Large-Scale Collection of Usage Data to Inform Design

David M. Hilbert[1] & David F. Redmiles[2]

[1]FX Palo Alto Laboratory, 3400 Hillview Ave., Bldg. 4, Palo Alto, CA 94304 USA

[2]Information and Computer Science, University of California, Irvine, CA 92717 USA

[1]hilbert@pal.xerox.com & [2]redmiles@ics.uci.edu

Abstract: 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 on users at large, and (2) deciding where to focus development and evaluation resources to maximize the benefit for users at large. This paper describes an agent-based approach for collecting usage data and user feedback over the Internet that addresses these limitations to provide developers with a *complementary* source of usage- and usability-related information. Contributions include: a theory to motivate and guide data collection, an architecture capable of supporting very large scale data collection, and real-word experience suggesting the proposed approach is complementary to existing practice.

Keywords: Usability testing, beta testing, automated data collection, software monitoring, post-deployment evaluation

1 Introduction

Involving end users in the development of interactive systems increases the likelihood those systems will be useful and usable. The Internet presents hitherto unprecedented, and currently underutilized, opportunities for increasing user involvement by: (1) enabling cheap, rapid, and large-scale distribution of software for evaluation purposes and (2) providing convenient mechanisms for communicating application usage date and user feedback to interested development organizations.

Unfortunately, a major challenge facing development organizations today is that there are already more suspected problems, proposed solutions, and novel design ideas emanating from various stakeholders than there are resources available to address these issues (Cusumano & Selby, 1995). As a result, developers are often more concerned with addressing the following problems, than in generating more ideas about how to improve designs:

- *Impact assessment:* To what extent do suspected or observed problems actually impact 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?

Current usability and beta testing practices do not adequately address these questions. This paper describes an approach to usage data and user feedback collection that *complements* existing practice by helping developers address these questions more directly.

2 Problems

2.1 Usability Testing

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

- Size, because data collection and analysis limitations result in evaluation effort being linked to the number of evaluation subjects.
- Scope, because typically only a small fraction of an application's functionality can be exercised in any given evaluation.
- Location, because users are typically displaced from their normal work environments to more controlled laboratory conditions.
- Duration, because users cannot devote extended periods of time to evaluation activities that take them away from their day-to-day responsibilities.

Perhaps more significantly, however, once problems have been identified in the lab, the *impact assessment* and *effort allocation* problems remain: What is the actual impact of identified problems on users at large? How should development resources be allocated to fix those problems? Furthermore, because usability testing is itself expensive in terms of user and evaluator effort: How should scarce usability resources be focused to produce the greatest benefit for users at large?

2.2 Beta Testing

Data quality is the critical limiting factor in beta tests. When beta testers report usability issues in addition to bugs, data quality is limited in a number of ways.

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 tests often volunteer comments in which they report problems with the interface although the problems did not affect their ability to complete tasks. When asked for a justification, these users say things like: "Well, it was easy for me, but I think other people would have been confused." On the other hand, users who have difficulties using a particular interface often do not volunteer comments, and if pressed, report that the interface is well designed and easy to use. When confronted with the discrepancy, these users say things like: "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 unfounded and potentially misleading feedback from beta testers not having 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 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. Smilowitz and colleagues offered the following as one possible explanation:

> 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 (Smilowitz et al., 1994).

Thus, users are limited in their ability to identify and report problems due to a *lack of knowledge* regarding expected use.

Another 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. Without this information (or information regarding the frequency with which features associated with reported problems are used) it is difficult to assess the impact of reported problems, and therefore, to decide how to allocate resources to fix those problems.

3 Related Work

3.1 Early Attempts to Exploit the Internet

Some researchers have investigated the use of Internet-based video conferencing and remote application sharing technologies, such as Microsoft NetMeeting™, to support remote usability evaluations (Castillo & Hartson, 1997). Unfortunately, while leveraging the Internet to overcome geographical barriers, these techniques do not 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 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 that sends video data surrounding user-identified incidents back to experimenters. While addressing, to a limited degree, the lack of detail in beta tester-reported data, this approach still suffers from 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.

[1] These examples were taken from a discussion group for usability researchers and professionals involved in usability evaluations.

