Supporting Ongoing User Involvement in Development via Expectation-Driven Event Monitoring

David M. Hilbert, Jason E. Robbins, David F. Redmiles
{dhilbert,jrobbins,redmiles}@ics.uci.edu

Technical Report UCI-ICS-97-19
Department of Information and Computer Science
University of California, Irvine
Irvine, California, 92697-3425
April 1997

ABSTRACT

Involving end users in the development of interactive systems increases the likelihood those systems will be useful and usable. User involvement, however, is both time and resource intensive. Internet- and World-Wide-Web-based software release models have magnified these problems. At the same time, these practices have begun to blur the distinction between development and use and, in so doing, have provided developers with unprecedented, and currently underutilized, opportunities for increasing user involvement. We propose an approach — based on expectation-driven event monitoring and expectation agents — that leverages these opportunities to support ongoing user involvement in the software development process.

Keywords event monitoring, expectation agents, usability engineering, interactive systems, software engineering

1. INTRODUCTION

Involving end users in the development of interactive systems increases the likelihood those systems will be useful and usable (Nielsen 1993, Baecker 1995). User involvement, however, is both time and resource intensive. New development practices pose challenges to traditional ideas about how to involve users. Time-to-market has become so critical that user involvement is sometimes sacrificed altogether. If a development organization does not make this sacrifice, by the time it has a product ready for release, the market may already have become locked into a less usable competing product (Arthur 1996).

New software release models now common on the Internet and World-Wide-Web have magnified these pressures. Large-scale beta releases are now used to reach as many potential customers as possible, as early as possible. The amount of time spent on development prior to release has decreased. Pressure to produce frequent upgrades also reduces time available for involving users. While exacerbating the challenges, these practices also present unprecedented opportunities for increasing user involvement.

In the past, most of the development cycle was devoted to preparing a product for beta release. Significantly less time was spent on transitioning it from beta to product release. Due to increased pressures to reach beta customers early, less of the development cycle is being spent preparing the beta, and more time is being spent in the period between beta and product release. As a result, the distinction between development and use is breaking down. The product is already being used during the preponderance of the development cycle. Reduced cycle time between versions also contributes to this effect. Since new versions are constantly being released, the product is essentially always in the development phase, as it is being used. Thus, development and use are concurrent, however user involvement is often limited due to lack of effective tools and techniques for incorporating user feedback.

We offer an approach to address this problem — based on distributed user event monitoring — that takes advantage of the fact that usage and development are concurrent in order to support ongoing user involvement. Usage data is collected from users and can be used by developers in making decisions about potential changes or enhancements. Thus, our approach contributes to an empirically guided development process based on information gathered from actual usage.

Section 2 discusses how usage expectations can be treated explicitly in the software development process, and how this can help improve the process and contribute to the development of more usable systems. Section 3 describes an implementation of this approach, and discusses how it goes beyond traditional event monitoring in supporting extended, or indefinite, user involvement in development. This is followed by Discussion, Challenges, and Summary and Conclusions sections.

This paper is expected to be of particular interest to developers of interactive applications, researchers in software development, researchers in applications of autonomous agents, and anyone interested in new tools for capturing user interactions. A working prototype has been implemented in Java™, and is scheduled for beta release in first quarter 1998.

2. APPROACH

Usability breakdowns occur when developers' expectations about system usage do not match users' expectations. Several benefits can be realized when these mismatches are detected and resolved.

A *usage expectation* determines whether a given interaction (e.g., a sequence of mouse clicks) is expected or unexpected. For example, a developer may hold the usage expectation that users will fill in forms from top to bottom with minor variation, while a user may hold the expectation that independent sections may be filled out in any order.

