interest that could not be uniquely named automatically. Automatic naming is useful in allowing general data collection to be implemented quickly with minimal impact on developers. However, such automatically-generated names invariably depend, in some way, on attributes of the implementation that may evolve over time, such as the labels associated with components or the windows in which components appear. As a result, if user interface components are to be referred to explicitly in data collection code, then establishing explicit mappings either in application code or in a database that can be easily updated based on application code, while requiring extra effort, is a more robust approach.

In some cases, there may already be existing infrastructure that can be exploited for this purpose. For instance, is there already some place where links are maintained between user interface components and help files? Is there already a database used by the testing team that maps component IDs to human-readable names to support automated GUI testing? Ideally, such a database could be created and maintained through a combination of automated and manual means. For instance, data collection infrastructure may be able to traverse the entire user interface component hierarchy of the application in order to create the database initially, and later to identify broken dependencies (i.e., component additions, deletions, or modifications) not properly reflected in the database or that directly affect data collection code.¹¹

One way of minimizing the impact of changes on data collection code is to minimize the number of direct references to user interface components. A number of the agents presented in Chapter 6 avoid direct references completely and as a result are not impacted by user interface changes. Such agents are referred to as “default agents” since they can be used across multiple applications. However, in some cases, it may be useful to refer to components directly, for example, to relate the use of menu items to their parent menus or to relate the use of application commands to the various ways in which those commands can be invoked in the user interface. Such relations must be updated when new

11. Note that once such a database exists, it may also be exploited for other purposes in addition to supporting help functionality, GUI testing, and usage data collection. For instance, assuming user interface components can be linked to arbitrary sets of attributes, users could be given a function to highlight components in the interface that match certain search criteria, for example, all features associated with spelling and grammar, or all features associated with table manipulation, or all new and enhanced features added in the latest release. The results of such queries could be indicated using color and/or graphics to highlight components in the interface matching the query. For instance, small “New!” icons could be dynamically placed in menus and dialogs to help users learn about recently added functionality, as is sometimes done in Web page design.

menu items (or new menus) are added or new ways for invoking the same command (or new commands) are added. There are ways to partially automate the maintenance of such relations, particularly when changes are deletions as opposed to additions, however, the problem is challenging, particularly since not all such relations are reflected (in obvious ways) in the structure of the user interface or application code. Another important question is *where* to represent such relations. Should they be represented explicitly in application code, in data collection code (as in the prototype presented here), in a database, or computed dynamically during data collection when possible? The trade-offs involved in these differing approaches must be investigated further.

9.2 Authoring

While involving developers in data collection is likely to yield benefits in its own right, it is also desirable to make authoring of data collection code accessible to non-developers. To this end, a number of different authoring approaches might be explored, including high-level domain-specific languages, visual languages, and perhaps even “programming by demonstration” to specify patterns of interest. Related work in these areas is discussed further below.

This research has adopted what is essentially a mode-transition-based representation, in which agents are specified in terms of triggers, guards, and actions. A similar approach was used in early work on agents by Malone and colleagues [Malone et al. 1992]. The authoring mechanism is essentially a “template-based” approach, in which investigators select among pre-defined data collection options to configure agents. This

has the potential of making authoring accessible to non-programmers. Also, because agents are modular and can be configured to capture useful information without explicitly referring to user interface components, they can often be reused and adapted across multiple applications, thereby easing the authoring task, particularly when generic data collection is being performed. However, in cases where more flexibility is required, an API is provided to allow generic monitoring and data collection services to be exploited and augmented by arbitrary application-specific code implemented in a standard programming language. Other authoring approaches should also be investigated.

