Collecting Usage Data and User Feedback on a Large Scale to Inform Software Development

DAVID M. HILBERT and DAVID F. REDMILES
University of California, Irvine

The two most commonly used techniques for evaluating the fit between application design and use — namely, usability testing and beta testing with user feedback — 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). They also fail to provide developers with an adequate basis for: (1) assessing the impact of suspected problems (and proposed solutions) on users at large, and (2) deciding where to focus scarce development and evaluation resources to maximize the benefit for users at large. This article describes an approach to usage data and user feedback collection that addresses these limitations to provide developers with a complementary source of usage- and usability-related information. This research has been subjected to a number of evaluative activities including: (1) the development of three research prototypes at NYNEX Corporation, 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, at NYNEX and Microsoft Corporations, in which developers sought to improve the application development process based on usage data and user feedback. The proposed approach 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.

Categories and Subject Descriptors: H.5.2 **[Information Interfaces and Presentation]**: User Interfaces—Evaluation/methodology

General Terms: Human factors, Measurement, Experimentation

Additional Key Words and Phrases: Usability testing, User interface event monitoring, Human-computer interaction

This work was supported by the National Science Foundation, grant number CCR-9624846, and the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.
Authors' address: Department of Information and Computer Science, University of California, Irvine, Irvine, CA 92697-3425; email: {dhilbert,redmiles}@ics.uci.edu.

1. INTRODUCTION

Involving end users in the development of interactive systems increases the likelihood those systems will be useful and usable [Gould and Lewis 1983, 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.

This article describes an approach to usage data and user feedback collection that addresses these limitations to provide developers with a complementary source of usage- and usability-related information. The proposed approach 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 principles and techniques underlying this work have been subjected to a number of evaluative activities including: (1) the development of three research prototypes at NYNEX Corporation, 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, at NYNEX and Microsoft Corporations, in which developers sought to improve the application development process based on usage data and user feedback.

Each of these experiences contributed evidence to support the hypothesis that automated software monitoring techniques can indeed be used to capture usage- and usability-related information useful in supporting design, impact assessment, and effort allocation decisions in the development of interactive systems. The latest prototype was constructed based on the experience and insights gained from the first two prototypes, an in-depth survey of related work, and the Microsoft experience, and serves as a basis for demonstrating solutions (presented in Section 4) to five key problems limiting the scalability and data quality of existing automated data collection techniques (presented in Section 2).

2. PROBLEMS

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?

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 to: lack of proper incentives, a paradoxical relationship between user performance and subjective reports, lack of knowledge regarding expected use, 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 often 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 that 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 often 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 may fail to be reported while potentially misleading or unfounded feedback from beta testers having no difficulties may 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 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 knowledge regarding expected use, and lack of details in reported data. Furthermore, without more information regarding the frequency of problem occurrences (or the frequency

1.These examples were taken from an electronic discussion group for usability research and design professionals to discuss issues related to usability evaluations.

with which features associated with reported problems are used) the impact assessment and effort allocation problems continue to remain unresolved.

2.3 Early Attempts to Exploit the Internet

A number of researchers 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 (such as Microsoft NetMeeting [Summers 1998]) 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 still 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 knowledge regarding expected use.

2.4 Automated Techniques

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 that might be used for this purpose [Chen 1990, Badre and Santos 1991, Weiler 1993, Hoiem and Sullivan 1994, Badre et al. 1995, Cook et al. 1995, Kay and Thomas 1995, ErgoLight Usability Software 1998, Lecerof and Paterno 1998]. 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 software component 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.

Figure 1 indicates the extent to which existing techniques address these problems. A small ‘x’ indicates limited support while a large ‘X’ indicates more extensive support. “instr.” indicates that the problem can be addressed, but only by modifying hard-coded instrumentation embedded in application code.

Technique	**Abstraction Problem**	**Selection Problem**	**Reduction Problem**	**Context Problem**	**Evolution Problem**
Chen 1990		X		x	X
Badre & Santos 1991	x	X			X
Weiler 1993		x			
Hoiem & Sullivan 1994		x			
Badre et al. 1995	x	X			X
Cook et al. 1995; Kay & Thomas 1995	instr.	instr.	instr.	instr.	
ErgoLight 1998	X				
Lecerof & Paterno 1998	X				

Figure 1. Existing data collection approaches and support for problems

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

3. APPROACH

We propose a semi-automated, Internet-mediated approach to large-scale usage data and user feedback collection that addresses these limitations. The approach 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.

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. We also conjecture 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.

3.1 Theory of Expectations

This section discusses, in more detail, a simple "theory of expectations" that provides the basis 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 present a brief overview of the proposed approach, a usage scenario to illustrate how the approach might be applied in practice, and a summary of key technical details underlying the implementation of the approach.

3.2 Technical Overview

The proposed approach 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.

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.

Because the means for accessing event and state information varies 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.[2] Furthermore, because agents are often reusable, or at least adaptable, across applications, the notion of *default agents* was

2.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 JavaBeans™, might be implemented.

introduced to allow higher-level generic data collection services to be reused across applications.

Figure 2 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:

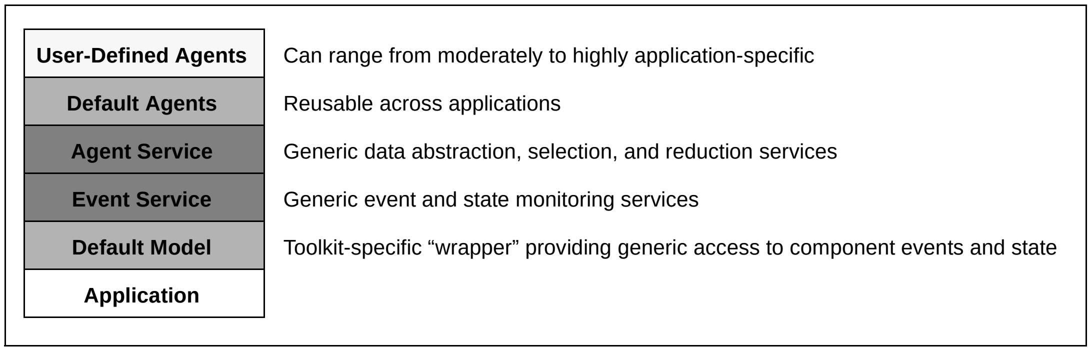

User-Defined Agents
Can range from moderately to highly application-specific
Default Agents
Reusable across applications
Agent Service
Generic data abstraction, selection, and reduction services
Event Service
Generic event and state monitoring services
Default Model
Toolkit-specific “wrapper” providing generic access to component events and state
Application

Figure 2. Basic services

Before going into more detail regarding the implementation of the approach, the following section presents a usage scenario to illustrate how the approach might be applied in practice.