Figure 1 shows a conceptual picture of expectations held by two groups of stakeholders (developers and users) and expectations encoded in the system being used. Each lowercase "e" represents a tacit expectation held in the mind of a person or in the code of a program. Developers get their expectations from their knowledge of the requirements, past experience in developing systems, domain knowledge, and past experience in using applications themselves. Users get their expectations from domain knowledge, past experience using applications, and interactions with the system at hand. The software system embodies implicit assumptions about usage that are encoded in screen layout, key assignments, program structure, and user interface libraries. Each uppercase "E" represents an explicit expectation. Several usability methods seek to make implicit expectations of developers and users explicit (e.g., cognitive walk-throughs, participatory design, and use cases). Expectations embedded in the system can be made explicit though representations that we discuss below. Many expectations will remain implicit despite these methods. We can treat such expectations as unknowns and attempt to detect mismatches by comparing observed usage against expectations that *have* been made explicit.

Once a mismatch between users' and developers' expectations is detected it can be corrected
1 in one of two ways. Developers can change their expectations about usage to better match users' expectations, thus refining the system requirements and eventually making a more usable system. For example, features that were

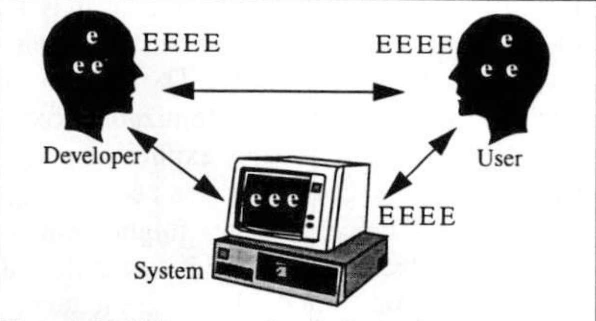


Figure 1. Usability expectations in the development process.

expected to be used rarely, but are used often in practice should be made easier to access. Alter- 2
natively, users can adjust their expectations to better match those of developers, thus learning how to use the existing system more effectively. Learning that they are not expected to type full URL's in Netscape Navigator™ can lead users to omit characters such as "http://".

When developers expectations do not match the expectations embedded in the system, they can be made to match by adjusting one or the other. The developers can change (or debug) the system to make the expectations embodied in it conform to theirs. For example, the developer may remove a function from the toolbar if it is expected to be used infrequently, or add accelerator keys to functions that are expected to be used often. Alternatively, the developers can adopt the current expectations embedded in the system if they are too hard to change, or if they are based on usability knowledge embedded in libraries that enforce user interface guidelines. For example, the system may be easier to implement if it uses standard accelerator key assignments regardless of how often the developer expects a function to be used.

When a user's expectations do not meet those embedded in the system, the user may adapt to the system, or the system may adapt to the user. Users adapt to the system when they are learning how to use it or when the tool enforces proper usage as defined by stakeholders such as customers, standards organizations, or regulatory agencies. For example, the user may take breaks from typing every 50 minutes if the system reminds them to do so. The system may adapt to the user through customizable expectations that are determined by explicit or implicit user preferences. For example, a user writing in a specific technical domain might configure their spell checker to be used very differently than users writing standard business letters.

In each case, detecting a breakdown or difference in expectations is useful, but aligning expectations requires knowledge of the other expectations and the specific differences. For developers to learn about users' expectations they need specific details of actual usage, including context, history, timing, and intent. For users to learn of developers' expectations they need clear documentation of the intended system operation and rationale to be presented to them at the time of breakdown. In either case, dialog between users and developers can help clarify and expose expectations.

3. IMPLEMENTATION

We support detecting and resolving mismatches in expectations (as described above) by allowing developers to specify their usability expectations in terms of user interface events. These expectations are encoded in the form of *expectation agents*, or EA's, that continually monitor usage of the application and perform various actions when encapsulated expectations are violated.

In order to support this functionality, expectation agents need access to events occurring in an application's user interface. However, in order for expectations to be expressed at an appropriate level of abstraction, they need access to higher level events than are produced by the window system. As a result, expectation agents are implemented on top of an expectation-driven event monitoring substrate that provides a multi-level event model.