9.3 Privacy and Security

Privacy of user data is another important issue that must be seriously considered. First and foremost, organizational policies should be developed to place restrictions on what data may and may not be collected. One rule of thumb is to restrict data collection to data about user interactions *excluding* details about user-supplied content (e.g., document text). However, statistical information about the nature or structure of user-supplied content may be appropriate, assuming that the content itself is not captured. Furthermore, users should always be made aware that data collection is being performed. Since the proposed approach does not collect arbitrary low-level data for unspecified purposes, but rather, higher level information for specified purposes, it should be possible to justify collection, and users can be given discretionary control over what is reported. For instance, users may be given the option to review a description of the data that will be (or has been) collected, an explanation of the purposes for collection, as well as the collected data itself before allowing data to be reported. Users may also be given the option to

report data and feedback anonymously. Finally, users may be given the option to deactivate data collection altogether if privacy or security concerns are significant. In beta test situations, however, consent to allow data to be collected might be included as one of the terms of the license agreement. Finally, data encryption techniques may be used as an added measure to protect the privacy and security of user data.

9.4 User Involvement

One of the main goals of collecting usage information is to better understand users' needs, desires, likes, and dislikes in order to improve the fit between design and use. Automatically-collected information can provide important "indicators" regarding these questions, however, these indicators often require significant interpretation (due to lack of sufficient context), follow-up data collection (due to missing data), and further evaluation or dialogue with users (when necessary context and/or data cannot be captured automatically).

It is therefore important to determine when to capture data automatically, and when to enlist the help of users. Automatically-collected data is good when: (1) data is easy to capture automatically, (2) interpretation is unproblematic, (3) there is a lot of it, and (4) objective data is necessary. User feedback can be useful when: (1) data is difficult to capture automatically, (2) interpretation is problematic, (3) there isn't a lot of it, and (4) subjective data is useful. In other words, it may make sense to enlist the help of users when automatic collection is problematic, the cost to users for assisting is low, and the benefit of collection is high. It also makes sense, in many cases, to collect user feedback

while users are actively engaged in using the application, so that feedback is rich in contextual detail.

There are a number of strategies that might be employed. For instance, users may be given the option to volunteer feedback at any time. Assuming low level events are being abstracted into higher level events associated with application features, a feedback dialog could present the user with a list of recently exercised features to allow the user to easily specify which feature, if any, the user is commenting on. This helps in the process of associating feedback with application features with only a little added effort on the part of users. Another approach is to provide new affordances in the interface to allow users to communicate feedback implicitly while using the interface. For instance, dialogs could be outfitted with new variations on the “Cancel” button to differentiate between ordinary dismissals, dismissals due to the fact that the dialog had been opened “by mistake”, versus dismissals due to the fact that the dialog simply does not perform the function desired by the user. While interpretation of such data may be problematic, such an approach allows users to provide useful added information with little added effort. Finally, it may make sense, in some cases, to explicitly request user feedback under particular conditions, for example, when the user uses a new feature of particular interest or when users violate expected usage in significant ways. However, the benefit of capturing such data must be weighed against the cost of interrupting and potentially annoying users. One strategy would be to only request feedback the *first time* such conditions are met, and present users with the opportunity to disable future requests for

feedback altogether. There are a number of practical and research problems that must be addressed in this area.

9.5 Post-Hoc Analysis

Finally, this research has focused primarily on problems associated with data collection. Less attention has been paid to problems associated with aggregating and analyzing data once it has been captured. At Microsoft Corporation, post-hoc analysis was performed using standard relational database functionality. It would be interesting to explore other possibilities, including the use of more sophisticated analysis techniques. For instance, some of the hybrid neural network and rule-based techniques used to detect credit card fraud might be applicable in detecting interesting changes in application usage patterns over time. New unforeseen challenges may also arise once data is captured over arbitrarily extended periods of time with arbitrarily shifting user populations.¹²

9.6 Possible Areas of Cross-Pollination

There are a number of related areas that have been explored, both in academia and industry, that have the potential of providing useful insights that could benefit this research. Likewise, some of these areas may benefit from insights gained as a result of this work.

12. Furthermore, new challenges may also arise in analyzing the use of collaborative applications with multiple, concurrent users as well as applications using substantially new interaction metaphors.