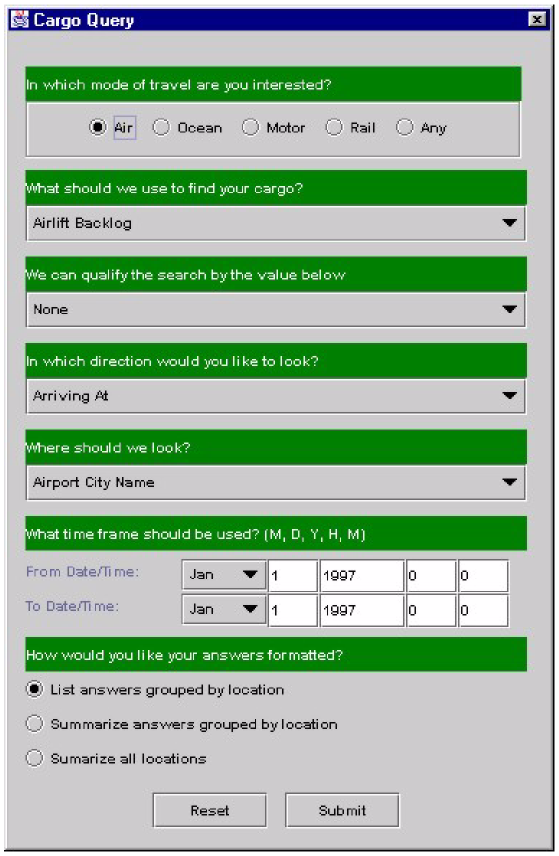
3.3 Usage Scenario

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 is 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 3 shows the prototype interface.


Cargo Query

In which mode of travel are you interested?

[✓ Air] Ocean Motor Rail Any

What should we use to find your cargo?

Airlift Backlog

We can qualify the search by the value below

None

In which direction would you like to look?

Arriving At

Where should we look?

Airport City Name

What time frame should be used? (M, D, Y, H, M)

From Date/Time:	Jan	1	1997	0	0
To Date/Time:	Jan	1	1997	0	0

How would you like your answers formatted?

[✓ List answers grouped by location]

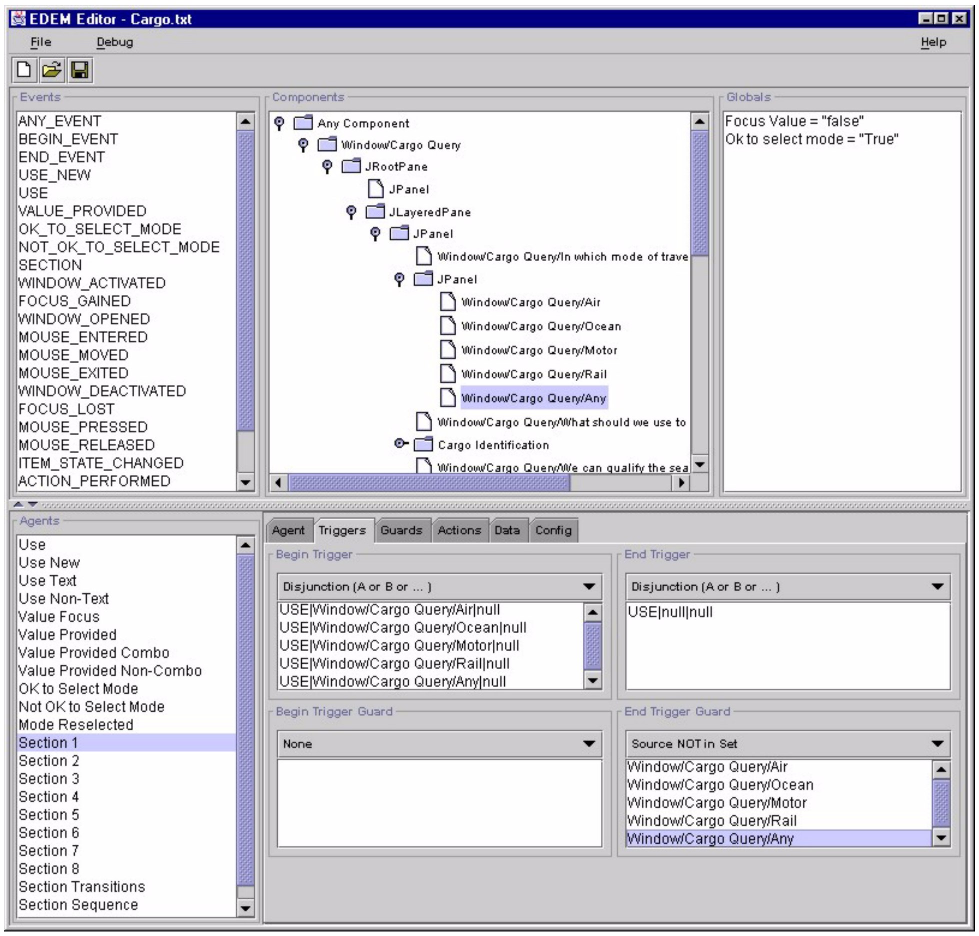
Summarize answers grouped by location

Sumarize all locations

Figure 3. 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 4 is a screenshot of the agent authoring user interface provided by the agent service to allow agents to be defined without writing code. In Figure 4, 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 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.



Events:
ANY_EVENT
BEGIN_EVENT
END_EVENT
USE_NEW
USE
VALUE_PROVIDED
OK_TO_SELECT_MODE
NOT_OK_TO_SELECT_MODE
SECTION
WINDOW_ACTIVATED
FOCUS_GAINED
WINDOW_OPENED
MOUSE_ENTERED
MOUSE_MOVED
MOUSE_EXITED
WINDOW_DEACTIVATED
FOCUS_LOST
MOUSE_PRESSED
MOUSE_RELEASED
ITEM_STATE_CHANGED
ACTION_PERFORMED

Components:
Any Component
Window/Cargo Query
JRootPane
JPanel
JLayeredPane
JPanel
Window/Cargo Query/In which mode of trave
JPanel
Window/Cargo Query/Air
Window/Cargo Query/Ocean
Window/Cargo Query/Motor
Window/Cargo Query/Rail
Window/Cargo Query/Any
Window/Cargo Query/What should we use to
Cargo Identification
Window/Cargo Query/We can qualify the sea

Globals:
Focus Value = "false"
Ok to select mode = "True"

Agents:
Use
Use New
Use Text
Use Non-Text
Value Focus
Value Provided
Value Provided Combo
Value Provided Non-Combo
OK to Select Mode
Not OK to Select Mode
Mode Reselected
Section 1
Section 2
Section 3
Section 4
Section 5
Section 6
Section 7
Section 8
Section Transitions
Section Sequence

Begin Trigger: Disjunction (A or B or ...)
USE|Window/Cargo Query/Air|null
USE|Window/Cargo Query/Ocean|null
USE|Window/Cargo Query/Motor|null
USE|Window/Cargo Query/Rail|null
USE|Window/Cargo Query/Any|null