Other authors have proposed event monitoring as a means for collecting usability data, however, existing approaches often suffer from one or more of the following limitations: (1) low-level semantics: events are captured and analyzed at the window system level, or just slightly above (Chen 1990), (2) decontextualization: analysis is done post-hoc on raw event traces – potentially relevant contextual cues are lost. (3) "one-way" communication: data flows from users to developers who must then infer meaning – no "dialogue" is established to facilitate mutual understanding, (4) batch orientation: hypothesis formation and analysis is performed after large amounts of (potentially irrelevant) data have been collected – no means for hypotheses to be analyzed and action taken while users are engaged. (5) privacy issues: arbitrary events are collected without any explicit constraints on the purposes of collection – no way to provide users with discretionary control over what information is collected and what information is kept private.

In the following subsections we discuss how our implementation addresses these issues and goes beyond existing approaches in supporting user involvement in development. We describe our implementation in terms the following key benefits: (1) *multi-level event model*: allowing agents to compare usability expectations against actual usage at reasonable levels of abstraction, (2) *contextualization*: taking action and collecting information while users are engaged in using the application, (3) *two-way communication*: helping initiate dialog between users and developers when breakdowns occur, and finally, (4) *specializable monitoring and analysis*: promoting a shift from batch-oriented data collection and analysis to hypotheses-guided collection and analysis.

Goal/Problem-Related
(e.g. ordering new software)

Domain/Task-Related
(e.g. providing billing information)

Abstract Interaction Level
(e.g. providing values in input fields)

Application Level
(e.g. shifts in editing attention)

Window System Events
(e.g. shifts in input focus, key events)

Input Device Events
(e.g. hardware generated key or mouse interrupts)

Physical Events
(e.g. fingers pressing keys or hand moving mouse)

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

3.1. A Multi-Level Event Model

We propose a multi-level event model to allow event monitoring to be raised to the level of expectations (Figure 2). At the lowest level are *physical events*, such as pressing keys with one's fingers or moving the mouse with one's hand. *Input device events*, such as key and mouse interrupts, are generated by hardware in response to physical events. *Window system events* associate input device events with windows and widgets on the screen.

Window system events are the lowest level events that EA's can monitor. Events at this level include button presses, list and menu selections, focus events in input fields, and window movements and resizing. An EA could, for example, perform input field validation when the "Submit" button is pressed on a web-based input form.

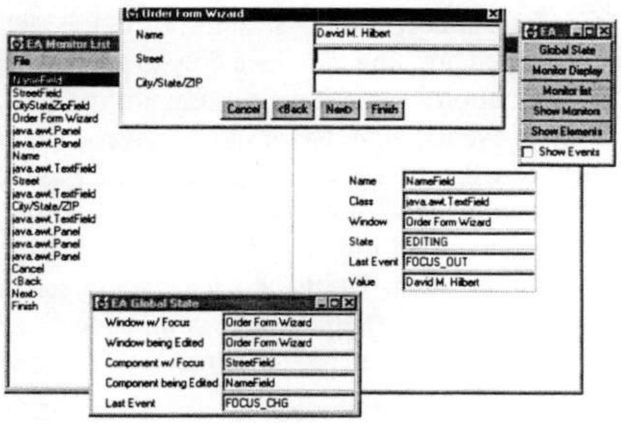
Application level events are generated by the expectation-driven event monitoring substrate based on computations involving window system events and global window system state. Events at this level are intended to indicate changes in the application interface that correlate with salient shifts in the users' state of mind.

Consider a user editing a field at the top of a form, then pressing tab repeatedly to edit a field at the bottom of the form. In terms of window system events, input focus shifted several times between the first and last fields. In terms of application level events, the user's editing attention shifted directly from the top field to the bottom field. Until the user starts actually editing another application component, the monitoring substrate assumes the user's editing attention has not shifted.

Application level events are associated with application components, groups of components, and application windows. To infer that the user has shifted editing attention away from a given component, group of components, or a window, the monitoring substrate must look for editing events in components outside of that component, group of components, or window. The event-monitoring substrate performs the computations required to generate application-level events so that agents can detect such events on particular application components without monitoring all components on the screen. Not only does this simplify individual EA's, it also factors code that would be common across agents, and avoids redundant computation.[1]

Other examples of events that could be defined at the application level include ACCEPT and DISMISS events. These events would indicate changes in the user's state of mind when they perform standard "Ok", "Apply", and "Cancel" operations on dialogs. Other possibilities include a SUBMIT event for web-based form applications and a HELP event to indicate when a user has requested help. This allows EA's to be independent of the particular means by which these functions are invoked, when such independence is desired.

