An Approach to Large-Scale Collection of Application Usage Data Over the Internet

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

Empirical evaluation of software systems in actual usage situations is critical in software engineering. Prototyping, beta testing, and usability testing are widely used to refine system requirements, detect anomalous or unexpected system and user behavior, and to evaluate software usefulness and usability. The World Wide Web enables cheap, rapid, and large-scale distribution of software for evaluation purposes. However, current techniques for collecting usage data have not kept pace with the opportunities presented by Web-based deployment. This paper presents an approach and prototype system that makes large-scale collection of usage data over the Internet a practical possibility. A general framework for comparing software monitoring systems is presented and used to compare the proposed approach to existing techniques.

Keywords
Internet-scale usability data collection, remote usability testing, user interface event monitoring, agent-based architectures, human-computer interaction and software engineering

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 data about application use with actual users of the systems being tested. In practice, however, this can be difficult due to the 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 small scale tests in the usability lab, and feedback from beta testing is typically reported manually by beta testers themselves. Since data are reported manually, and because beta testers pay the price of bug reporting while vendors receive most of the benefit, the quality and quantity of data is limited. Typically only the most obvious or show-stopping problems are identified.

Despite these challenges, large-scale, Internet-based collection of usage data with prototype and beta releases has the potential of providing useful empirical guidance for application development. Data collection is also important beyond initial prototype and beta evaluation stages. For example, data about which application features are most frequently used in practice can suggest which features to optimize as well as how to best focus development and testing effort. Continued collection is also necessary to detect when usage patterns shift, thereby invalidating results of data collected in earlier stages. Ongoing collection is necessary to provide empirical guidance in subsequent application maintenance and enhancement.

We propose an approach to automatic usability data collection that makes ongoing, large-scale use a practical possibility. The specific contributions of our approach include: (a) treating "usage expectations" explicitly in the development process to improve design and focus data collection, (b) a flexible and incrementally evolvable monitoring architecture that separates evolution of monitoring code from evolution of application code, and (c) event abstraction mechanisms embedded within probes to provide distributed filtering and multiple levels of abstraction in collected data. A prototype implementation has been developed that monitors Java^{TM} applets and applications, however, the underlying concepts may be applied to systems developed in any language in which user interface functionality is provided by an event-based windowing system.

Like other forms of experimentation, usability testing involves numerous, interrelated activities including hypothesis formation, data collection, data analysis, and interpretation of results. Each of these activities may be addressed using multiple techniques. This paper focuses on a particular technique, namely automatic usage monitoring, for a specific activity, namely data collection, while acknowledging that no single activity or technique is sufficient in isolation. We address hypothesis formation to a certain degree, but refer readers to existing techniques for analyzing collected data [9][11][15][16][24][25]. Our approach compliments existing techniques for hypothesis formation, data collection, analysis, and interpretation.

We begin by discussing the state of the practice in application usage monitoring. This is followed by discussion of a novel approach for extending this technique to large-scale use on the Internet and a usage scenario to illustrate its application. The following section develops a general framework for comparing software monitoring systems that is used to compare the proposed approach with existing techniques. Finally, the status of a working prototype implementation and its evaluation are discussed, followed by related work and conclusions.

2 APPLICATION USAGE MONITORING

Application usage monitoring is a technique for collecting data about human-computer interactions for the purpose of evaluating application usability. Often referred to as "monitoring" or "logging" techniques in the HCI literature [2][19], usage monitoring involves instrumenting applications (or windowing systems) to log information about user interactions while test subjects complete pre-specified tasks with interactive applications. The data collected by these means are often used in conjunction with other forms of data, such as video and/or experimenters' notes, to identify potential flaws in user interface design. Analysis is often aided by spreadsheets or other more specialized analysis tools, and presented to developers potentially resulting in changes to the system being studied.

Scalability is important in usage monitoring because it impacts who can be monitored (small numbers of laboratory subjects vs. large numbers of actual users), under what circumstances (usability laboratory vs. natural working conditions), and for what duration (short experiments vs. ongoing evaluation). Collecting usability information on a large scale, however, is challenging. Existing tools are not designed for large-scale use. To begin with, many of them do not appropriately separate instrumentation from application code. As a result, independent evolution is not possible. In order to modify the type, format, or amount of data that is captured, the application must be modified and re-delivered to all subjects.