A number of researchers and practitioners have addressed related issues in capturing and evaluating event data in the realm of software testing and debugging:

- Work in distributed event monitoring, e.g., GEM [Mansouri-Samani and Sloman 1991], and model-based testing and debugging, e.g., EBBA [Bates 1995] and TSL [Rosenblum 1991], have addressed a number of problems in the specification and detection of composite events and the use of context in interpreting the significance of events. The languages and experience that have come out of this work are likely to provide useful insights that might be applied to the problem of extracting usage information from user interaction data.
- Automated user interface testing techniques, e.g. Mercury Interactive WinRunner™ [Mercury Interactive 1998] and Sun Microsystems JavaStar™ [Sun Microsystems 1998], are faced with the problem of robustly identifying user interface components in the face of user interface change, and evaluating events against specifications of expected behavior in test scripts. The same problem is faced in maintaining mappings between user interface components and higher-level specifications of event and state information of interest in usage data collection.
- Monitoring of application programmatic interfaces (APIs), e.g., Hewlett Packard's Application Response-time Measurement API [Hewlett Packard 1998], addresses the problem of monitoring API usage to help software developers evaluate the fit between the design of an API and how it is actually used. Insights might be shared between investigators working in this area and investigators working in the area of monitoring interactive application usage to evaluate the fit between the design of the application itself and how it is actually used.
- Internet-based application monitoring systems, e.g., Aqueduct AppScope™ [Aqueduct Software 1998] and Full Circle TalkBack™ [Full Circle Software 1998], have begun to address issues of collecting information regarding application crashes on a potentially large and ongoing basis over the Internet. The techniques developed to make this practical for crash monitoring could also be applicable in the domain of large-scale, ongoing usage data collection over the Internet.

A number of researchers have addressed problems in the area of mapping between lower level events and higher level events of interest:

- Work in the area of event histories, e.g., [Kosbie and Myers 1994], and undo mechanisms has addressed issues involved in grouping lower level user interface events into more meaningful units from the point of view of users' tasks. Insights gained from this work, and the actual event representations used to support undo mechanisms, might be exploited to capture events at higher levels of abstraction than are typically available at the window system level.
- Work in the area of user modeling [User Modeling 1998] is faced with the problem of inferring users' tasks and goals based on user background, interaction history, and current context in order to enhance human-computer interaction. The techniques developed in this area, which range from rule-based to statistically-oriented machine-learning techniques, may eventually be harnessed to infer higher level events from lower level events in support of usage data collection. However, user modeling work is perhaps more likely to benefit from the techniques explored in this research than vice versa.
- Work in the area of programming by demonstration [Cypher 1994] and plan recognition and assisted completion [Cypher 1991] also addresses problems involved in inferring user intent based on lower level interactions. This work has shown that such inference is feasible in at least some structured and limited domains, and programming by demonstration appears to be a desirable method for specifying expected or unexpected patterns of events for sequence detection and comparison purposes.
- Layered protocol models of interaction, e.g., [Nielsen 1986; Taylor 1988a & 1988b], allow human-computer interactions to be modeled at multiple levels of abstraction. Such techniques may be useful in mapping between lower level events and higher level events of interest. Command language grammars (CLG's) [Moran 1981] and task-action grammars (TAG's) [Payne and Green 1986] are other potentially useful modeling techniques for specifying relationships between human-computer interactions and users' tasks and goals.

Work in the area of automated discovery and validation of patterns in large corpora of event data may also provide valuable insights:

- Data mining techniques for discovering association rules, sequential patterns, and time-series similarities in large data sets [Agrawal et al. 1996] may be applicable in uncovering patterns relevant to investigators interested in evaluating application usage.

- The process discovery techniques investigated by [Cook and Wolf 1996] provide insights into problems involved in automatically generating models to characterize the sequential structure of event traces, and the process validation techniques investigated by [Cook and Wolf 1997] provide insights into problems involved in comparing traces of events against models of expected behavior.