3.2 Automated Techniques

An alternative approach 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 to identify potential problems. A number of instrumentation and monitoring techniques have been proposed for this purpose (Figure 1). However, as argued in more detail in (Hilbert & Redmiles, 2000), existing approaches all suffer from some combination of the following problems, limiting 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. This implies the need for "data abstraction" mechanisms to relate low-level data to higher-level concepts such as user interface and application features as well as users' tasks and goals.

The *selection problem:* The data required to answer usage questions is typically a small subset of the much larger set of data that might be captured. Failure to properly select data 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 separate necessary from unnecessary data prior to reporting and analysis.

The *reduction problem:* Much of the analysis needed to answer usage questions can actually be performed *during* data collection. Performing reduction during capture not only decreases the amount of data that must be reported, but also increases the likelihood that all the data necessary for analysis is actually captured. This implies the need for "data reduction" mechanisms to reduce data prior to reporting and analysis.

The *context problem:* Potentially critical information necessary in interpreting the significance of events is often not available in event data alone. However, such information may be available "for the asking" from the user interface, application, artifacts, or user. This implies the need for "context-capture" mechanisms to allow state data to be used in abstraction, selection, and reduction.

The *evolution problem:* Finally, data collection needs evolve over time independently from 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 that can be modified over time without impacting application deployment or use.

Figure 1 summarizes the extent to which existing approaches address these problems.

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 their support for identified 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

4 Approach

We propose a novel approach to large-scale usage data and user feedback collection that addresses these problems. Next we present the theory behind this work.

4.1 Theory of Expectations

When developing systems, developers rely on a number of expectations 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, the specific tasks and work environments of users, the application domain, and past experience in developing and using applications themselves. Some expectations are explicitly represented, for example, those 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 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 "http://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 may then be directed at capturing information that is helpful in detecting mismatches between expected and actual use, and mismatches may be used as opportunities to adjust the design based on usage-related information, or to adjust usage based on design-related information.

4.2 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 this end, the following process is employed: (1) developers design applications and identify usage expectations; (2) developers create agents to monitor application use and capture usage data and user feedback; (3) agents are deployed over the Internet independently of the application to run on users' computers; (4) agents perform in-context data abstraction, selection, and reduction as needed to allow actual use to be compared against expected use; and (5) 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, we introduced the notion of a default model to mediate between monitored components and the event service. Furthermore, because numerous agents were observed to be useful across multiple applications, we introduced the notion of default agents to allow higher-level generic data collection services to be reused across applications. See Figure 2.

Note that the proposed approach is different from traditional event monitoring approaches in that data abstraction, selection, and reduction is performed during data collection (by software agents) as opposed to after data collection (by human investigators). This allows

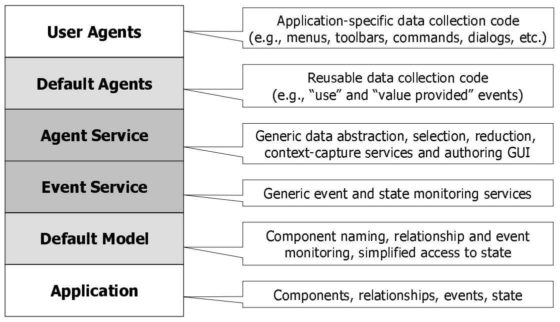

User Agents
Application-specific data collection code (e.g., menus, toolbars, commands, dialogs, etc.)
Default Agents
Reusable data collection code (e.g., "use" and "value provided" events)
Agent Service
Generic data abstraction, selection, reduction, context-capture services and authoring GUI
Event Service
Generic event and state monitoring services
Default Model
Component naming, relationship and event monitoring, simplified access to state
Application
Components, relationships, events, state

Figure 2: The layered relationship between the application, default model, event service, agent service, default agents, and user-defined agents. Shading indicates the degree to which each aspect is believed to be generic and reusable.

abstraction, selection, and reduction to be performed in-context, resulting in improved data quality and reduced data reporting and post-hoc analysis needs. For more technical details see (Hilbert, 1999).