To avoid modifying instrumentation that is intermingled with code, or as a result of inserting probes directly into the windowing system, the practice has been to collect as much data as possible — at very low levels of abstraction — and to defer processing and analysis until after data have been collected. This presents a problem for Internet-scale use. The volume of user interface events generated by a single user engaged in a single session is extremely high. In the context of the Internet, that volume must be multiplied by numerous users, engaged in numerous sessions, at numerous distributed sites. The network load that would be generated by transmitting every mouse movement of even a small percentage of the networked Microsoft Word™ users, for example, would be staggering. Furthermore, experience from testing in software engineering as well as HCI suggests that data should be collected and analyzed at multiple levels of abstraction [27].

3 EXPECTATION-DRIVEN EVENT MONITORING

3.1 Expectations in the Development Process

Before presenting our solution, we begin with a theoretical discussion of expectations in the development process. This discussion suggests a theoretically principled way of focusing data collection and making large-scale usage monitoring feasible.

When developers design systems, they have numerous expectations about how users and the operational environments in which those systems are embedded will behave. We call these usage expectations [10]. When the environment in which a system is deployed or its users behave in unexpected ways, various problems may ensue. These problems can lead to sub-optimal user and system performance, and, in safety or security critical systems, to more dire consequences.

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 that are specified as requirements. Some are implicit, including assumptions about usage that are encoded in screen layout, key assignments, program structure, and user interface libraries.

For example, implicit in the layout of most data entry forms is the expectation that users will complete them from top to bottom, with only minor variation. In laying out menus and toolbars, it is usually expected that frequently used or important functions can be easily recognized and accessed, and that functions 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, frequently fail to be tested adequately.

Several benefits can be realized if mismatches between developers' expectations and actual usage can be detected and resolved. Once a mismatch is detected, it may be corrected in one of two ways. Developers may change their expectations about usage to better match actual use, thus refining the 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 adjust their behavior to better match 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://".

3.2 Expectation Agents

We propose an approach to application usage monitoring that is based on making usage expectations explicit. These expectations are encoded in the form of agents that monitor application usage and perform various actions when encapsulated expectations are violated. Figure 1 depicts a development process in which developers (and/or usability specialists) identify usability expectations to be checked as applications are developed, create agents to monitor user

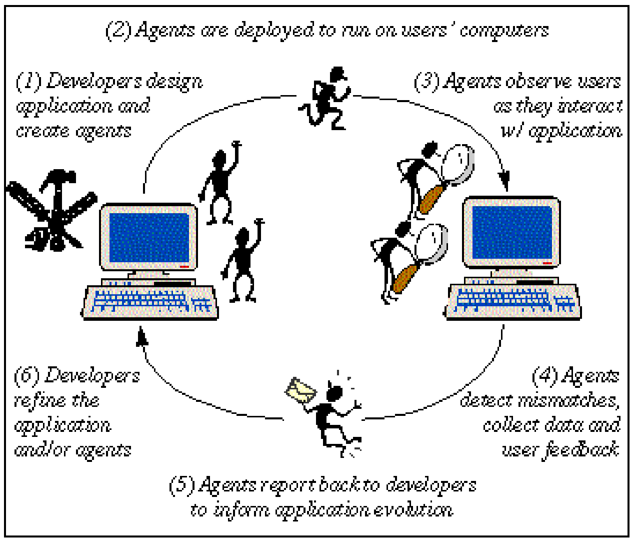

(2) Agents are deployed to run on users' computers
(1) Developers design application and create agents
(3) Agents observe users as they interact w/ application
(6) Developers refine the application and/or agents
(4) Agents detect mismatches, collect data and user feedback
(5) Agents report back to developers to inform application evolution

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

interactions, deploy agents to monitor application usage, and receive feedback from agents regarding mismatches in expected versus actual usage.