1. Only one line of code must be added to an application in order to allow application level events to be generated. This is necessary to pass window system events to the expectation-driven event monitoring substrate.



Name: David M. Hilbert
Street:
City/State/ZIP:

NameField
StreetField
CityStateZipField
Order Form Wizard
java.awt.Panel
java.awt.Panel
Name
java.awt.TextField
Street
java.awt.TextField
City/State/ZIP
java.awt.TextField
java.awt.Panel
java.awt.Panel
java.awt.Panel
Cancel
<Back
Next>
Finish

Name	NameField
Class	java.awt.TextField
Window	Order Form Wizard
State	EDITING
Last Event	FOCUS_OUT
Value	David M. Hilbert

Window w/ Focus	Order Form Wizard
Window being Edited	Order Form Wizard
Component w/ Focus	StreetField
Component being Edited	NameField
Last Event	FOCUS_CHG

Figure 3. Monitoring of a simple order form wizard.

Abstract interaction events occur when recurring, idiomatic patterns of user interface events (from the window system and/or application levels) are recognized. For example, an abstract interaction event may be generated when a user provides a value by manipulating an application component. If that component were an input field, this would mean the field had been edited, was no longer being edited, and now contains data. The patterns of lower level events that indicate an abstract interaction event such as VALUE_PROVIDED will differ from one type of application component to another, and from one application domain to another. This is why detection of abstract interaction events is not performed directly by the event monitoring substrate. Abstract interaction event EA's can be defined to generate these events when the lower level events indicating them have occurred, so that other interested (or higher level) agents can use them in their own event monitoring.

In order for EA's to recognize abstract interaction events, they need access to arbitrary local and/or global state to keep track of things such as time, sequencing, and so on (e.g., to keep track of position in a state machine representation of the compound event). In order for agents to cooperate as described, they need to be able to generate events that can be detected by one another

Domain/task-related and *goal/problem-related events* are at the highest levels. Unlike previous levels, these events indicate progress in the user's tasks and goals. Detecting these events is straightforward when interfaces provide explicit support for structuring tasks or indicating goals. For example, Wizards in Microsoft Word™ lead users through a sequence of steps in a predefined task. EA's can easily recognize the user's progress by recognizing simple lower level events, such as button presses on the "Next" button. In other cases, task and goal related events might be detected by EA's working individually or in groups to recognize combinations of lower level events. For example, the goal of placing an order includes the task of providing customer contact information. An expectation agent could recognize the task-related event CONTACT_INFO_PROVIDED by recognizing the VALUE_PROVIDED abstract interaction events generated for every field in the contact information section of the form. In order to allow EA's to understand task and goal related events within the context of a broader process, the event monitoring substrate could be integrated with process and work-flow modeling tools.

Our multi-level event model provides a hierarchy of abstraction not provided by traditional event monitoring approaches. Figure 3 shows a simple wizard for filling out an order form that has been connected to the event monitoring substrate. Notice that while input focus has shifted from the name field to the street field, the event monitoring substrate is indicating that the user's editing attention has not yet been confirmed to have shifted.

3.2. Contextualization

Currently, users and developers bear full responsibly for recognizing when breakdowns occur, determining the reasons for the breakdown, and deciding how to recover. Because EA's operate within the context of use, they can assist users and developers in making these determinations.