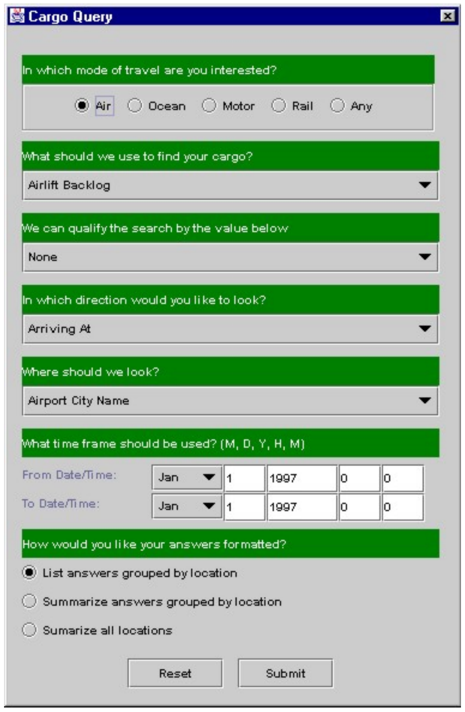
The current prototype works with Java applications and requires developers to insert two lines of code into their application: one to start data collection and one to stop data collection and report results. Once this has been done, developers use an agent authoring user interface to define agents without writing code (Figure 4). Once agents have been defined, they are serialized and stored in an ASCII file with a URL on a development computer. The URL is passed as a command-line argument to the application of interest. When the application is run, the URL is automatically downloaded and the latest agents instantiated on the user's computer. Agent reports are sent to development computers via E-mail upon application exit. For more technical details see (Hilbert, 1999).

4.3 Usage Scenario

To see how these services may be used in practice, consider the following scenario developed by Lockheed Martin C2 Integration Systems as part of a government-sponsored research demonstration.

A group of engineers is tasked with designing a web-based user interface to allow users to request information regarding Department of Defense cargo in transit. After involving users in design, constructing use cases, performing task analyses, doing cognitive walkthroughs, and employing other user-centered design methods, a prototype interface is ready for deployment (Figure 3).

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 available in subsequent sections. Expecting that this would not be a common problem, the engineers


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)

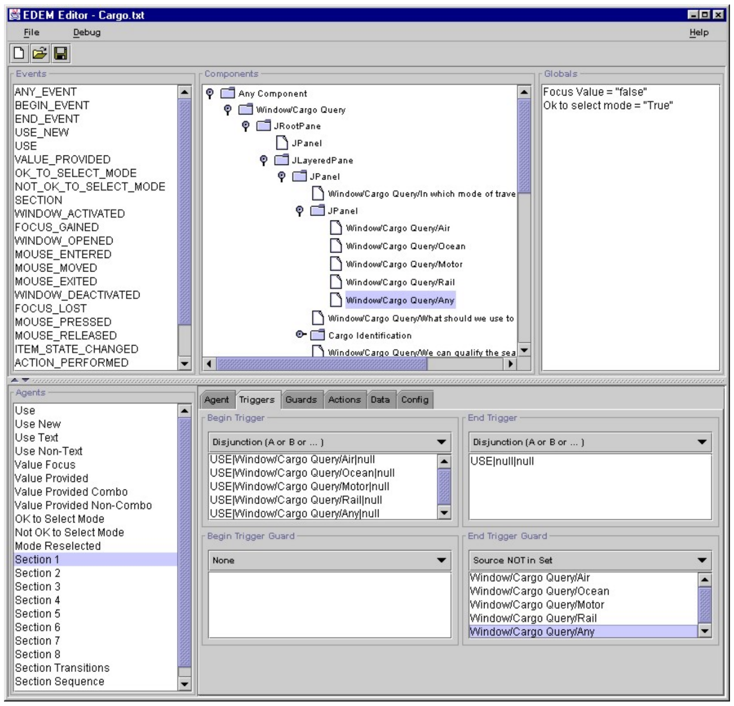
From Date/Time:	Jan	1	1997	0	0
To Date/Time:	Jan	1	1997	0	0

How would you like your answers formatted?

[✓ List answers grouped by location]
Summarize answers grouped by location
Sumarize all locations

Reset Submit

Figure 3: A prototype user interface for tracking Department of Defense cargo in transit.