End Trigger: Disjunction (A or B or ...)
USE|null|null

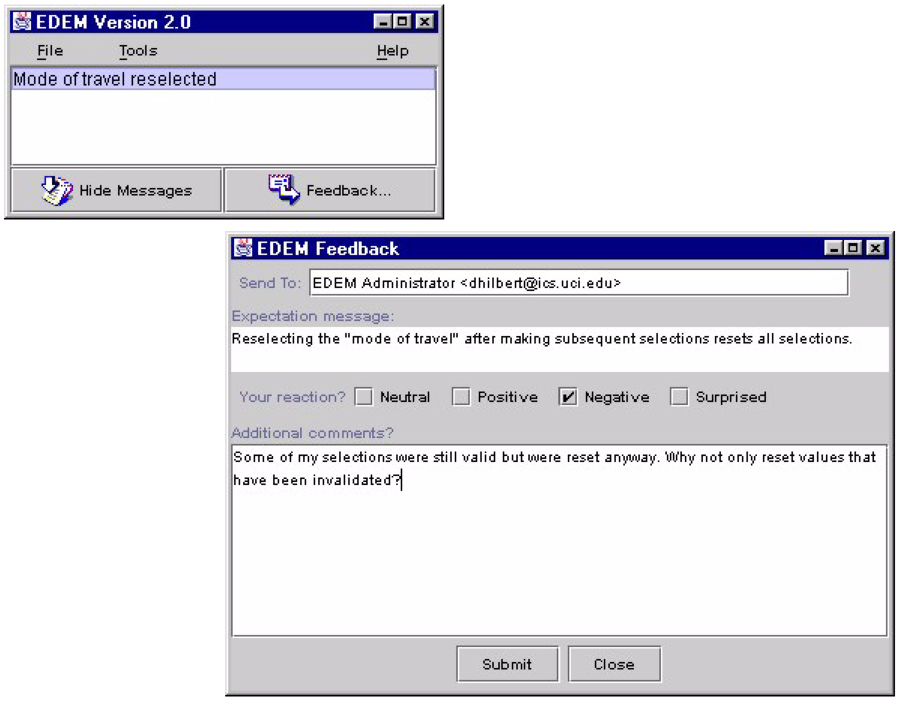
Begin Trigger Guard: None

End Trigger Guard: Source NOT in Set
Window/Cargo Query/Air
Window/Cargo Query/Ocean
Window/Cargo Query/Motor
Window/Cargo Query/Rail
Window/Cargo Query/Any

Figure 4. Agent authoring interface

In this case, the engineers decided to configure an agent to unobtrusively notify users when it detected behavior in violation of developers' expectations (Figure 5). 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 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,



Mode of travel reselected

Send To: EDEM Administrator <dhilbert@ics.uci.edu>

Expectation message:
Reselecting the "mode of travel" after making subsequent selections resets all selections.

Your reaction? Neutral Positive [✓ Negative] Surprised

Additional comments?
Some of my selections were still valid but were reset anyway. Why not only reset values that have been invalidated?

Figure 5. Agent notification and user feedback

let us assume that one of the users actually responds to the agent with exactly this suggestion (Figure 5). 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 this type of empirical data in 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.

3.4 Technical Details

3.4.1 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 a setName() method provided by the event service API.

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 1: 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.

Event dispatching is achieved using a hash table 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 hash table 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.

3.4.2 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 6. 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 7. 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 include 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 occurring between 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.