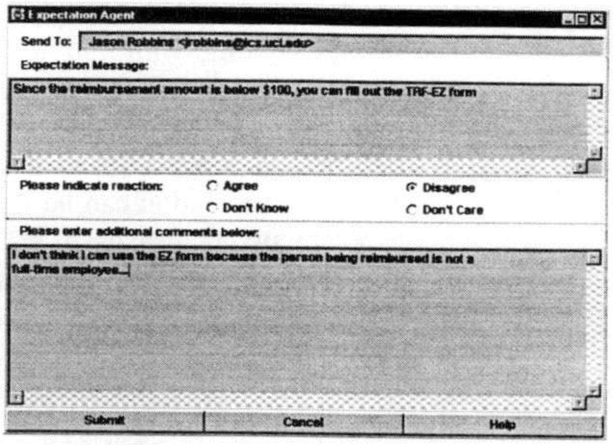
When a breakdown occurs, EA's can provide developers with important contextual information such as system state and event history. They may also collect information from users regarding the reasoning and incidents leading up to breakdowns — while that information is still fresh in users' minds. When breakdowns are due to errors in the code, EA's can help provide developers with much richer contextual information for bug reporting purposes than has typically been possible. EA's can help make external bug reports as useful as internally generated bug reports.

Another benefit of EA's is that they can operate in real usage contexts. Because they don't significantly affect user interface operation, the environmental context is true to actual usage. Also, since monitoring is unobtrusive, EA's are less likely than direct observation methods to influence users' behavior.

3.3. Two-Way Communication

When unexpected breakdowns occur, it may not be enough to simply provide context. Dialogue between users and developers may need to be established in order to evolve mutual understanding. When an EA detects a breakdown, it can prompt the user to communicate with developers (Figure 4). The same facilities can also be used to volunteer comments when EA's fail to detect breakdowns experienced by the user. Communication can be synchronous or asynchronous, via voice, video, or electronic mail.

Once communication has been initiated, ongoing dialogue between developers and users may continue outside the scope of the agent-based system. The communication policy appropriate



Send To: Jason Robbins <jrobbins@ics.uci.edu>

Expectation Message:

Since the reimbursement amount is below $100, you can fill out the TRF-EZ form

Please indicate reaction: Agree [✓ Disagree] Don't Know Don't Care

Please enter additional comments below:

I don't think I can use the EZ form because the person being reimbursed is not a full-time employee.

Figure 4. An expectation agent message dialog.

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 Web-based product development. When users greatly outnumber developers, information gathered from EA's will need to be filtered through information management mechanisms before being presented to developers. Mediator roles (Grudin 1991) may need to be established to manage communication between users and developers.

3.4. Specializable Monitoring and Analysis

Expectation-driven event monitoring represents a shift from traditional batch-oriented approaches to a more proactive, hypothesis-guided approach. Instead of forming and analyzing hypotheses after large amounts of (potentially irrelevant) data have been collected, data collection can be tuned based on a-priori hypotheses (or expectations) that are analyzed *while* users are engaged. Our approach is hypothesis-guided in that only data, and results of analyses, that are relevant to specific hypotheses (expectations) are reported.[2]

2. EA's can support traditional event monitoring in addition to the expectation-based approach we advocate here.

Specializablity makes monitoring tractable on a larger scale than is possible with traditional approaches. It is scalable in terms of the number of users that can be monitored because it allows analysis to be computed on the client-side. This means that computation can be distributed among potentially thousands of processors, and only relevant data, or results of analyses, need to be reported to developers. Thus, usability information can be captured on a scale that is statistically significant, observations can be categorized a-priori as well as a-posteriori, and factor analysis is facilitated.

Specializable monitoring and analysis can thus contribute to an empirically guided development process. Since developers often have more candidate design changes than time to implement them, effort can be focused on those changes that will benefit the greatest number of users, or resolve the greatest number of non-trivial breakdowns. Also, the impact of proposed changes can be analyzed in terms of how well they "agree" with existing user expectations. For instance, before making a change, a developer could deploy an agent to the current user base to look for user expectations that would be violated by introducing the change.

Since EA's can be dynamically added or removed, investment in EA's can be made incrementally. There is no need to delay deployment of a product until all EA's are in place. Even a single EA can yield some useful feedback.

4. DISCUSSION

In the proceeding sections we have primarily focused on how EA's provide feedback to developers, but EA's can also be used to yield benefits to other stakeholders. Since EA's actions can be customized, they can be used to add new functionality to existing applications. EA's could produce feedback for stakeholders such as system administrators and organizational training staff. For example, if a user cannot access a certain web site, he or she can volunteer comments to the local system administrator and that communication would include the state and history of the application being used.