EDEM Editor - Cargo.txt

File Debug Help

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

Agent | Triggers | Guards | Actions | Data | Config

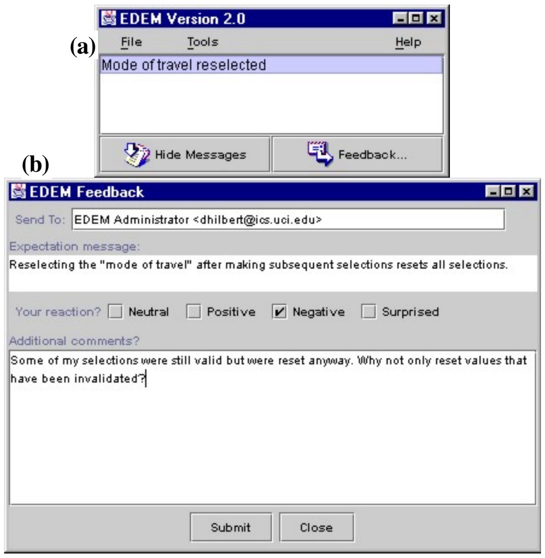
Begin Trigger: Disjunction (A or B or ...)
USE|Window/Cargo Query/Air|null
USE|Window/Cargo Query/Ocean|null
USE|Window/Cargo Query/Motor|null
USE|Window/Cargo Query/Rail|null
USE|Window/Cargo Query/Any|null

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 showing events (top left), components (top middle), global variables (top right), agents (bottom left) and agent properties (bottom right).



EDEM Version 2.0

File Tools Help

Mode of travel reselected

Hide Messages Feedback...

EDEM Feedback

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?

Submit Close

Figure 5: Agent notification (a) and user feedback (b). Use of these data collection features is optional.

decided to reset all selections to their default values whenever the "mode of travel" selection is reselected.

In Figure 4, the developer has defined an agent that "fires" whenever the user selects one of the controls in the "mode of travel" section of the interface and then selects controls outside that section. This agent is then used in conjunction with other agents to detect when the user has changed the mode of travel after having made subsequent selections. In addition to capturing data unobtrusively, the engineers decided to configure an agent to notify users (by posting a message) when it detected behavior in violation of developers' expectations (Figure 5a). By selecting an agent message and pressing the "Feedback" button users could learn more about the violated expectation and respond with feedback if desired (Figure 5b). Feedback was reported along with general usage data via E-mail each time the application was exited. Agent-collected data was later reviewed by support engineers who provided a new release based on the data collected in the field.

It is tempting to think that this example has a clear design flaw that, if corrected, would obviate the need for data collection. Namely, one might argue, the application should detect which selections must be reselected and only require users to reselect those values. To illustrate how this objection misses the mark, the Lockheed personnel deliberately fashioned the scenario to include a user responding to the agent with exactly this suggestion (Figure 5b). 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 is one of the key contributions of this approach.

5 Discussion

5.1 Lab Experience

We (and Lockheed personnel) have authored data collection agents for a number of example applications including the database query interface described in the usage scenario (15 agents), an interface for provisioning phone service accounts (2 agents), and a word processing application (53 agents).[1]

Figure 6 illustrates the impact of abstraction, selection, and reduction on the number of bytes of data generated by the word processing application (plotted on a log scale) over time. The first point in each series indicates the number of bytes generated by the 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.

The "raw data" series indicates the number of bytes of data generated if all window system events are captured including all mouse movements and key presses. The "abstracted data" series indicates the amount of data generated if only abstract events corresponding to proactive manipulation of user interface components 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 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 to simple counts of unique observed events (and event sequences) and state values (and state value vectors) prior to reporting (less than 1% of the size of raw data). There is little to no growth of reduced data over time because data size increases only when *unique* events or state values are observed for the *first* time.

5.2 Non-Lab Experience

We have also attempted to evaluate and refine our ideas and prototypes by engaging in a number of evaluative activities outside of the research lab. While these activities have been informal, we believe they have been practical and provide meaningful insights given our hypotheses and domain of interest. Namely, these experiences have all

[1] An uncompressed ASCII agent definition file containing 11 default agents and 53 user-defined agents for the word processor example is less than 70K bytes. The entire data collection infrastructure is less than 500K bytes.