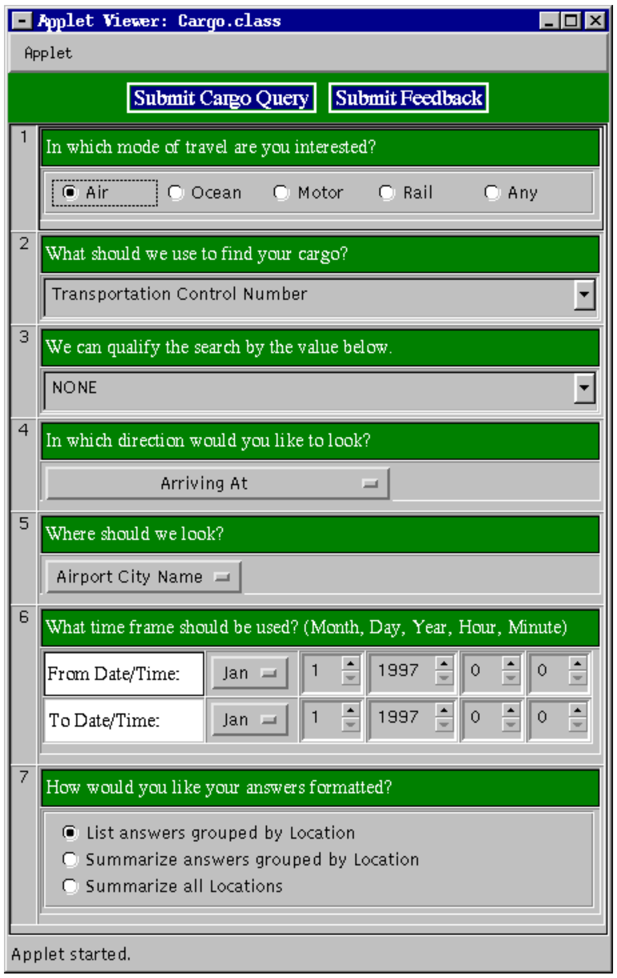
The particular action highlighted in Figure 1 and in this paper in general involves agents reporting data back to developers. However, agents can perform numerous actions including notifying the user and/or developer of mismatches, reporting system state and/or event history for debugging purposes, providing guidance or suggestions to the user, or collecting feedback directly from the user [13].

3.3 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 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 can be used by developers to collect valuable usage information, consider the following usage scenario, which is adapted from a demonstration performed by Lockheed Martin C2 Integration Systems within the context of a large-scale, governmental logistics and transportation information system.

A group of engineers are tasked with designing a web-based user interface to allow end users access to a large store of transportation-related information. The interface in this scenario 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 ordered for his troops in Bosnia.



1 In which mode of travel are you interested?

[✓ Air] Ocean Motor Rail Any

2 What should we use to find your cargo?

Transportation Control Number

3 We can qualify the search by the value below.

NONE

4 In which direction would you like to look?

Arriving At

5 Where should we look?

Airport City Name

6 What time frame should be used? (Month, Day, Year, Hour, Minute)

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

7 How would you like your answers formatted?

[✓ List answers grouped by Location]
Summarize answers grouped by Location
Summarize all Locations

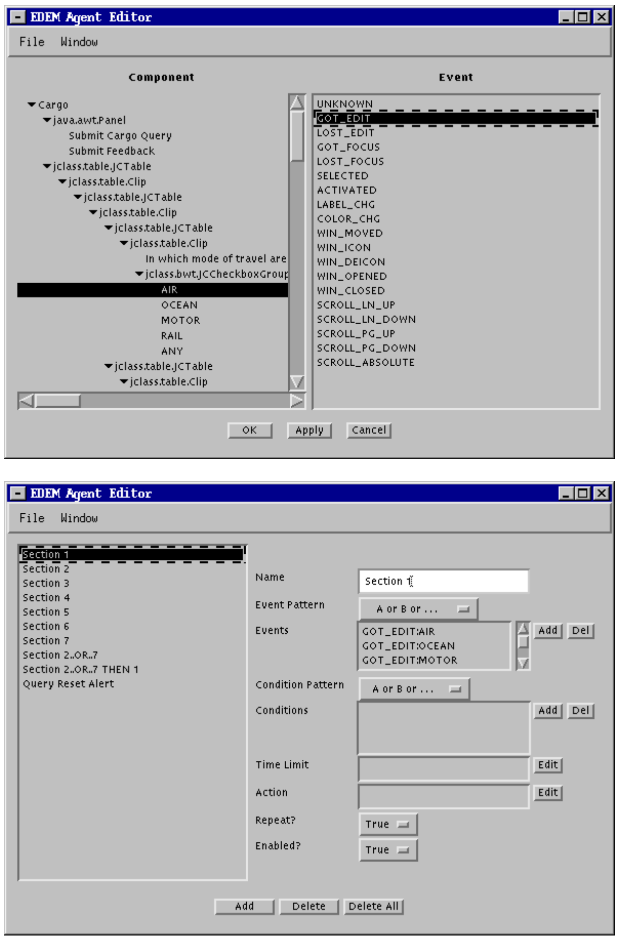
Applet started.

