

Why Let Perfectly Good Usability Data Go to Waste?

David M. Hilbert

David F. Redmiles

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
+1 714 824 3100
{dhilbert,redmiles}@ics.uci.edu

ABSTRACT

The World Wide Web enables cheap, rapid, and large-scale distribution of software for evaluation purposes. However, current techniques for collecting usability data have not kept pace with the opportunities presented by Web-based deployment. This paper presents a technique for remotely capturing meaningful usability data that is currently going to waste. This approach has the potential of improving user-developer communication, increasing user involvement in design, and promoting a more empirically grounded development process. All of this might potentially be performed on a large and ongoing basis over the Internet.

Keywords

Remote usability evaluation, Internet-scale usability data collection, expectation-driven user interface monitoring

1 INTRODUCTION

The Internet and World-Wide-Web make it possible to rapidly distribute prototypes and beta releases to large numbers of users at low cost. In principle, the Internet could become a large-scale test-bed for gathering usability data with actual users of the applications being evaluated. In practice, however, this is difficult due to the number and distribution of users, the time and labor involved in collecting data, the lack of scalable tools for automatic data collection, and the lack of proper incentives to support high-quality voluntary data collection on the part of users. As a consequence, most usability evaluations are limited to relatively small scale tests in the usability lab, and feedback from beta tests is typically reported manually by beta testers themselves. Since data are reported manually, and because beta testers are usually more interested in getting their work done than paying the price of problem reporting while vendors receive most of the benefit, the quantity and quality of data is limited. Typically only the most obvious or unrecoverable errors are reported.

Nevertheless, beta tests appear to offer good opportunities for collecting useful usability data [7]. Smilowitz and colleagues report the results of an experiment in which traditional usability testing was compared against beta tests and forum tests to determine the relative effectiveness of each technique at identifying software usability problems. The beta test condition — in which participants record usability problems as they arise while using the software to do their work — was

shown to be quite effective at identifying usability problems. A later case study performed by Hartson and associates, using a remote data collection technique, seemed to support these results [3].

While the number of usability problems identified in the lab and beta test conditions was roughly equal, the number of common problems identified by both was rather small compared to the large number of unique problems identified in each. This suggests that lab and beta tests uncover different types of problems and are thus complementary. While the beta test appeared to be more cost effective and could be performed under more ecologically “natural” conditions, the lab test did appear to identify more severe problems than did the beta test. In summarizing their results, Smilowitz and colleagues offered the following as one possible explanation for the discrepancy in the types of problems identified in the lab versus beta test conditions:

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 [7].

Another issue mentioned by Smilowitz and colleagues is that the data reported in the beta test condition lacked details regarding user performance and the frequency of occurrences of problems.

In this paper, we propose a semi-automated approach to remote usability data collection that addresses these issues, making it possible to capture higher quality usability data in beta test situations on a much larger scale than is currently possible.

2 A NOVEL APPROACH

Our approach to large-scale remote collection of usability data is based on the notion of monitoring “usage expectations” as users interact with applications.

2.1 Expectations in the Development Process

When developers design systems, they have numerous expectations about how those systems will be used. We call these usage expectations [1]. When developers' expectations don't match actual usage, various problems may ensue, including usability problems.

Developers' expectations are based on their knowledge of the requirements, past experience in developing systems, knowledge of the domain, knowledge of the specific tasks and work environments of users, and past experience in using applications themselves. Some of these expectations are explicitly represented — for example, those specified in requirements, use cases, or in the form of cognitive walkthroughs. Some are implicit — including assumptions about usage that are encoded in screen layout, key assignments, program structure, and user interface libraries.

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. Also, menu and toolbars are typically laid out based on expectations about frequency of use. 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 usage is important in improving usability. Once mismatches are detected, they may be corrected in one of two ways. Developers may change their expectations about usage to better match actual use, thus refining system requirements and eventually making a more usable system. For example, features that were expected to be used rarely, but are used often in practice can be made easier to access. 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 URL's in Netscape Navigator can lead users to omit characters such as "http://".