Finally, there are numerous domains in which event monitoring has been used as a means of identifying, and in some cases, diagnosing and repairing breakdowns in the operation of complex systems. For example:

- Network and enterprise management tools for automating network and enterprise application administration, e.g., TIBCO Hawk™ [TIBCO 1998].
- Product condition monitoring, e.g., high-end photocopiers or medical devices that report data back to equipment manufacturers to allow performance, failures, and maintenance issues to be tracked remotely [Lee 1996].

CHAPTER 10: Conclusions

10.1 Conclusions

This dissertation has demonstrated technical and methodological solutions to enable usage- and usability-related information of much higher quality than currently available from beta tests to be collected on a much larger scale than currently possible in usability tests. Such data is complementary in that it can be used to address the impact assessment and effort allocation problems (described in the Introduction) in addition to evaluating and improving the fit between application design and use.

The principles and techniques underlying this work have been subjected to a number of evaluative activities including: (1) the development of two independent research prototypes at the University of Colorado and the University of California, (2) the incorporation of one prototype by independent third party developers as part of an integrated demonstration scenario performed by Lockheed-Martin Corporation, and (3) observation and participation in two industrial development projects, conducted at NYNEX and Microsoft Corporations, in which developers sought to improve the application development process based on usage data and user feedback.

These experiences all contributed evidence to support the hypothesis that automated software monitoring techniques could indeed be used to capture usage- and usability-related information useful in making design, impact assessment, and effort allocation decisions in the development of interactive systems. A third and final prototype

was then constructed — based on the experience and insights gained from the first two prototypes, the survey of related work, and the Microsoft experience — which served as a basis for demonstrating solutions to the abstraction, selection, reduction, context, and evolution problems within a single data collection architecture, thereby illustrating how usage- and usability-related information might be captured on a potentially large and ongoing basis.

10.2 Summary of Contributions

In summary, the main contributions of this work include:

- Identification of five key problems limiting the scalability and data quality of existing techniques for extracting usability information from user interface events (Chapter 4).
- A layered architecture that separates data collection code from generic data collection services, which in turn are separated from generic event and state monitoring services, which in turn are separated from details of the underlying component model in which applications are developed (Chapter 5).
- Solutions to the abstraction, selection, reduction, context, and evolution problems demonstrated within a single data collection architecture (Chapter 6).
- A reference architecture to provide design guidance regarding key components and relationships required to support large-scale data collection (Chapter 6).
- A theory motivating the significance of usage expectations in development and the importance of collecting usage information (Chapter 7).
- Methodological guidance regarding data collection, analysis, interpretation, and process integration (Chapter 7).

10.3 Other Potential Applications

Chapter 9 discusses a number of areas in which the results of this research might be applied to, or enriched by, work in other domains. This section simply emphasizes three key directions in which this work might be profitably extended.

While this research has focused on capturing data regarding the use of window-based interactive applications, the principles, techniques, and methodology underlying the approach can be generalized to the problem of capturing data about the operation of arbitrary software systems in which: (a) event and state information is generated as a natural by-product of system operation, (b) low-level data must be related to higher level concepts of interest, (c) available information exceeds that which can practically be collected, and (d) data collection needs evolve over time more quickly than the application. This would appear to apply to a large number of software systems, particularly those implemented in a component-based architectural style.

Furthermore, the techniques described here might also be applied to solve other problems not directly related to collecting usage information for development purposes. For instance, long-term data about user or users' actions might be captured in a similar way to support adaptive UI and application behavior as well as "smarter" delivery of help, suggestions, and assistance. Current approaches to this problem often rely only on users' most recent actions, and typically do not have a long-term picture of use, particularly not involving multiple users.

Finally, user-interface monitoring agents, such as those presented here, might also be explored as a means for supporting the extension, customization, and integration of interactive applications. For instance, agents might be used to embed organizational knowledge and rules into commercial off-the-shelf software based on particular user interactions of interest. Agents might also be used to support workflow integration and monitoring by abstracting low-level user interactions with interactive applications into higher-level events useful in triggering and monitoring workflow progress.