Figure 8 summarizes the algorithm governing agent execution:

```
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; 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; time = end - begin
        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 8. Agent algorithm

4. SOLUTIONS

Section 2 concluded with the identification of five key problems limiting the scalability and data quality of existing techniques for extracting usage- and usability-related information from user interface events. In the next subsection we present a number of example agents to demonstrate solutions to the first four problems (abstraction, selection, reduction, and context). In the following subsection we present a reference architecture to illustrate how independent evolution is achieved to address the fifth problem (evolution). The examples below comprise only a subset of the ways in which the proposed approach might be used to address these problems. A full treatment is beyond the scope if this article.

4.1 The Abstraction, Selection, Reduction, and Context Problems

The following examples illustrate how agents might be used to capture information regarding the use of a simple word processing application. The word processor has a standard menu, toolbar, and dialog-based user interface:[3]

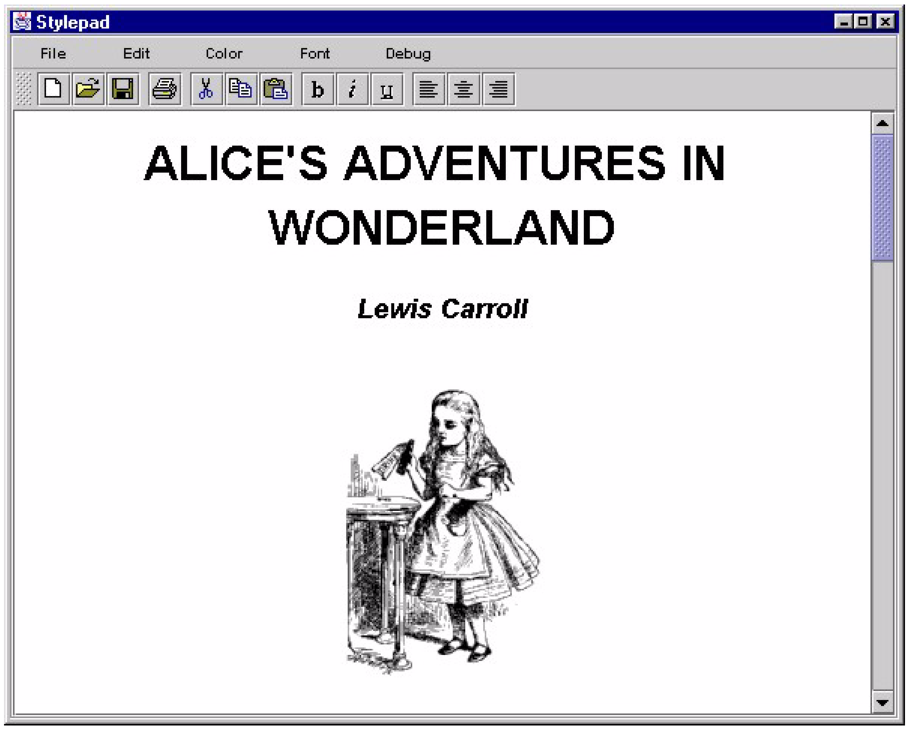


Figure 9. A simple word processing application

Before discussing application-specific *user-defined agents*, we begin by discussing *default agents* that provide generic, cross-application data collection support.

4.1.1 Default Agents

At the most basic level, analyzing application use presupposes the ability to identify when users are pro-actively using the user interface and providing values to the application.

3. "Stylepad" is a demonstration application shipped by Sun Microsystems as part of their Java Development Kit (JDK^{TM}). We added two lines of code to the application to support the following examples: one to start data collection and one to stop data collection and report results.

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 10. "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 11. "Use Non-Text" agent (abstract interaction events)

These agents apply *selection* to pick out events of interest indicating "pro-active user interface use" and apply *abstraction* to relate these events to this higher-level concept.

Other default agents can be defined to indicate when users have provided new values to the application via the user interface. The following agent stores an initial "Focus Value" whenever a "`FOCUS_GAINED`" event is observed in any user interface component:

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

Figure 12. "Value Initial" agent (abstract interaction events)

The following agent generates a "`VALUE_PROVIDED`" abstract interaction event whenever a "`FOCUS_LOST`" event is observed 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 13. "Value Provided" agent (abstract interaction events)

These agents combine *selection* and *access to context* to pick out events of interest indicating "provision of values" and apply *abstraction* to relate these events to this higher-level concept.

The purpose of such default agents is to allow other data collection code to refer to these higher-level abstract events without referring 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 semantics.

4.1.2 Menu and Toolbar Agents

User-defined agents can be defined to relate data to features of the user interface such as menus and toolbars. 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 14. “File Menu” agent (relating data to UI features)

This agent is triggered whenever any of the items in the “File Menu” is used and generates an abstract “MENU” event (with the agent itself as the event source) to indicate that the “File Menu” has been used. It uses the “Events” *reduction* algorithm to capture data regarding the number of times each “File Menu” item has been used:[4].

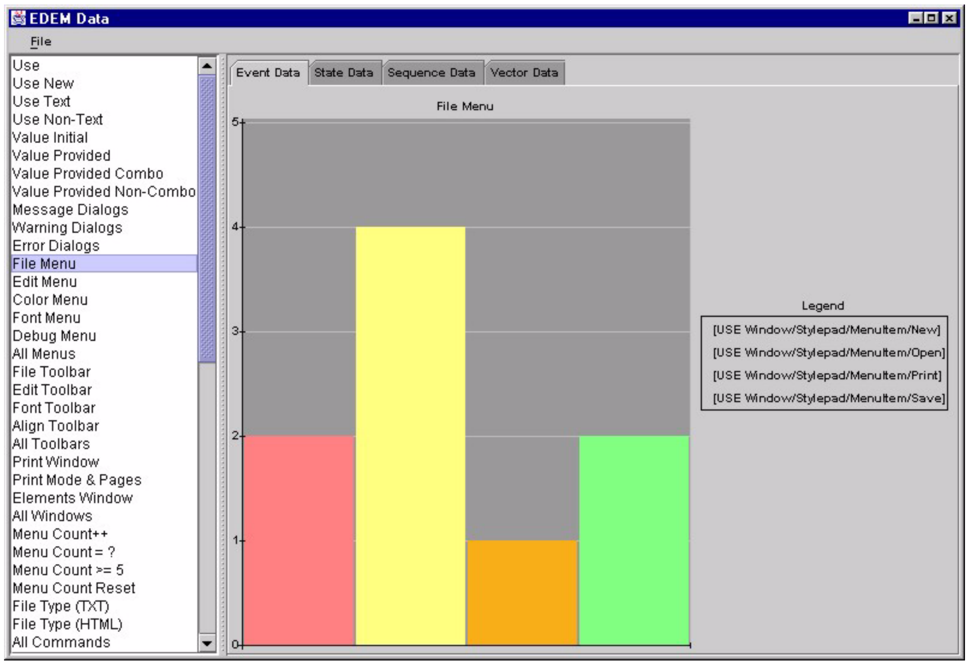


Figure 15. “File Menu” data (relating data to UI features)

Similar agents can be defined to perform analogous actions for all menus and toolbars and the abstract events generated by such agents can be used by other agents to characterize the use of whole menus and toolbars as opposed to their constituent parts, as illustrated below.

Once similar agents have been defined for each of the application’s menus, an agent of the following form can be defined to summarize the use of all menus:

Begin	Trigger	MENU|*|*
	Data	RecordEventData()

Figure 16. “All Menus” agent (relating data to UI features)

This agent is triggered whenever any “`MENU`” event is observed coming from any agent, and records data regarding the triggering event. Figure 17 illustrates the data captured by this agent, namely, the number of times each menu has been used:

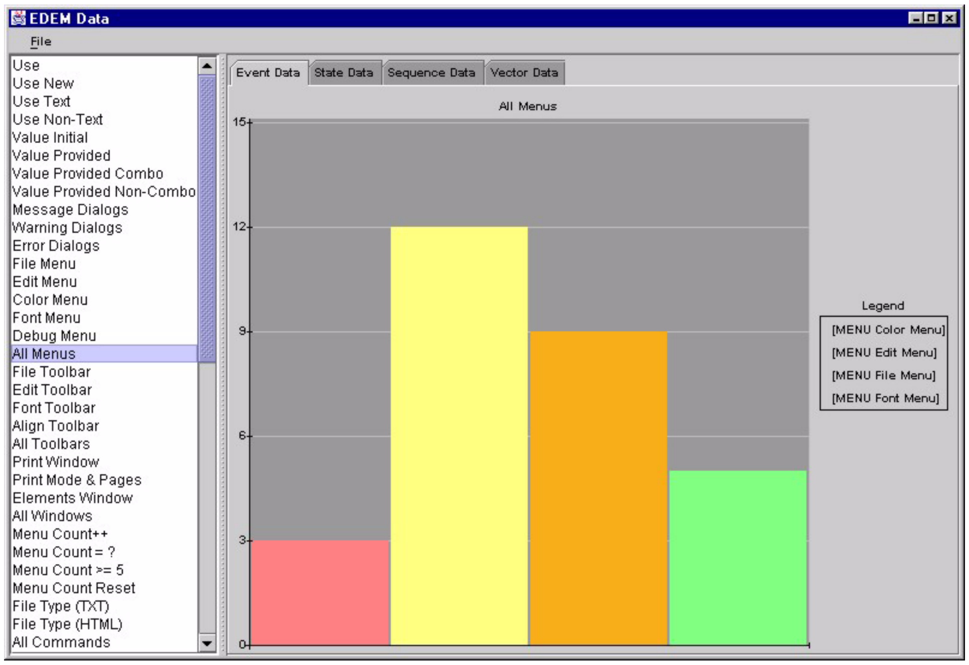


Figure 17. “All Menus” data (relating data to UI features)

4.Three simple *event data* reduction algorithms were implemented as part of this research. The “Events” algorithm stores event data in a hash table format in which keys indicate unique observed events and values indicate the number of times each event was observed. The “Transitions” algorithm stores event data in a hash table format in which keys indicate unique observed event transitions (between triggering events) and values indicate the number of times each transition was observed. The “Sequences” algorithm stores event data in a hash table format in which keys indicate unique observed event sequences (occurring between Begin and End Triggers) and values indicate the number of times each sequence was observed.

4.1.3 Command Agents

User-defined agents can also be defined to relate the use of application commands to the ways in which commands are invoked via the user interface. The following agent generates "CMD" events relating invocations (and cancellations) of the "File->Print" command to the various ways in which the command can be controlled via the user interface:

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

Figure 18. "File->Print" agent (relating data to application features)

Similar agents can be defined for all commands and the abstract events generated by such agents can be used by other agents to characterize general command use, as opposed to specific command invocation methods, as illustrated below.

In this case, we demonstrate how event data regarding command use can be related to state information regarding the type of document being edited at the time of command invocations. The following agents update a global state variable indicating the current "File Type" being edited in the word processor 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 19. "File Type" agents (incorporating state)

The following agent then invokes the "Append to Event Data" *reduction* algorithm to append the value of the "File Type" global state variable to each observed "CMD" event:

Begin	Trigger	CMD|*|*

	Data	RecordEventData()
		RecordStateDataPerEvent("File Type")

Figure 20. "All Commands" agent (incorporating state)

Figure 21 illustrates the resulting data:

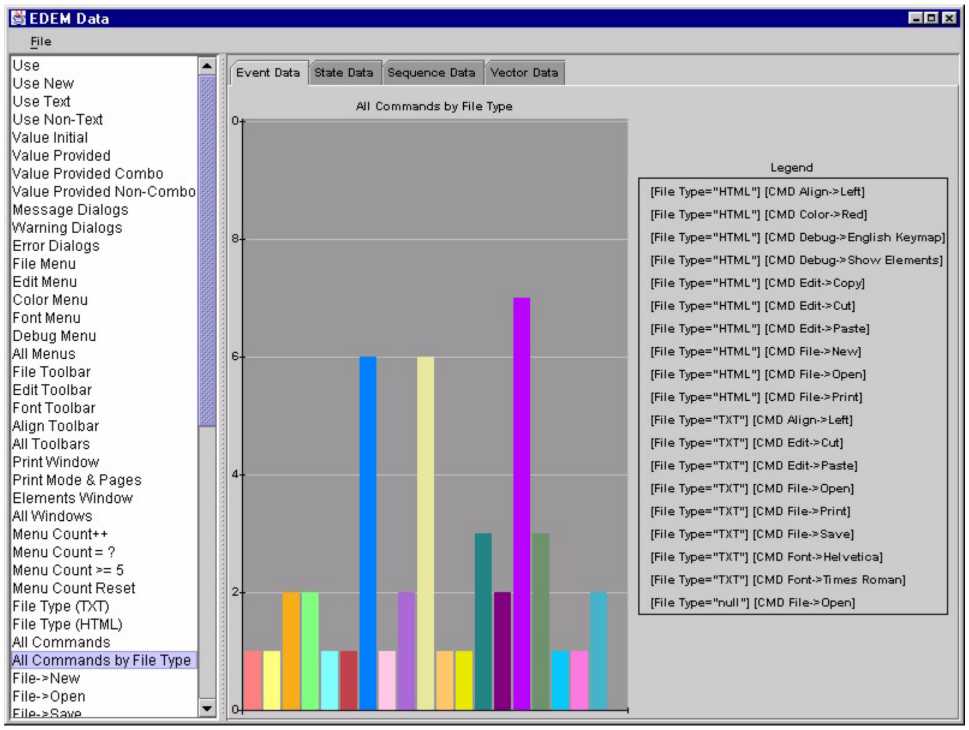

EDEM Data
File
Use
Use New
Use Text
Use Non-Text
Value Initial
Value Provided
Value Provided Combo
Value Provided Non-Combo
Message Dialogs
Warning Dialogs
Error Dialogs