2.2 Expectation Agents

We propose an approach to remote usability data collection in which expectations are encoded in the form of software agents, called expectation agents, that monitor usage and perform various actions when encapsulated expectations are violated. Figure 1 depicts a software development process in which developers and usability experts: (1) identify usage expectations to be checked as applications are developed, (2) create agents to monitor user interactions, (3) deploy agents to run on users' computers, and (4) receive feedback from agents regarding application usage and mismatches in expected versus actual use. Agents can perform a number of actions including notifying users and/or developers of mismatches, reporting usage statistics as well as the user interface events leading up to and following "critical incidents", providing guidance or suggestions to users, or collecting feedback directly from users [4].

2.3 Usage Scenario

Our prototype expectation-driven event monitoring system (EDEM) provides developers with tools for defining agents, dynamic displays for visualizing the components and events of the interface being monitored as well as agent activity, and

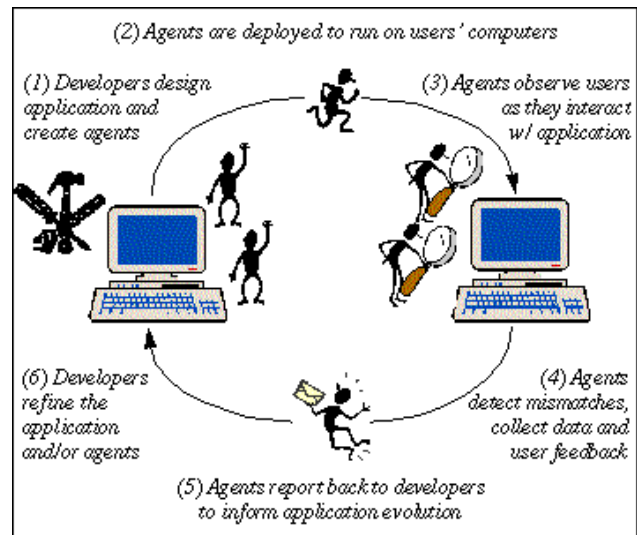


Figure 1. A development process augmented with agents for collecting usability data.

an agent runtime system that allows agents to be downloaded to monitor user interactions on user computers, while reporting data back to centralized or federated groups of developer computers.

To see how EDEM might be used to collect valuable usage information, consider the following scenario.¹ A group of engineers are tasked with designing a web-based user interface to allow users access to a large store of transportation-related information. The interface in this example is modeled after an existing interface (originally written in HTML and JavaScript) that allows users to request information regarding Department of Defense cargo in transit between points of embarkation and debarkation. For example, an officer might use the interface to determine the current location of munitions that he/she ordered for his/her 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 techniques, a prototype implementation of the form is ready for deployment. Figure 2 shows the prototype interface. The designers in this scenario were particularly interested in verifying the expectation that users would not frequently change their "mode of travel" selection in the first section of the form (e.g. "Air", "Ocean", etc.) after having made subsequent selections, since the

1. This scenario is adapted from a demonstration performed by Lockheed Martin C2 Integration Systems within the context of a large-scale, governmental transportation information system based on the Global Transportation Network (GTN). The GTN is a system that gathers, integrates, and distributes transportation-related information and acts as the central clearinghouse of transportation information for the U.S. Department of Defense.