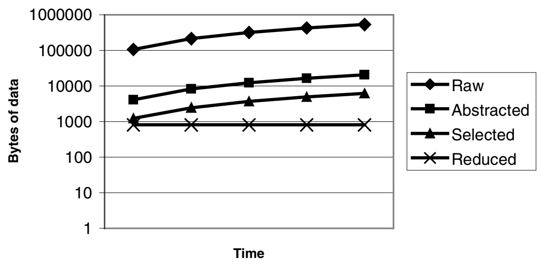


Figure 6: Impact of abstraction, selection, and reduction on bytes of data generated (plotted on a log scale) over time.

contributed evidence to support the hypothesis that automated software monitoring can indeed be used to capture data to inform design, impact assessment, and effort allocation decisions.

NYNEX Corporation: The Bridget System

The Bridget system is a form-based phone service provisioning system developed in cooperation between the Intelligent Interfaces Group at NYNEX Corporation and the Human-Computer Communication Group at the University of Colorado at Boulder (Girgensohn et al., 1994). The Bridget development process was participatory and iterative in nature. In each iteration, users were asked to perform tasks with a 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.

There were two major results of this experience. First, the design process outlined above led to successful design improvements that might not have been introduced otherwise. Second, we identified a number of areas in which automated support might improve the process.

Motivated by this experience, the second author of this paper helped develop a prototype agent-based system for observing usage on developers' behalf and initiating remote communication between developers and users when mismatches between expected and actual use were observed. This prototype served as the basis for the work described in this paper. However, the idea of capturing generic usage data in addition to detecting specific mismatches was not addressed. The significance of this oversight is highlighted below.

Lockheed Martin Corporation: The GTN Scenario

Based on the NYNEX experience, and with the goal of making the approach more scalable, the authors developed a second prototype (described in this paper) at the University of California at Irvine. Independent developers at Lockheed Martin Corporation then integrated this second prototype into a logistics information system as part of a government-sponsored demonstration scenario (described in the usage scenario).

There were two major results of this experience. First, the experience suggested that independent developers could successfully apply the approach with only moderate effort and that significant data could nonetheless be captured. Second, the data that was collected could be used to support impact assessment and effort allocation decisions in addition to design decisions (as illustrated in the usage scenario). This outcome had not been anticipated since, up to this point, our research had focused on capturing design-related information.

Microsoft Corporation: The "Instrumented Version"

Finally, in order to better understand the challenges faced by organizations attempting to capture usage data on a large scale in practice, the first author of this paper managed an instrumentation effort at Microsoft Corporation. The effort involved capturing basic usage data regarding the behavior of 500 to 1000 volunteer users of an instrumented version of a well-known Microsoft product over a two-month period.[1]

Because this was not the first time data would be collected regarding the use of this product, infrastructure already existed to capture data. The infrastructure consisted of instrumentation code inserted directly into application code that captured data of interest and wrote it to binary files. Users then copied these files to floppy disks and mailed them to Microsoft after a pre-specified period of use. Unfortunately, due to: (a) the sheer amount of instrumentation code already embedded in the application, (b) the limited time available for updating instrumentation to capture data regarding new features, and (c) the requirement to compare the latest data against prior data collection results, we were unable to reimplement the data collection infrastructure based upon this research. Thus, the existing infrastructure was used allowing us to observe, first-hand, the difficulties and limitations inherent in such an approach, and to compare it against our proposed approach.

The results of this experience were instructive in a number of ways. First and foremost, it further supported the hypothesis that automated software monitoring can indeed be used to inform design, impact assessment, and effort allocation decisions. Furthermore, it was an existence proof that there are in fact situations in which the benefits of data collection are perceived to outweigh the maintenance and analysis costs, even in an extremely competitive development organization in which time-to-market is of utmost importance. Lessons learned follow.

[1] Due to a non-disclosure agreement, we cannot name the product nor discuss how it was improved based on usage data. However, we *can* describe the data collection approach employed by Microsoft.

How Practice can be Informed by this Research

The Microsoft experience has further validated our emphasis on the abstraction, selection, reduction, context, and evolution problems by illustrating the negative results of failing to adequately address these problems in practice. 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. As a result, development and maintenance of instrumentation is costly resulting in studies only being conducted irregularly. 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 (such as missing context) until after data has been collected. Also, because data is not reduced prior to reporting, a large amount of data is reported, post-hoc analysis is unnecessarily complicated, and most data is never used in analysis (particularly sequential aspects). Finally, the approach does not allow users to provide feedback to augment automatically captured data.