File Menu
Edit Menu
Color Menu
Font Menu
Debug Menu
All Menus
File Toolbar
Edit Toolbar
Font Toolbar
Align Toolbar
All Toolbars
Print Window
Print Mode & Pages
Elements Window
All Windows
Menu Count++
Menu Count = ?
Menu Count >= 5
Menu Count Reset
File Type (TXT)
File Type (HTML)
All Commands
All Commands by File Type
File->New
File->Open
File->Save
Event Data
State Data
Sequence Data
Vector Data
All Commands by File Type
0
8
6
4
2
0
Legend
[File Type="HTML"] [CMD Align->Left]
[File Type="HTML"] [CMD Color->Red]
[File Type="HTML"] [CMD Debug->English Keymap]
[File Type="HTML"] [CMD Debug->Show Elements]
[File Type="HTML"] [CMD Edit->Copy]
[File Type="HTML"] [CMD Edit->Cut]
[File Type="HTML"] [CMD Edit->Paste]
[File Type="HTML"] [CMD File->New]
[File Type="HTML"] [CMD File->Open]
[File Type="HTML"] [CMD File->Print]
[File Type="TXT"] [CMD Align->Left]
[File Type="TXT"] [CMD Edit->Cut]
[File Type="TXT"] [CMD Edit->Paste]
[File Type="TXT"] [CMD File->Open]
[File Type="TXT"] [CMD File->Print]
[File Type="TXT"] [CMD File->Save]
[File Type="TXT"] [CMD Font->Helvetica]
[File Type="TXT"] [CMD Font->Times Roman]
[File Type="null"] [CMD File->Open]

Figure 21. "All Commands by File Type" data (incorporating state)

This allows event data to be analyzed across multiple "modes of use" (a topic discussed further below).

4.1.4 Dialog Agents

User-defined agents can also be defined to capture data regarding the use of application dialogs. The following agent captures selected event and state information regarding "Print Dialog" use:

Begin	Trigger	WINDOW_ACTIVATED|Window/Print|*
	Action	PostEvent("OPEN", AgentSource)

	Data	RecordEventData(VALUE_PROVIDED|*|*)
		RecordEventSequenceData(VALUE_PROVIDED|*|*)

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 22. "Print Window" agent (dialog event and state data)

This agent uses the “Events” reduction algorithm to capture data regarding the number of times controls in the “Print Dialog” are used to provide new values. It uses the “Sequences” reduction algorithm to capture data regarding the order in which controls are used to provide new values. And it uses the “Values” reduction algorithm to capture data regarding the specific values provided using the controls.[5]

4.1.5 Impact Analysis

Figure 23 illustrates the impact of abstraction, selection, and reduction on the number of bytes of data generated (plotted on a log scale) over time. The first point in each series indicates the number of bytes generated by the proposed approach when applied to a simple word processing session in which a user opens a file, performs a number of menu and toolbar operations, edits text, and saves and closes the file. The subsequent four points in each series indicate the amount of data generated assuming the user performs the same basic actions four times over. Thus, this graph represents an approximation of data growth over time based on the assumption that longer sessions primarily consist of repetitions of the same high-level actions performed in shorter sessions.

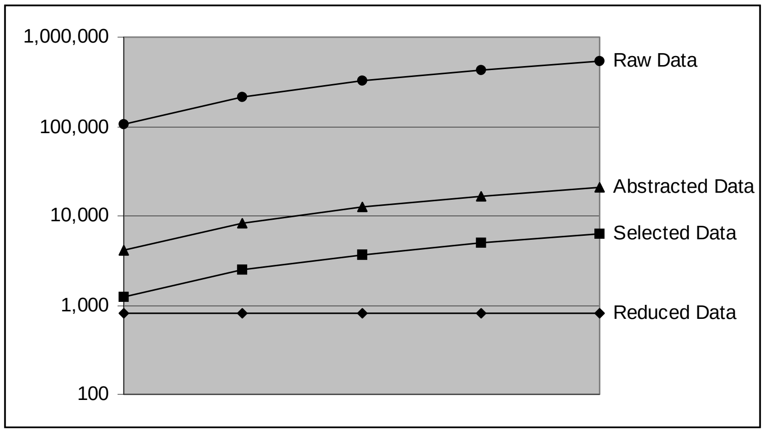


Figure 23. Impact of abstraction, selection, and reduction on bytes of data generated (plotted on a log scale) over time

The “raw data” series indicates the number of bytes of data generated if all window system events are captured. The “abstracted data” series indicates the amount of data generated if only abstract “`USE`” and “`VALUE_PROVIDED`” events are captured (about 4% of the size of raw data). The “selected data” series indicates the amount of data generated if only selected abstract events and state values regarding menu, toolbar, and dialog use are

5.Three simple *state data* reduction algorithms were implemented as part of this research. The “Values” algorithm stores state data in a hash table format in which keys indicate unique observed values and hash table values indicate the number of times each value was observed. The “Vectors” algorithm stores state data in a hash table format in which keys indicate unique observed vectors of values and hash table values indicate the number of times each vector of values was observed. This allows co-occurrence of values to be analyzed. The “Append to Event Data” algorithm allows state data to be appended to events that are then reduced using the “Events” reduction algorithm. This allows event data to be analyzed based on state information captured at the time of event occurrences.

captured (about 1% of the size of raw data). Finally, the “reduced data” series indicates the amount of data generated if abstracted and selected data is reduced prior to reporting (less than 1% of the size of raw data with little to no growth over time depending on the number of new, unique events or state values observed as time progresses).[6]

4.2 The Evolution Problem

Figure 24 depicts a reference architecture that illustrates the components typically required to perform large-scale usage data collection:

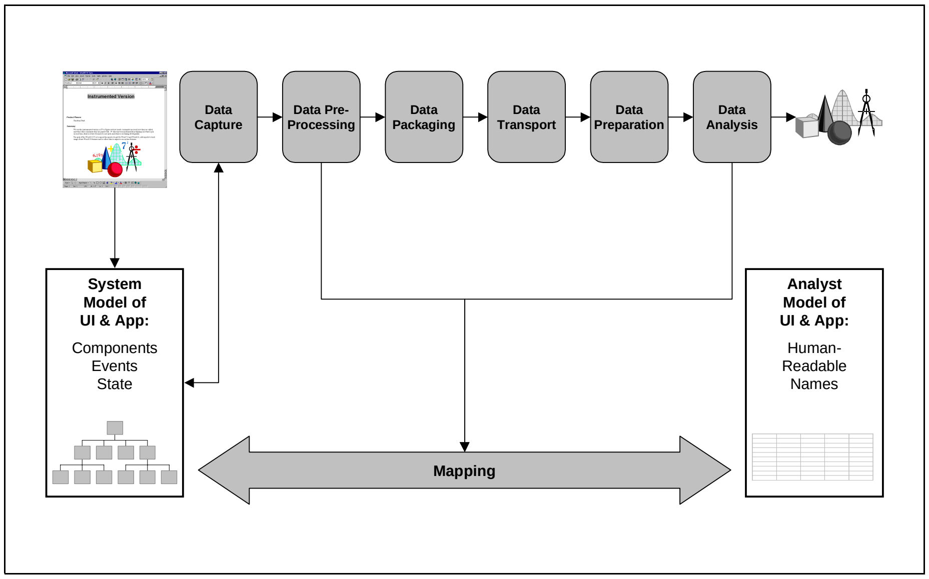


Figure 24. Data collection reference architecture

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).

6.Note that all of these byte counts can be further reduced using standard text or data compression techniques.

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).

Figure 25 illustrates how the reference architecture is instantiated when instrumentation code is inserted directly into application code:

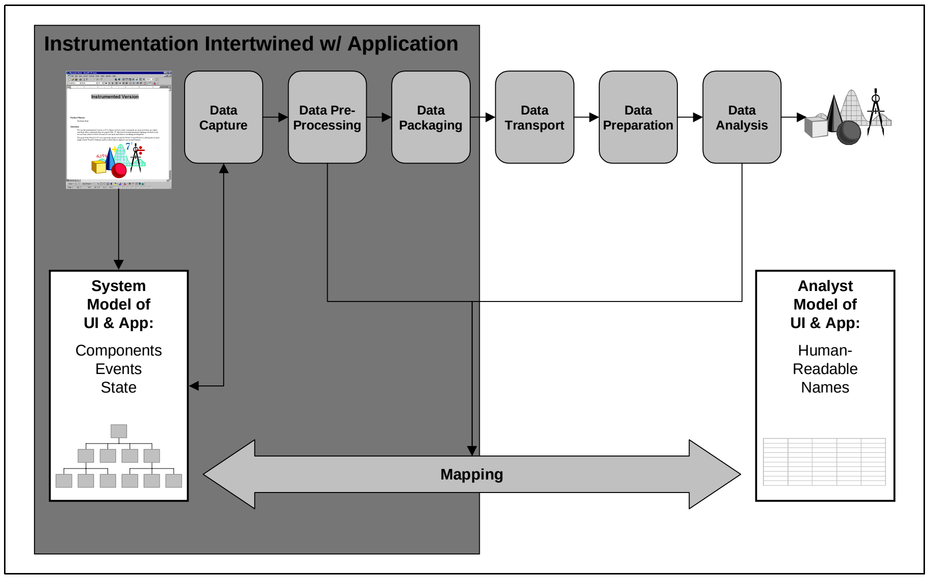


Figure 25. Instrumentation-based data collection architecture

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.

Figure 26 illustrates an improved instantiation of the reference architecture based on an event monitoring approach:

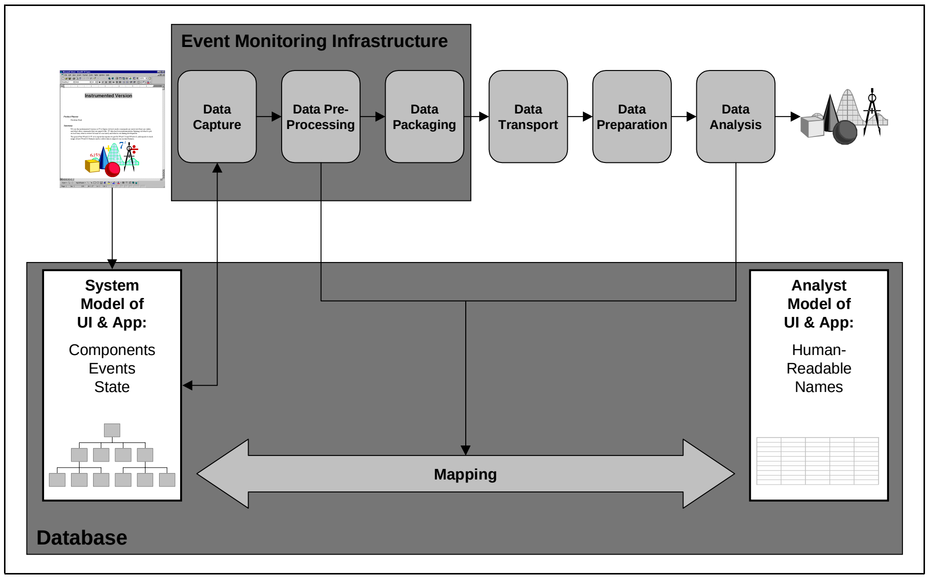


Figure 26. Event monitoring-based data collection architecture

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, Figure 27 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:

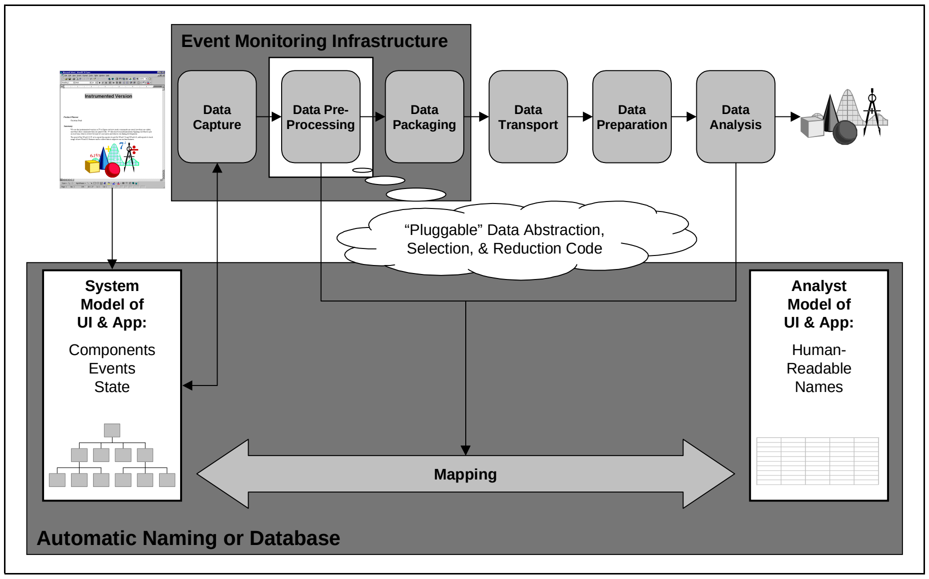

Event Monitoring Infrastructure
Data Capture
Data Pre-Processing
Data Packaging
Data Transport
Data Preparation
Data Analysis
“Pluggable” Data Abstraction, Selection, & Reduction Code
System Model of UI & App:
Components
Events
State
Mapping
Analyst Model of UI & App:
Human-Readable Names
Automatic Naming or Database

Figure 27. 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 article 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.[7] 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.

7.An uncompressed, ASCII agent definition file containing 64 agents for the Stylepad example is less than 70K bytes in size.

5. METHODOLOGICAL CONSIDERATIONS

The previous sections have focused primarily on technical issues. This section discusses some methodological issues involved in collecting, analyzing, and interpreting usage information and incorporating results into development.

5.1 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 section 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 section presented a number of example 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 as well as 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.[8] As mentioned above, data of this type should be linked 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.

8.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 research is needed in this area.

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 updating such "settings" may occur relatively infrequently.