Figure 2. A prototype cargo query interface.

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 engineers 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 "mode of travel" selection affects the choices that are available in subsequent sections. Operating under the expectation that this would not be a common source of problems, the engineers made the design decision to simply reset all selections to their default values whenever the "mode of travel" selection is reselected.



```
Component                          Event
▾Cargo                             UNKNOWN
  ▾java.awt.Panel                  GOT_EDIT
     Submit Cargo Query            LOST_EDIT
     Submit Feedback               GOT_FOCUS
  ▾jclass.table.JCTable            LOST_FOCUS
    ▾jclass.table.Clip             SELECTED
      ▾jclass.table.JCTable        ACTIVATED
        ▾jclass.table.Clip         LABEL_CHG
          ▾jclass.table.JCTable    COLOR_CHG
            ▾jclass.table.Clip     WIN_MOVED
               In which mode of travel are   WIN_ICON
              ▾jclass.bwt.JCCheckboxGroup    WIN_DEICON
                 AIR               WIN_OPENED
                 OCEAN             WIN_CLOSED
                 MOTOR             SCROLL_LN_UP
                 RAIL              SCROLL_LN_DOWN
                 ANY               SCROLL_PG_UP
          ▾jclass.table.JCTable    SCROLL_PG_DOWN
            ▾jclass.table.Clip     SCROLL_ABSOLUTE
```

```
Section 1
Section 2
Section 3
Section 4
Section 5
Section 6
Section 7
Section 2..OR..7
Section 2..OR..7 THEN 1
Query Reset Alert
```

Name	Section 1
Event Pattern	A or B or ...
Events	GOT_EDIT:AIR GOT_EDIT:OCEAN GOT_EDIT:MOTOR
Condition Pattern	A or B or ...
Conditions	
Time Limit	
Action	
Repeat?	True
Enabled?	True

Figure 3. A simple agent editor.

Figure 3 depicts a simple agent editor that developers 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 engineers 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 then reported a log of all violations unobtrusively via email each time the applet 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 applet 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 agent. Namely, 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 current 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.

The ability to base development and management 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 been detected.

4 EVENT MONITORING FRAMEWORK

In this section, we develop a general framework for comparing monitoring systems to help distinguish our approach from existing techniques. Our framework is related in some ways to other frameworks that have been proposed for event-based software integration [3] and internet-scale event observation and notification [22]. However, our framework differs in its focus on monitoring and data collection issues as opposed to tool integration and wide-area messaging issues. Further discussion of how our framework compares to previous frameworks is presented in the "Related Work" section.

4.1 Activity Space v. Event Space

First of all, we distinguish between the phenomena occurring within the system being monitored and the phenomena that is made visible to the outside world by the monitoring system. These may or may not be identical. For example, event monitoring systems frequently emit higher level events based on computations involving lower level activities occurring within the system being monitored. In some cases, the word "event" is used to refer to both the low level, transient system activities being monitored (e.g. user interface events), and the higher level, persistent information subsequently made visible to the outside world. To avoid confusion, we distinguish between objects and activities which reside in "activity space", and entities and events which reside in "event space" (Figure 4).

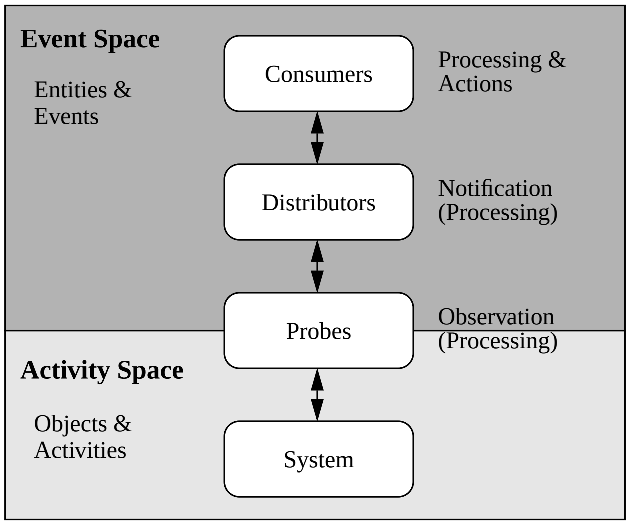


Figure 4. Probes translate information about objects and activities in “activity space” into information about entities and events for use by distributors and consumers in “event space”.