REFERENCES

1. Abbott, A. A Primer on sequence methods. *Organization Science*, Vol.4, 1990.
2. Agrawal, R., Arning, A., Bollinger, T., Mehta, M., Shafer, J., Srikant, R. The Quest data mining system. In *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*. 1996.
3. Aho, A.V., Kernighan, B.W., and Weinberger, P.J. *The AWK programming language*. Addison-Wesley, Reading, MA. 1988.
4. Allison, P.D. and Liker, J.K. Analyzing sequential categorical data on dyadic interaction: A comment on Gottman. *Psychological Bulletin*, 2, 1987.
5. Aqueduct Software. AppScope Web pages. URL: <http://www.aqueduct.com/>. 1998.
6. Atwood, M.E., Burns, B., Girgensohn, A., Zimmermann, B. "Dynamic Forms: Intelligent Interfaces to Support Customer Interactions", Technical Memorandum TM 93-0048, NYNEX Science and Technology, White Plains, NY, 1993.
7. Badre, A.N. and Santos, P.J. CHIME: A knowledge-based computer-human interaction monitoring engine. Tech Report GIT-GVU-91-06. 1991a.
8. Badre, A.N. and Santos, P.J. A knowledge-based system for capturing human-computer interaction events: CHIME. Tech Report GIT-GVU-91-21. 1991b.
9. Badre, A.N., Guzdial, M., Hudson, S.E., and Santos, P.J. A user interface evaluation environment using synchronized video, visualizations, and event trace data. *Journal of Software Quality*, Vol. 4, 1995.
10. Baecker, R.M, Grudin, J., Buxton, W.A.S., and Greenberg, S. (Eds.). *Readings in Human-Computer Interaction: Toward the Year 2000*. Morgan Kaufmann, San Mateo, CA, 1995.
11. Balbo, S. EMA: Automatic analysis mechanism for the ergonomic evaluation of user interfaces. CSIRO Technical report. 1996.
12. Bates, P.C. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, Vol. 13, No. 1, 1995.

13. Bellotti, V. A framework for assessing applicability of HCI techniques. In Proceedings of INTERACT'90. 1990.
14. Buxton, W., Lamb, M., Sheman, D., and Smith, K. Towards a comprehensive user interface management system. In Proceedings of SIGGRAPH'83. 1983.
15. Castillo, J.C. and Hartson, H.R. Remote usability evaluation Web pages. URL: http://hci.ise.vt.edu/~josec/remote_eval/. 1997.
16. Chang, E. and Dillon, T.S. Automated usability testing. In Proceedings of INTERACT'97.
17. Chen, J. Providing intrinsic support for user interface monitoring. In Proceedings of INTERACT'90. 1990.
18. Cook, J.E. and Wolf, A.L. Software process validation: quantitatively measuring the correspondence of a process to a model. Tech Report CU-CS-840-97, Department of Computer Science, University of Colorado at Boulder. 1997.
19. Cook, J.E. and Wolf, A.L. Toward metrics for process validation. In Proceedings of ICSP'94. 1994.
20. Cook, J.E., and Wolf, A.L. Automating process discovery through event-data analysis. In Proceedings of ICSE'95. 1995.
21. Cook, R., Kay, J., Ryan, G., and Thomas, R.C. A toolkit for appraising the long-term usability of a text editor. *Software Quality Journal*, Vol. 4, No. 2, 1995.
22. Cuomo, D.L. Understanding the applicability of sequential data analysis techniques for analysing usability data. Nielsen, J. (Ed.). *Usability Laboratories Special Issue of Behaviour and Information Technology*, Vol. 13, No. 1 & 2, 1994.
23. Cusumano, M.A. and Selby, R.W. *Microsoft Secrets: How the world's most powerful software company creates technology, shapes markets, and manages people*. The Free Press, New York NY, 1995.
24. Cypher, A. (Ed.). *Watch what I do: programming by demonstration*. MIT Press, Cambridge MA, 1993.
25. Cypher, Allen. Eager: programming repetitive tasks by example. In Proceedings of CHI'91. 1991.
26. Doubleday, A., Ryan, M., Springett, M., and Sutcliffe, A. A comparison of usability techniques for evaluating design. In Proceedings of DIS'97. 1997.