EA's also provide a way for other stakeholders to express expectations so that their interests are represented to users. EA's can make organizational rules active. For example, when filling in a travel reimbursement form, if the cost of reimbursement is bellow some threshold, the user could be advised to fill in an alternative form (see Figure 4). EA's can help transform the experience of individual users into active guidance for other users. For example, users who print multiple copies of a document can be reminded by other users that photocopying is faster, cheaper, and more courteous.

These benefits do not come without cost. In each case, someone must take the time to express their expectations in the form of EA's. We cannot assume that EA's will be authored in cases where the person paying the cost does not derive some benefit. However, there appear to be several situations in which incentives are in fact in place. Developers can gain important usability and bug-reporting information that justifies the cost of authoring agents. Users may author agents in order to save themselves work by automatically representing their interests to other users. Users may volunteer information if it gives them a chance to express their frustration, especially if their past comments have resulted in noticeable improvements in new versions of the system.

No large-scale monitoring effort should be employed if the risk to user's privacy outweighs the potential benefit. For example, users don't want private email shown to developers, and corporations don't want strategy memos shown to competitors. Our approach can support privacy more fully than traditional event monitoring. First, EA's collect only data needed for specific purposes and reporting is limited to expectation violations. In comparison, traditional event monitoring approaches report all events, potentially permitting data to be used for purposes other than specified. Second, the user can better control violation reporting than event reporting because the reports are fewer and more specific.

Beside asking for permission to send reports, EA's also interact with users to request comments and provide help. Thus they run the risk of distracting the user from the task at hand. Under most circumstances, we advocate a non-disruptive feedback model that gives the user an indication that an agent is requesting their attention but allows the user to continue their task. In cases where users are presented with too many requests for their attention, we have investigated various scheduling and control mechanisms that can be used to limit agent execution and filter agent feedback (Robbins et al. 1996).

5. CHALLENGES

Three challenges in making our approach practical include finding appropriate representations for expressing usage expectations, facilitating authoring of expectation agents by various stakeholders, and addressing maintenance of expectation agents over time.

5.1. Representation

Tacit usage expectations must be converted into an explicit representation before they can be used to evaluate actual usage. Since there are a wide variety of expectations, we seek to employ a variety of representations.

State-based representations are well suited for expressing expectations about sequences of actions regardless of the values of input fields or the state of the system. For example, the expectation that users will fill in fields left to right and top to bottom. State-based agents rely primarily on the order of events at various levels in the multi-level event model. By registering interest in particular events, transitions can be triggered when those events occur. Current technologies for state-based systems are well developed and used in both requirements engineering and coding (Wing 1991).

Rule-based representations (Girgensohn et al. 1994) are well suited for expressing expectations that hold over entire interactions regardless of the order of events. For example, developers might expect users to not fill in fields for both credit card payment and COD (cash on delivery). Rule-based agents rely primarily on the value of input fields and the state of the system. By registering interest in particular fields, these agents can be triggered when those fields change. Current technologies for rule-based systems are also well developed.

Mode-transition-based representations incorporate features of both rule-based and state-based representations. They represent expected behavior as tables of modes (i.e., states) and transitions which are guarded by conditions (i.e., rules). For example, when an airline customer representative is searching for a group of seats on a single flight, they might be expected to enter another query whenever the previous query yielded less available seats than was specified in the "number of travelers" input field. Mode-transition-based technologies have been well developed and are primarily used in requirements engineering (Atlee and Gannon 1993).

Each of these approaches is well-suited to some aspects of expectation agent representation. We hope to provide a range of options that combine these approaches and extend them to make use of interactions between agents.

5.2. Authoring

Stakeholders author expectation agents by using an agent development tool that provides them with means for specifying usage expectations. Since there are a wide variety of stakeholders and expectations, we seek to employ a variety of authoring techniques.