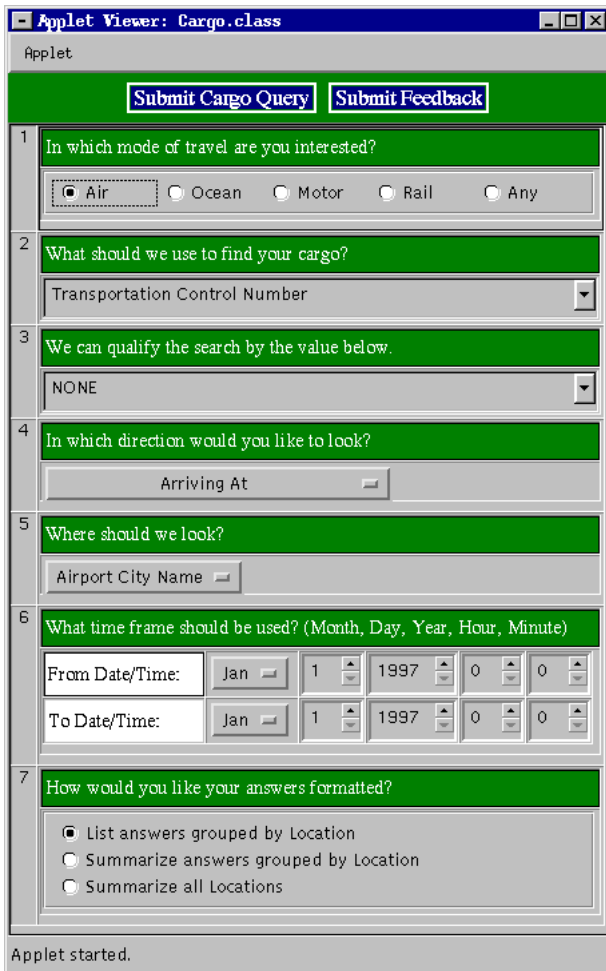


Figure 2. A prototype cargo query interface.

“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 designers made the decision to simply reset all selections to their default values whenever the “mode of travel” selection is reselected.

Figure 3 depicts a simple agent editor that designers can use to author agents without writing code. In Figure 3, (top) the developer expresses interest in detecting when the user selects the “Air” button in the “mode of travel” section, and adds this event to an agent (bottom) that “fires” whenever the user edits any of the buttons in 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. These agents are then downloaded to users’ computers (automatically upon application start-up) where they monitor user interactions and report data back to developers when expectations are violated by actual usage.

In this case, the designers configured the agent to indicate to users that it had detected a violation. Users were then given the option (using EDEM facilities) to request more information describing why the agent had fired, and to respond via email with comments if they desired. The agent

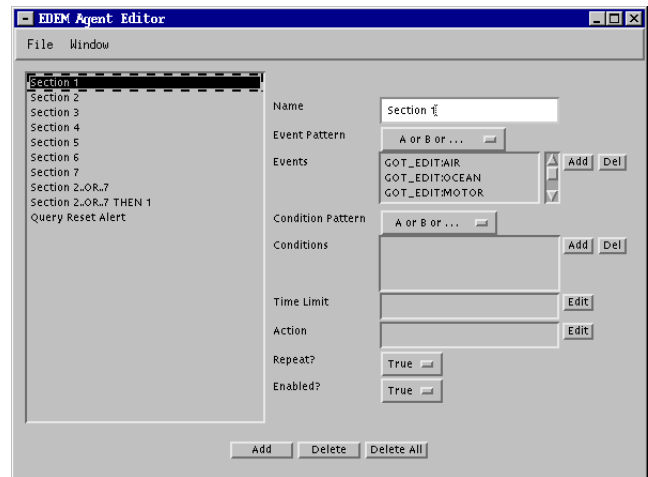
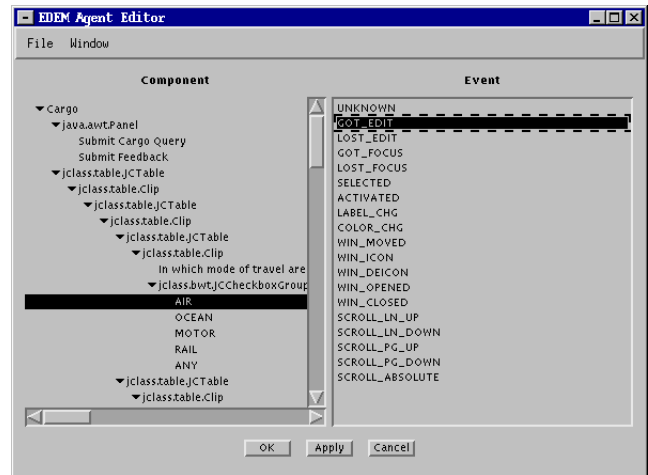


Figure 3. A simple agent editor.

then reported a log of all violations unobtrusively via email each time the applications was exited. Collected data and user responses were emailed to a help desk where they were 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 usage information 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 an expectation agent. Namely, one might argue, the application should automatically detect which selections must be reselected and direct the user to reselect only those values. To illustrate how this objection misses the mark, let us assume that one of the users actually responds to the agent with exactly this suggestion. After reviewing the agent-collected feedback, the engineers consider the suggestion, but unsure of whether to implement it (due to its impact on the design, implementation, and test plans), decide to review the log of expectation violations. The log, which documents over a month of use with over 100 users, indicates that this problem has only occurred 5 times, and always with the same user. As a result, the developers decide to put the change request on hold.