5.2 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, generic usability problem indicators, or 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 constraints on users, for example:

- Number and % of actions that are of type X for users of type Y
- Number and % of time spent doing X for users of type Y
- Number and % of actions that are of type X for users who do X
- Number and % of time spent doing X for users who do X

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, 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 point 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.

5.3 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.

The purpose of this section is simply to point out that usage data must often be augmented by other sources of information in order to appropriately interpret and act on results.

6. EVALUATION

This research has been subjected to a number of evaluative activities, both inside and outside of the research lab. Section 4 presents an analytical assessment of the proposed approach in terms of its demonstrated support for the problems raised at the end of Section 2. This section presents a number of empirical assessments of the principles and techniques underlying this 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. While these empirical assessments have not been formal, we believe they have been practical and provide meaningful results given our hypotheses and domain of interest. In summary, they provide informal evidence that suggests:

- Observing users to identify mismatches between expected and actual use can indeed lead to design improvements (NYNEX).
- There are a number of areas in which automated support might be used to support the mismatch identification and resolution process (NYNEX).
- Independent third-party developers can successfully apply the approach, and the effort and expertise required to author agents is not prohibitive (Lockheed Martin).
- The proposed approach can capture impact assessment and effort allocation-related information in addition to design-related information (Lockheed Martin).
- Software monitoring techniques can indeed be exploited to capture strategic information useful in supporting design, impact assessment, and effort allocation decisions in practice, not just in theory (Microsoft).

- Current practice, as exemplified by the Microsoft approach, can be substantially improved based on the concepts developed in this research (Microsoft).

We describe these experiences and results in greater detail in the following subsections.

6.1 NYNEX Corporation:[9] The Bridget Project

6.1.1 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 provided informal evidence to support the hypothesis that observing use to compare it against expectations can 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].

6.1.2 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. Once a prototype of Bridget was 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.

6.1.3 Results

There were two major results of this experience. First, the design 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.

9.NYNEX Corporation is now known as Bell Atlantic Corporation as a result of a merger.

A second result was that a number of areas in which automated support might be used to improve this development process were identified. First, it was observed that much time was spent traveling between developer 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 jointly by NYNEX Corporation and 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.

6.2 Lockheed Martin Corporation: The GTN Scenario

6.2.1 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 the goal of 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].

6.2.2 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.

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 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 Section 3 for more details.

6.2.3 Results

There were two major results of this experience. First, the experience suggested that independent developers could successfully apply the approach, that the effort and expertise required to author agents was not extensive, and 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 (as opposed to impact assessment and effort allocation-related information).[10]

6.3 Microsoft Corporation: The “Instrumented Version”

6.3.1 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 indeed 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.

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

6.3.2 Description

Due to a non-disclosure agreement, we cannot name the specific product that was the subject of the usage study nor discuss how the product was improved based on usage data. However, we can describe the product and the data collection process employed by Microsoft. The application in question features over 1000 application "commands" accessible through menus and toolbars and over 300 user interface dialogs. The first author of this article 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.

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 concepts developed in this research.

6.3.3 Results

The results of this experience were instructive in a number of ways. First and foremost, it further supported the hypothesis that software monitoring techniques can 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 code and performing 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.

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.

The Microsoft experience has helped validate our emphasis on the selection, abstraction, reduction, context, and evolution problems by illustrating the practical results of failing to adequately address these problems. 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. As a result, development and maintenance of instrumentation continues to be problematic resulting in studies only being conducted at irregular intervals. Furthermore, there is no mechanism for flexibly mapping between lower level events and higher level events of interest (i.e., abstraction). As a result, abstraction must be performed as part of the post-hoc analysis process resulting in failures to notice errors in data collection that affect abstraction until *after* data has actually been collected. 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, data is not reduced in context, resulting in every user or application action being recorded as a separate data record. As a result, a majority of the collected data was never used in analysis, particularly sequential aspects. 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.[11]

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. The ease with which these insights have been incorporated into the proposed approach (and associated methodological considerations) further increases our confidence in the flexibility of the approach.

Most importantly, the experience 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 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.

11.The author briefed the product planning and program management teams regarding some of the concepts developed in this research. They expressed enthusiasm about adding such advanced features incrementally after the initial event monitoring infrastructure had been implemented and evaluated.