Developers can author EA's simply by writing programming language source code that makes calls to the expectation-based event monitoring substrate. They can make use of existing CASE tools that support the development of state-based, rule-based, or mode-transition-based software.

Users may find tools based on templates or programming-by-demonstration more accessible. (Girgensohn et al. 1994) describes a form for defining agents consisting of a single triggering event, a set of simple conditions combined with an implicit logical-and, and a choice of one of several predefined actions. Demonstrational techniques could be especially accessible to users because such approaches only demand knowledge of how to use the system.

5.3. Maintenance

The intent of our approach is to enhance evolution of usable systems. Any approach encouraging change must consider the cost of changes. Expectation agents are potentially costly to maintain since they depend on the very interactions that the approach seeks to change.

Fortunately, several features mitigate this problem: (1) Agents that look for higher-level events are insulated by the multi-level event model from low-level changes in widget labeling, size, and position. Only those agents dealing specifically with layout issues will need to be modified when the layout changes. (2) When the interface changes there are ways to automatically determine the subset of agents that might be affected, since expectation representations contain references to specific (high- or low-level) events and user interface components. (3) Changes to the interface tend to produce corresponding changes in the expectations. For example, if an interface is being improved to shorten a common interaction by combining or removing widgets, then affected expectations can be modified by combining or removing corresponding states.

6. SUMMARY AND CONCLUSIONS

In summary, we propose a conceptual model of how usage expectations might be treated explicitly in the software development process. We discuss how this could help improve the process and contribute to the development of more usable systems. We describe an implementation — based on expectation-driven event monitoring and expectation agents — that realizes facets of this conceptual model.

Our approach can be used to unobtrusively discover how applications are being used, detect and report discrepancies between actual and expected usage, and report instances of erroneous system behavior, unexpected failures, and exceeded thresholds. When breakdowns or mismatches occur, they can be used to initiate communication between users and developers. Our approach supports an empirically guided development process in which developers can strategically plan user interface changes based on actual usage information.

We believe our technology is well-suited for gathering and analyzing usability information in the context of software testing activities, focused experiments, usability testing, beta testing, and in actual usage (both industrial and in field studies). Once more progress has been made addressing the issues raised in this paper, we believe that much useful usability information that is currently lost can be captured and used by developers and researchers to improve the designs of interactive systems.

7. ACKNOWLEDGEMENTS

The authors are greatly indebted to A. Girgensohn, F. Shipman, A. Lee, and A. Turner who worked on precursors to this paper and continue to provide insight and support. The authors would also like to thank J. Grudin for his thoughtful comments. This work is financially supported by the National Science Foundation, grant number CCR-9624846.

8. REFERENCES

Arthur, W. B. (1996) Increasing Returns and the New World of Business. *Harvard Business Review.* July-August 1996.

Atlee J. M., and Gannon, J. (1993) State-based Model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*. January 1993.

Baecker, R. M., Grudin, J., Buxton, W. A. S., Greenberg S., eds. (1995) *Readings in Human-Computer Interaction: Toward the Year 2000.* Morgan Kaufmann Publishers, Inc. San Francisco, CA, USA.

Chen, J. (1990) Providing Intrinsic Support for User Interface Monitoring. In *Human-Computer Interaction - INTERACT '90.*

Girgensohn, A., Redmiles, D. F., and Shipman, F. M. III. (1994) Agent-Based Support for Communication between Developers and Users in Software Design. In *Proceedings of the Knowledge-Based Software Engineering Conference '94.* Monterey, CA, USA.

Grudin, J. (1991) Interactive systems: bridging the gaps between developers and users. *IEEE Computer.* April, 1991.

J. Nielsen. (1993) *Usability Engineering*. Academic Press, Inc., Cambridge, MA, 1993.

Robbins, J. E., Hilbert D. M., and Redmiles, D. F. (1996) Using Critics to Analyze Evolving Architectures. In *Proceedings of the Second International Software Architecture Workshop*. San Francisco, CA, USA. October, 1996.

Wing, J. A Specifier's Introduction to Formal Methods. *IEEE Computer.* September, 1990.