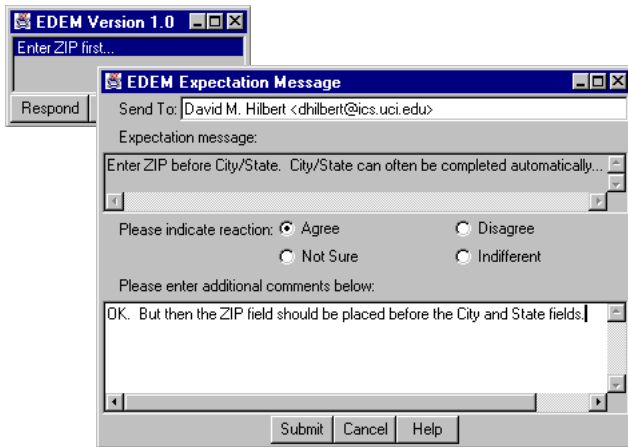


Figure 4. An expectation agent message dialog.

The ability to base design decisions on empirical data in this way is one of the key contributions of this approach. Another important contribution is the explicit treatment of usage expectations in the development process. Treating usage expectations explicitly helps developers think more clearly about the implications of design decisions. Because expectations can be expressed in terms of user interactions, they can be monitored automatically, thereby allowing information to be gathered on a potentially large scale. Finally, expectations provide a principled way of focusing data collection so that data is only collected surrounding “critical incidents” in which usability problems have actually occurred.

3 BENEFITS

3.1 Improved Communication

When agents detect potential usability problems, they may report data to developers unobtrusively, or they may be used to prompt users to communicate with developers. Figure 4 depicts a user responding to an agent message that occurred while the user was filling out a hypothetical phone service provisioning form. See [6] for a more detailed account. The same facilities may also be activated directly by users to volunteer information regarding “critical incidents” not detected by agents. Communication may be synchronous or asynchronous, via voice, video, or electronic mail. The appropriate communication policy will depend on the development situation. For instance, a direct video link might work well in small-scale, in-house development situations, while asynchronous policies might be preferable in large-scale, Web-based product development. When users greatly outnumber developers, information gathered from agents will need to be filtered through information management mechanisms before being presented to developers. Mediator roles [2] may need to be established to manage communication between users and developers.

3.2 Decoupling Data Collection and Application Code

Expectation agents are currently represented as instances of a simple Java™ class with parameters corresponding to triggers, guards, and actions. Triggers are specified in terms of user interface event patterns that are continually checked as