27. Elgin, B. Subjective usability feedback from the field over a network. In Proceedings of CHI'95. 1995.
28. ErgoLight Usability Software. Operation Recording Suite (EORS) and Usability Validation Suite (EUVS) Web pages. URL: <http://www.ergolight.co.il/>. 1998.
29. Faraone, S.V. and Dorfman, D.D. Lag sequential analysis: Robust statistical methods. Psychological Bulletin, 101, 1987.
30. Feather, M.S., Narayanaswamy, K., Cohen, D., and Fickas, S. Automatic monitoring of software requirements. Research Demonstration in Proceedings of ICSE'97. 1997.
31. Fickas, S. and Feather, M.S. Requirements monitoring in dynamic environments. IEEE International Symposium on Requirements Engineering. 1995.
32. Finlay, J. and Harrison, M. Pattern recognition and interaction models. In Proceedings of INTERACT'90. 1990.
33. Fisher, C. Advancing the study of programming with computer-aided protocol analysis. In Olson, G., Soloway, E., and Sheppard, S. (Eds.). Empirical Studies of Programmers, 1987 Workshop. Ablex, Norwood, NJ, 1987.
34. Fisher, C. and Sanderson, P. Exploratory sequential data analysis: exploring continuous observational data. Interactions, Vol.3, No. 2, ACM Press, Mar. 1996.
35. Fisher, C. Protocol Analyst's Workbench: Design and evaluation of computer-aided protocol analysis. Unpublished PhD thesis, Carnegie Mellon University, Department of Psychology, Pittsburgh, PA, 1991.
36. Fitts, P.M. Perceptual motor skill learning. In Melton, A.W. (Ed.). Categories of human learning. Academic Press, New York, NY, 1964. S.J.
37. Full Circle Software. Talkback Web pages. URL: <http://www.fullcirclesoftware.com/>. 1998.
38. Girgensohn, A., Redmiles, D.F., and Shipman, F.M. III. Agent-Based Support for Communication between Developers and Users in Software Design. In Proceedings of the Knowledge-Based Software Engineering Conference '94. Monterey, CA, USA, 1994.
39. Gottman, J.M. and Roy, A.K. Sequential analysis: A guide for behavioral researchers. Cambridge University Press, Cambridge, England, 1990.
40. Grudin, J. Utility and usability: Research issues and development contexts". Interacting with computers, Vol. 4, No. 2, 1992.

41. Guzdial, M, Walton, C., Konemann, M., and Soloway, E. Characterizing process change using log file data. Tech Report GIT-GVU-93-44. 1993.
42. Guzdial, M. Deriving software usage patterns from log files. Tech Report GIT-GVU-93-41. 1993.
43. Guzdial, M., Santos, P., Badre, A., Hudson, S., and Gray, M. Analyzing and visualizing log files: A computational science of usability. Presented at HCI Consortium Workshop, 1994.
44. Hartson, H.R., Castillo, J.C., Kelso, J., and Neale, W.C. Remote evaluation: the network as an extension of the usability laboratory. In Proceedings of CHI'96. 1996.
45. Hewlett Packard. Application response measurement API Web pages. URL: <http://www.hp.com/openview/rpm/arm/>. 1998.
46. Hilbert, D.M., Robbins, J.E., and Redmiles, D.F., Supporting ongoing user involvement in development via expectation-driven event monitoring. Tech Report UCI-ICS-97-19, Department of Information and Computer Science, University of California, Irvine. 1997.
47. Hilbert, D.M. and Redmiles, D.F. An approach to large-scale collection of application usage data over the Internet. In Proceedings of ICSE'98. 1998a.
48. Hilbert, D.M. and Redmiles, D.F. Agents for collecting application usage data over the Internet. In Proceedings of Autonomous Agents'98. 1998b.
49. Hirschberg, D.S. A linear space algorithm for computing maximal common subsequences. Communications of the ACM, Vol. 18, 1975.
50. Hoiem, D.E. and Sullivan, K.D. Designing and using integrated data collection and analysis tools: challenges and considerations. Nielsen, J. (Ed.). Usability Laboratories Special Issue of Behaviour and Information Technology, Vol. 13, No. 1 & 2, 1994.
51. Hoppe, H.U. Task-oriented parsing: A diagnostic method to be used by adaptive systems. In Proceedings of CHI'88. 1988.
52. John, B.E. and Kieras, D.E. Using GOMS for user interface design and evaluation: which technique? ACM Transactions on Computer-Human Interaction, Vol. 3, No. 4, 1996.
53. John, B.E. and Kieras, D.E. The GOMS family of user interface analysis techniques: comparison and contrast. ACM Transactions on Computer-Human Interaction, Vol. 3, No. 4, 1996.