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 in the print dialog 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 1,000,000 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 take precedence 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. Fortunately, these strengths and weaknesses nicely complement the strengths and weaknesses inherent in current usability testing practice, in which subtle usability problems may be identified through careful human observation, but in which there is little sense of the "big picture" of how applications are used on a large scale. 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 for developers wishing to assess the impact of suspected problems or to focus development effort for the next version. However, it is not information that can be reliably 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 other issues in data collection. 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 (e.g., by calling an API as described in Section 3), 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.[12]

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

12.Note that while this insight occurred as a result of observing the Microsoft approach, the Microsoft approach, unlike the proposed approach, does not exploit this insight to reduce sequential data.

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 spent 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 can be more economically captured in periodic “snapshots” 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.[13]

7. CHALLENGES

This section 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.

7.1 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 article. 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.[14]

13.Note, again, that while this insight occurred as a result of observing the Microsoft approach, the Microsoft approach, unlike the proposed approach, does not support flexible associations between event and state data (without significant impact on data preprocessing, packaging, preparation, and analysis code).

14.A similar problem arises in maintaining test code and test cases.

The solution to the evolution problem presented in this article 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 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.

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 Section 4 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 via the user interface. Such relations must be updated when new 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.

7.2 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 other stakeholders in the development process. 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 in the following section.

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. Alternative authoring approaches must also be investigated.

7.3 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 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 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. However, in beta test situations, consent to allow data to be reported 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.

7.4 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, as in the current prototype. Assuming low level events are being abstracted into higher level events associated with application features, a feedback dialog might also 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 user had intended to open a different dialog, versus dismissals due to the fact that the dialog simply does not provide the functionality 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 (as illustrated in the usage scenario). 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.

7.5 Integration With Requirements and Design

There is a gap between developers’ tacit or informal usage expectations and the formal expression of usage expectations in data collection agents. Currently, designers must bridge this gap on their own. We are investigating ways to narrow the gap by making use of existing requirements and design artifacts (such as usability requirements, cognitive walkthroughs, and use cases) and are also exploring new modeling techniques. Ideally, usage expectations might be expressed as part of the requirements and design specification process, and the resulting documents used to informally derive, or automatically generate, data collection code. An early language and approach to automatic monitoring of requirements (not focused on interactive systems) was proposed by Fickas and Feather [Fickas and Feather 1995, Feather et al. 1997]. This is an area we are actively investigating, particularly in relation to the design of interactive systems based on the Unified Modeling Language (UML) [Booch et al. 1998] and Usage-Centered Design (UCD) [Constantine and Lockwood 1999]. We are exploring this area by integrating our work on usage data collection with our work on UML-based design [Robbins and Redmiles 1999].

7.6 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 successfully 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. However, the use of sophisticated post-hoc analysis techniques does not obviate the need for performing selection, abstraction, reduction, and context-capture in data collection. We view the proposed approach as complementary to, if not a prerequisite to, the successful application of most post-hoc analysis techniques.

8. RELATED WORK AND FUTURE DIRECTIONS

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.

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 1993] 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. GOMS [John and Kieras 1996a & 1996b], 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 1995] 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 1994] 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^{TM} [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].

9. CONCLUSIONS

This article has presented technical and methodological strategies to enable usage- and usability-related information of higher quality than currently available from beta tests to be collected on a 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. While such data collection techniques are stronger in terms of capturing indicators of the "big picture" than in identifying subtle, nuanced, and unexpected usability issues, these strengths and weaknesses nicely complement the strengths and weaknesses inherent in current usability testing practice, in which subtle usability problems may be identified through careful human observation, but in which there is little sense of the "big picture" of how applications are used on a large scale.

The principles and techniques underlying this work have been subjected to a number of evaluative activities including: (1) the development of three research prototypes at NYNEX Corporation, 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, 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 can indeed be used to capture usage- and usability-related information useful in supporting design, impact assessment, and effort allocation decisions in the development of interactive systems. The third and final prototype was constructed based on the experience and insights gained from the first two prototypes, an in-depth survey of related work, and the Microsoft experience, and 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 meaningful usage- and usability-related information might be captured on a potentially large and ongoing basis.