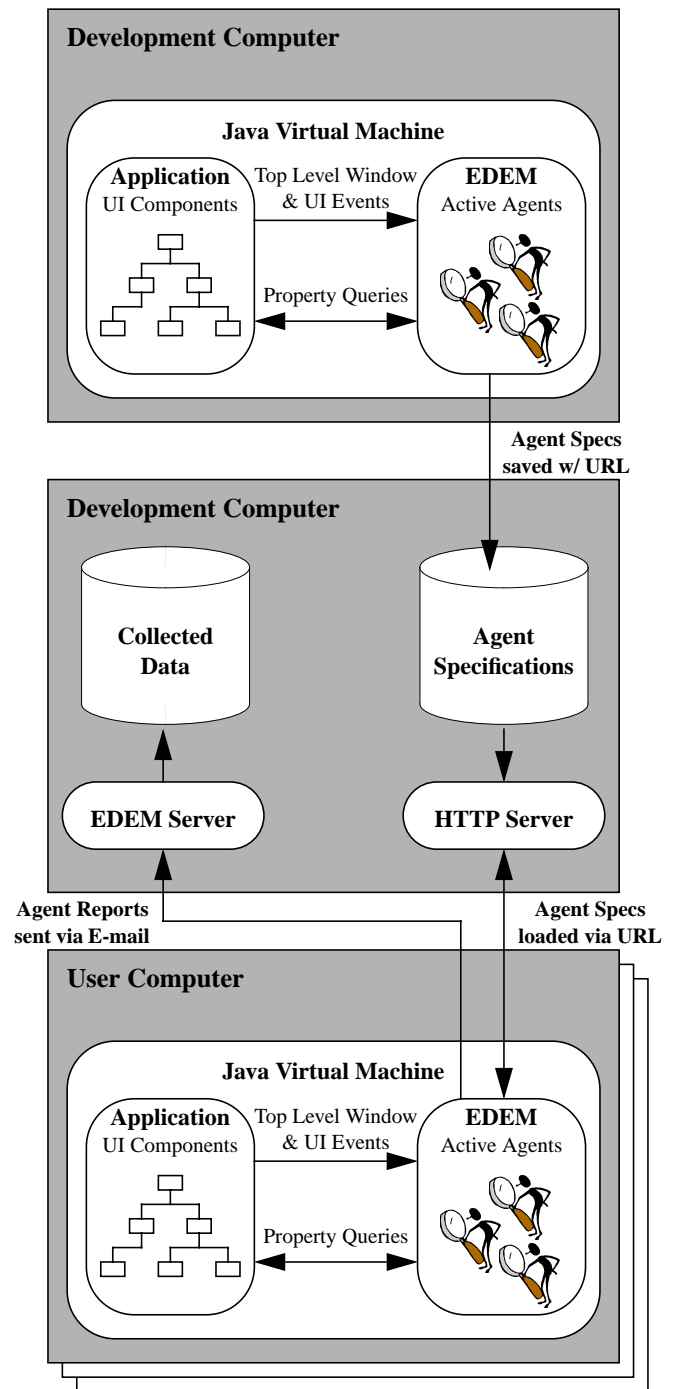


Figure 5. The EDEM architecture.

users interact with the application. Guards are specified in terms of predicates involving user interface component state variables that are only checked once an agent trigger has been activated. Actions may include arbitrary code, but usually involve pre-supplied actions such as generating higher level events for further hierarchical event processing (explained below), interacting with users to provide suggestions and/or collect feedback, and reporting data back to developers.

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 server computer. See Figure 5.

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 agents are instantiated on the user's computer. A standard HTTP server is used to field requests for agent specifications and a standard email protocol is used to send agent reports back to development computers. An EDEM server is used to compile and store agent collected data for later analysis. Agents may therefore be modified, added, and deleted incrementally without affecting deployment of the application that is being evaluated.

This architecture provides a general solution for allowing monitoring code to evolve flexibly in a large-scale, distributed system, without requiring the systems being monitored to be modified when monitoring needs change. Because our approach allows agents to be deployed incrementally, investment in data collection is incremental, and the number of agents can be kept down by focusing on only a limited number of usability questions at any given time.

3.3 Event Filtering and Abstraction

While separating monitoring code from application code is important in allowing monitoring to evolve without impacting application deployment, we do not enforce a separation between the collection of data (typically performed by instrumentation) and filtering and abstraction of the data (typically performed manually by analysts after data have been collected). This is because large-scale remote use demands that data be filtered close to the source to avoid undue network traffic. Placing filtering in deployed applications in the form of agents does not affect application deployment because our architecture allows agents to be modified dynamically as new data needs arise without impacting applications, as described above.

Filtering is accomplished by allowing agents to perform event abstraction. Instead of reporting every event that occurs, agents detect significant patterns of lower level events and generate higher level events for use in further processing. It is therefore possible to compose agents hierarchically to detect patterns of events at increasing levels of abstraction. When an agent detects a pre-specified pattern of lower level events, a higher level event is automatically generated (the "FIRED" event) that can be detected by other agents. This allows a multi-level model of events to be constructed in which higher level, abstract events are specified in terms of combinations of lower level events. A multi-level event model for usability data collection has been implemented using this approach and is described in [4]. Agent output is logged during execution and sent back to development computers via E-mail when the application of interest is exited.