54. Kay, J. and Thomas, R.C. Studying long-term system use. *Communications of the ACM*, Vol. 38, No. 7, 1995.
55. Kosbie, D.S. and Myers, B.A. Extending programming by demonstration with hierarchical event histories. In *Proceedings of East-West Human Computer Interaction'94*. 1994.
56. Krishnamurthy, B and Rosenblum, D.S. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, Vol. 21, No. 10, 1995.
57. Lecerof, A. and Paterno, F. Automatic support for usability evaluation. *IEEE Transactions on Software Engineering*, Vol. 24, No. 10, 1998.
58. Lee, B. Remote diagnostics and product lifecycle monitoring for high-end appliances: a new Internet-based approach utilizing intelligent software agents. In *Proceedings of the Appliance Manufacturer Conference*. 1996.
59. Lewis, C., Polson, P., Wharton, C., Rieman, J. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces, In *Proceedings of CHI '90*. 1990.
60. Lowgren, J. and Nordqvist, T. Knowledge-based evaluation as design support for graphical user interfaces. In *Proceedings of CHI'92*. 1992.
61. M. Helander (Ed.). *Handbook of human-computer interaction*. Elsevier Science Publishers B.V. (North Holland), 1998.
62. Macleod, M. and Rengger, R. The development of DRUM: A software tool for video-assisted usability evaluation. In *Proceedings of HCI'93*. 1993.
63. Malone, T.W., Lai, K.Y., and Fry, C. "Experiments with Oval: A radically tailorable tool for cooperative work". In *Proceedings of CSCW'92*. 1992.
64. Mansouri-Samani, M. and Sloman, M. GEM: A generalised event monitoring language for distributed systems. *IEE/BCS/IOP Distributed Systems Engineering Journal*, Vol 4, No 2, 1997.
65. Mercury Interactive. WinRunner and XRunner Web pages. URL: <http://www.merc-int.com/>. 1998.
66. Moran, T. P. The command language grammar: a representation for the user interface of interactive computer systems. *International Journal of Man-Machine Studies*, 15, 1981.
67. Neal, A.S. and Simons, R.M. Playback: A method for evaluating the usability of software and its documentation. In *Proceedings of CHI'83*. 1983.