In summary, the main contributions of this work include: (1) a theory motivating the significance of usage expectations in development and the importance of collecting usage information (Section 3), (2) 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 (Section 3), (3) solutions to the abstraction, selection, reduction, context, and evolution problems demonstrated within a single data collection architecture (Section 4), (4) a reference architecture to provide design guidance regarding key components and relationships required to support large-scale data collection

(Section 4), and (5) methodological guidance regarding data collection, analysis, and interpretation (Section 5).

While this research has focused on capturing data regarding the use of window-based interactive applications, the principles, techniques, and methodology underlying the approach might 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 be practically collected over extended periods of time, 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, as exemplified by Microsoft's Office Assistant [Bott and Leonhard 1999], 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.

ACKNOWLEDGEMENTS

The authors would like to thank the members of the Lockheed Martin C2 Integration Systems Team, Teri Payton, Lyn Uzzle, and Martin Hile, for their willingness to collaborate. Thanks also to members of Microsoft Research, Product Planning, Program Management, and Usability, including David Caulton, Debbie Dubrow, Jonathan Grudin, Paul Kim, Reed Koch, Ashok Kuppusamy, Dixon Miller, Chris Pratley, Roberto Taboada, Jose Luis Montero Real, Erik Rucker, Andrew Silverman, Kent Sullivan, and Gayna Williams. Thanks also to those who participated in laying the groundwork for this research including Andreas Girgensohn, Allison Lee, Frank Shipman, and Thea Turner.

This work is financially supported by the National Science Foundation, grant number CCR-9624846, and by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory or the U.S. Government.

REFERENCES

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.

AQUEDUCT SOFTWARE. AppScope Web pages. URL: http://www.aqueduct.com/. 1998.

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.

BADRE, A.N. AND SANTOS, P.J. CHIME: A knowledge-based computer-human interaction monitoring engine. Tech Report GIT-GVU-91-06. 1991a.

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.

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.

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.

BATES, P.C. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, Vol. 13, No. 1, 1995.

BOOCH, G., JACOBSON, I., AND RUMBAUGH, J. *The Unified Modeling Language User Guide*. Addison-Wesley. 1998.

BOTT, E. AND LEONHARD, W. *Special Edition Using Microsoft Office 2000.* Que Corp. 1999.

CASTILLO, J.C. AND HARTSON, H.R. Remote usability evaluation Web pages. URL: http://hci.ise.vt.edu/~josec/remote_eval/. 1997.

CHEN, J. Providing intrinsic support for user interface monitoring. In *Proceedings of INTERACT'90.* 1990.

CONSTANTINE, L.L. AND LOCKWOOD, L.A.D. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design.* ACM Press. 1999.

COOK, J.E. AND WOLF, A.L. Toward metrics for process validation. In *Proceedings of ICSP'94.* 1994.

COOK, J.E., AND WOLF, A.L. Automating process discovery through event-data analysis. In *Proceedings of ICSE'95.* 1995.

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.

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.

CYPHER, A. (ED.). *Watch what I do: programming by demonstration.* MIT Press, Cambridge MA, 1993.

CYPHER, A. Eager: programming repetitive tasks by example. In *Proceedings of CHI'91.* 1991.

ERGOLIGHT USABILITY SOFTWARE. Operation Recording Suite (EORS) and Usability Validation Suite (EUVS) Web pages. URL: http://www.ergolight.co.il/. 1998.

FEATHER, M.S., NARAYANASWAMY, K., COHEN, D., AND FICKAS, S. Automatic monitoring of software requirements. Research Demonstration in *Proceedings of ICSE'97.* 1997.

FICKAS, S. AND FEATHER, M.S. Requirements monitoring in dynamic environments. *IEEE International Symposium on Requirements Engineering.* 1995.

FULL CIRCLE SOFTWARE. Talkback Web pages. URL: http://www.fullcirclesoftware.com/. 1998.

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 KBSE'94.* 1994.

GOULD, J. D. AND LEWIS, C. Designing for usability - Key principles and what designers think. In *Proceedings of CHI'83.* 1983.

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.

HEWLETT PACKARD. Application response measurement API Web pages. URL: http://www.hp.com/openview/rpm/arm/. 1998.

HILBERT, D.M. AND REDMILES, D.F. Extracting usability information from user interface events. *ACM Computing surveys* (under review). 1999.

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.

HILBERT, D.M. AND REDMILES, D.F. Agents for collecting application usage data over the Internet. In *Proceedings of Autonomous Agents'98.* 1998b.

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.

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.

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, 1996a.

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, 1996b.

KAY, J. AND THOMAS, R.C. Studying long-term system use. *Communications of the ACM,* Vol. 38, No. 7, 1995.

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.

LECEROF, A. AND PATERNO, F. Automatic support for usability evaluation. *IEEE Transactions on Software Engineering,* Vol. 24, No. 10, 1998.

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.

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.

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.

MERCURY INTERACTIVE. WinRunner and XRunner Web pages. URL: http://www.merc-int.com/. 1998.

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.

NIELSEN, J. *Usability engineering.* Academic Press/AP Professional, Cambridge, MA, 1993.

NIELSEN, J. A virtual protocol model for computer-human interaction. *International Journal of Man-Machine Studies,* 24, 1986.

PAYNE, S.G. AND GREEN, T.R.G. Task-action grammars: A model of the mental representation of task languages. *Human-Computer Interaction,* Vol. 2, 1986.

ROBBINS, J.E. AND REDMILES D.F. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. Construction of Software Engineering Tools (CoSET'99). 1999.

ROSENBLUM, D.S. Specifying concurrent systems with TSL. *IEEE Software,* Vol. 8, No. 3, 1991.

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.

SUN MICROSYSTEMS. JavaStar Web pages. URL: http://www.sun.com/suntest/. 1998.

SWALLOW, J., HAMELUCK, D., AND CAREY, T. User interface instrumentation for usability analysis: A case study. In *Proceedings of Cascon'97.* 1997.

SUMMERS, R. *Official Microsoft Netmeeting 2.1 Book.* Microsoft Press. 1998.

TAYLOR, M.M. Layered protocols for computer-human dialogue I: Principles. *International Journal of Man-Machine Studies,* 28, 1988a.

TAYLOR, M.M. Layered protocols for computer-human dialogue II: Some practical issues. *International Journal of Man-Machine Studies,* 28, 1988b.

TIBCO. HAWK Enterprise Monitor Web pages. URL: http://www.tibco.com/products/products.html. 1998.

USER MODELING INC. User Modeling Inc. Web pages. URL: http://um.org/. 1998.

WEILER, P. Software for the usability lab: a sampling of current tools. In *Proceedings of INTERCHI'93.* 1993.