By allowing agents to perform event abstraction close to the source, event data can be filtered before being sent across the network. Second, by allowing higher level events to be specified in terms of lower level events, event data can be collected and analyzed at multiple levels of abstraction.

4 EVALUATION

It is important to evaluate whether and to what extent the data collected by agents is subsequently useful in design improvements. It is also important to characterize the types of

problems detected using this approach in comparison to traditional lab testing or beta testing without the help of EDEM. It is also important to weigh the benefits of collecting usability data against the costs of authoring and maintaining agents. Our initial experience with the Lockheed demonstration project (described in the Usage Scenario) suggests that the effort and expertise required to author agents is not extensive, and that significant data can nonetheless be captured. The most difficult part was indicating to the demonstration development team how EDEM might be used in this context. There were also some initial difficulties in understanding how to specify event patterns. 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". EDEM 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 research systems. While these initial results are encouraging, further evaluation with quantifiable results is planned for the future.

5 RELATED WORK

Other researchers have begun to explore remote usability evaluation using the Internet [3]. However, filtering and reporting of data is only partially automated in that users must be trained to identify "critical incidents" themselves, and then press a "report" button which sends data about events immediately preceding and following the user-identified incidents back to experimenters. This is useful and is included as a feature of EDEM, however, users are often unaware of when their actions violate developers' expectations [7]. Expectation agents are thus indispensable in detecting critical incidents that would otherwise go undetected by users. See [5] and [6] for comparisons to other related work.

6 CONCLUSIONS

The main contributions of this work include an expectation-driven approach to user interface event monitoring and an agent-based architecture that together make large-scale collection of usability data on the Internet a practical possibility. Initial experience with the Global Transportation Network demonstration project indicates that valuable usage data can be captured with only modest investment on the part of developers.

By treating usage expectations explicitly in the development process, we provide a principled way of focusing data collection. By encapsulating data collection code within expectation agents, we allow data collection to evolve flexibly without impacting the deployment of the applications being monitored. Finally, by allowing expectation agents to perform event abstraction, we allow data to be filtered in a scalable way, reducing network bandwidth requirements, and allowing data collection to address events at multiple levels of abstraction. This approach has the potential of improving user-developer communication, increasing user involvement in development, and allowing developers to capture perfectly good usability data that is currently going to waste. All of this could potentially be performed on a large and ongoing basis over the Internet.

ACKNOWLEDGMENTS

The authors would like to thank J. Robbins, A. Girgensohn, F. Shipman, A. Lee, and A. Turner who worked on precursors to this work and continue to provide insight and support. This work is financially supported by the National Science Foundation, grant number CCR-9624846.

REFERENCES

1. A. Girgensohn, D.F. Redmiles, and F.M. Shipman III. Agent-Based Support for Communication between Developers and Users in Software Design. In *Proceedings of the Knowledge-Based Software Engineering Conference 1994*.
2. J. Grudin. Interactive Systems: Bridging the Gaps between Developers and Users. *IEEE Computer*. April, 1991.
3. H.R. Hartson, J.C. Castillo, J. Kelso, and W.C. Neale. Remote Evaluation: The Network as an Extension of the Usability Laboratory. In *Proceedings of CHI'96*, 1996.
4. D.M. Hilbert, J.E. Robbins, and D.F. Redmiles. Supporting Ongoing User Involvement in Development via Expectation-Driven Event Monitoring. Technical Report UCI-ICS-97-19, Department of Information and Computer Science, University of California, Irvine, May 1997.
5. D.M. Hilbert and D.F. Redmiles. An Approach to Large-Scale Collection of Application Usage Data Over the Internet. In *Proceedings of the 20th International Conference on Software Engineering 1998*.
6. D.M. Hilbert and D.F. Redmiles. Agents for Collecting Application Usage Data Over the Internet. In *Proceedings of the Second International Conference on Autonomous Agents 1998*.
7. E.D. Smilowitz, M. J. Darnell, A.E. Benson. Are we overlooking some usability testing methods? A comparison of lab, beta, and forum tests. *Behaviour and Information Technology, Usability Laboratories Special Issue* (Ed.) J. Nielsen, Vol.13, No.1 & 2, 1994.