How Practice has Informed this Research

Despite these observed limitations, this experience also resulted in a number of insights that have informed and refined this research. The ease with which we incorporated these insights into the proposed approach (and associated methodological considerations) further increases our confidence in the flexibility and generality 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, we focused on expectations of the first kind. However, this sort of analysis, by itself, is challenging due to difficulties in inferring user intent and in anticipating all the important areas in which mismatches might occur. Furthermore, once mismatches are identified, whether or not developers should take action and adjust the design is not clear in the absence of more general data regarding how the application is used on a large scale.

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 different picture from 10,000 uses of the option out of 1,000,000 uses of the dialog. The first scenario suggests the option might be made default while the second does not. A related issue is the need for more general data against which to compare specific data collection results. This is the *baseline problem.* 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. Thus, generic usage information should be captured to provide developers with a better sense of the "big picture" of how applications are used.

6 Conclusions

We have presented a theory to motivate and guide usage data collection, an architecture capable of supporting larger scale collection (than currently possible in usability tests) of higher quality data (than currently possible in beta tests), and real-word experience suggesting the proposed approach is complementary to existing usability practice.

While our initial intent was to support usability evaluations directly, our experience suggests that automated techniques for capturing usage information are better suited to capturing indicators of the "big picture" of how applications are used than in identifying subtle, nuanced, and unexpected usability issues. However, 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. It was reported to us 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 effort for the next release. However, it is not information that can be reliably collected in the usability lab.

We are in the process of generalizing our approach to capture data regarding the operation of arbitrary software systems implemented in a component- and event-based architectural style, such as the JavaBeans™ standard. We also believe that long-term, aggregate usage data may be useful in supporting adaptive UI and application behavior as well as "smarter" delivery of help, suggestions, and assistance. Current approaches to this problem, for example, Microsoft's Office Assistant™, typically do not have a long-term picture of use, particularly not involving multiple users. We are also seeking development projects in which we might evaluate our research more formally.

References

Badre, A.N. & Santos, P.J. (1991). A knowledge-based system for capturing human-computer interaction events: CHIME. Tech Report GIT-GVU-91-21.

Badre, A.N., Guzdial, M., Hudson, S.E., & Santos, P.J. (1995). A user interface evaluation environment using synchronized video, visualizations, and event trace data. *Journal of Software Quality,* Vol. 4.

Castillo, J.C. & Hartson, H.R. (1997). Remote usability evaluation site. http://miso.cs.vt.edu/~usab/remote/.

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

Cook, R., Kay, J., Ryan, G., & Thomas, R.C. (1995). A toolkit for appraising the long-term usability of a text editor. *Software Quality Journal,* Vol. 4, No. 2.

Cusumano, M.A. & Selby, R.W. (1995). *Microsoft Secrets: How the world's most powerful software company creates technology, shapes markets, and manages people.* The Free Press, New York, NY.

Ergolight Usability Software (1998). Product web pages. http://www.ergolight.co.il/.

Girgensohn, A., Redmiles, D.F., & Shipman, F.M. III. (1994). Agent-based support for communication between developers and users in software design. In *Proceedings of KBSE'94.*

Hartson, H.R., Castillo, J.C., Kelso, J., & Neale, W.C. (1996). Remote evaluation: the network as an extension of the usability laboratory. In *Proceedings of CHI'96.*

Hilbert, D.M. & Redmiles, D.F. (2000). Extracting usability information from user interface events. *ACM Computing Surveys* (To Appear).

Hilbert, D.M. (1999). Large-scale collection of application usage data and user feedback to inform interactive software development. Doctoral Dissertation. Technical Report UCI-ICS-99-42. http://www.ics.uci.edu/~dhilbert/papers/.

Hoiem, D.E. & Sullivan, K.D. (1994). 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.

Kay, J. & Thomas, R.C. (1995). Studying long-term system use. *Communications of the ACM,* Vol. 38, No. 7.

Lecerof, A. & Paterno, F. (1998). Automatic support for usability evaluation. *IEEE Transactions on Software Engineering,* Vol. 24, No. 10.

Smilowitz, E.D., Darnell, M.J., & Benson, A.E. (1994). 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.

Swallow, J., Hameluck, D., & Carey, T. (1997). User interface instrumentation for usability analysis: A case study. In *Proceedings of Cascon'97.*

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