68. Nielsen, J. A virtual protocol model for computer-human interaction. *International Journal of Man-Machine Studies*, 24, 1986.
69. Nielsen, J. *Usability engineering*. Academic Press/AP Professional, Cambridge, MA, 1993.
70. Nielsen, J. and Mack, R.L., eds. *Usability Inspection Methods*. John Wiley & Sons, New York, NY, 1994.
71. Olsen, D.R. and Halversen, B.W. Interface usage measurements in a user interface management system. In *Proceedings of UIST'88*. 1988.
72. Olson, G.M., Herbsleb, J.D., and Rueter, H.H. Characterizing the sequential structure of interactive behaviors through statistical and grammatical techniques. *Human-Computer Interaction Special Issue on ESDA*, Vol.9, 1994.
73. Payne, T.R.G. Green. Task-action grammars: A model of the mental representation of task languages. *Human-Computer Interaction*, Vol. 2, 1986.
74. Pentland, B.T. A grammatical model of organizational routines. *Administrative Science Quarterly*. 1994.
75. Pentland, B.T. Grammatical models of organizational processes. *Organization Science*. 1994.
76. Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., and Carey, T. *Human-computer interaction*. Addison-Wesley, Wokingham, UK, 1994.
77. Rosenblum, D.S. Specifying concurrent systems with TSL. *IEEE Software*, Vol. 8, No. 3, 1991.
78. Sackett. G.P. *Observing behavior* (Vol. 2). University Park Press, Baltimore, MD, 1978.
79. Sanderson, P.M. and Fisher, C. Exploratory sequential data analysis: foundations. *Human-Computer Interaction Special Issue on ESDA*, Vol. 9, 1994.
80. Sanderson, P.M., Scott, J.J.P., Johnston, T., Mainzer, J., Watanabe, L.M., and James, J.M. MacSHAPA and the enterprise of Exploratory Sequential Data Analysis (ESDA). *International Journal of Human-Computer Studies*, Vol. 41, 1994.
81. Santos, P.J. and Badre, A.N. Automatic chunk detection in human-computer interaction. In *Proceedings of Workshop on Advanced Visual Interfaces AVI '94*. Also available as Tech Report GIT-GVU-94-4. 1994.

82. Schiele, F. and Hoppe, H.U. Inferring task structures from interaction protocols. In Proceedings of INTERACT'90. 1990.
83. Selby, R.W., Porter, A.A., Schmidt, D.C., and Berney, J. Metric-driven analysis and feedback systems for enabling empirically guided software development. In Proceedings of ICSE'91. 1991.
84. Siochi, A.C. and Ehrich, R.W. Computer analysis of user interfaces based on repetition in transcripts of user sessions. ACM Transactions on Information Systems. 1991.
85. Siochi, A.C. and Hix, D. A study of computer-supported user interface evaluation using maximal repeating pattern analysis. In Proceedings of CHI'91. 1991.
86. Smilowitz, E.D., Darnell, M.J., and Benson, A.E. Are we overlooking some usability testing methods? A comparison of lab, beta, and forum tests. Nielsen, J. (Ed.). Usability Laboratories Special Issue of Behaviour and Information Technology, Vol. 13, No. 1 & 2, 1994.
87. Sun Microsystems. JavaStar Web pages. URL: <http://www.sun.com/suntest/>. 1998.
88. Swallow, J., Hameluck, D., and Carey, T. User interface instrumentation for usability analysis: A case study. In Proceedings of Cascon'97. 1997.
89. Sweeny, M., Maguire, M., and Shackel, B. Evaluating human-computer interaction: A framework. International Journal of Man-Machine Studies, Vol.38, 1993.
90. Taylor, M.M. Layered protocols for computer-human dialogue I: Principles. International Journal of Man-Machine Studies, 28, 1988a.
91. Taylor, M.M. Layered protocols for computer-human dialogue II: Some practical issues. International Journal of Man-Machine Studies, 28, 1988b.
92. Taylor, R.N. and Coutaz, J. Workshop on Software Engineering and Human-Computer Interaction: Joint Research Issues. In Proceedings of ICSE'94. 1994.
93. TIBCO. HAWK Enterprise Monitor Web pages. URL: <http://www.tibco.com/products/products.html>. 1998.
94. Uehling, D.L. and Wolf, K. User Action Graphing Effort (UsAGE). In Proceedings of CHI'95. 1995.
95. User Modeling Inc. User Modeling Inc. Web pages. URL: <http://um.org/>. 1998.

96. Weiler, P. Software for the usability lab: a sampling of current tools. In Proceedings of INTERCHI'93. 1993.
97. Whitefield, A., Wilson, F., and Dowell, J. A framework for human factors evaluation. Behaviour and Information Technology, Vol. 10, No. 1, 1991.
98. Wolf, A.L. and Rosenblum, D.S. A Study in Software Process Data Capture and Analysis. In Proceedings of the Second International Conference on Software Process, 1993.