Activity space is comprised of the objects and activities of interest in the system being monitored. Objects may come in “active” forms (e.g. whole systems, subsystems, software agents, software components, programming language modules, and so forth) or “passive” forms (e.g. operating system files and directories, database tables and rows, and so forth). Activities of interest are typically manifested in terms of observable state changes, message passing, method invocations, procedure calls, events, and so forth. Objects and activities of interest are typically transitory and often identified by non-persistent, implementation-dependent identifiers in activity space.

Event space is characterized by events and entities that correspond to objects and activities in activity space. Entities and events, however, have persistent names for use in event-space. The mappings between objects in activity space and entities in event space are sometimes invertible so that objects can be queried or otherwise manipulated from event space. The mappings between activities in activity space and events in event space are frequently not invertible since events may be inferred or computed by compilation of information about activities (with information loss).

4.2 Monitoring Roles

There are three major roles that are typically fulfilled in any sophisticated monitoring approach (Figure 4):

- Probes to capture data about the system being monitored
- Distributors of captured data
- Consumers of data

These roles are logical in that the mechanisms instantiating them may be loosely coupled, tightly coupled, or entirely integrated.

Probes capture information about transitory objects and activities in activity space and translate it into more persistent information about entities and events for use in event space. *Distributors* are responsible for distributing collected information to *consumers*. Probes therefore bridge the gap between activity and event spaces, while distributors and consumers operate mainly in event-space.

4.3 Monitoring Activities

Probes, distributors, and consumers typically engage in the following major activities (Figure 4):

- Observation
- Processing
- Notification
- Actions

Observation involves collecting basic information about objects and activities in activity space. Observation can be achieved through automatic synchronous detection techniques or through polling. Observation is primarily performed by probes for the purpose of making information available to distributors and consumers.

Processing involves performing computations based on basic information about objects and activities or entities and events. Processing may involve pattern-matching, filtering, or aggregation, potentially for the purpose of generating higher level events. Processing is frequently performed by probes and/or distributors for the purpose of filtering notifications as well as providing event abstraction. Consumers also engage in processing, and may ask probes and/or distributors to perform processing on their behalf.

Notification involves letting other interested parties know about observations or results of processing, frequently leading to further observation, processing, notification, and actions. Notification is primarily performed by distributors.

Actions are sometimes performed in response to observations, processing, and notifications. Actions may involve manipulating the system (e.g. to reconfigure it), interacting with other systems (e.g. to store data), manipulating probes or distributors (e.g. to register or cancel interest in events), interacting with humans (e.g. to inform them of critical conditions). Actions are primarily performed by consumers. Consumers may also ask probes and/or distributors to perform certain actions on their behalf.

4.4 Summary

In summary, monitoring can be understood in terms of three major roles and four activities. The roles include: probes, distributors, and consumers. The activities include: observation, processing, notification, and actions. Approaches to monitoring differ in terms of how they instantiate these roles and activities, and to what extend activities can be distributed amongst components fulfilling these roles in an overall monitoring architecture. This has important implications on reusability of monitoring assets, scalability of the monitoring effort, and ultimately the types of evaluation that can be performed.

4.5 Applying the Framework

In terms of the concepts developed in this framework, traditional application usage monitoring can be characterized in the following way. The objects of interest are user interface components (such as text fields, buttons, and selection lists). The activities of interest are user interface events (such as key presses, mouse button presses, and list selections). Observation is typically achieved by inserting probes directly into application code or by tapping into the windowing system's event queue. Probes also act as distributors by writing collected data directly to a file or other stream for later consumption. Processing is typically performed by usability analysts, who are the consumers, after all data have been collected. This may ultimately result in actions involving changes to the system being studied.

The main problems with traditional approaches are that probes are intertwined with application code and processing is deferred until after distribution. Our approach separates probe code from application code and allows processing to occur within probes so that filtering can be performed prior to distribution.

5 IMPLEMENTATION

5.1 Agent Architecture

EDEM agents are currently represented as instances of a simple Java^{TM} template class with parameters corresponding to triggers, guards, and actions. Triggers are specified in terms of user interface event patterns that are continually checked as 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, interacting with users to provide suggestions and/or collect feedback, and finally 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 machine.[1] 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 monitored. Figure 5 depicts a high-level view of the EDEM architecture.

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.

1.See [14] (forthcoming) for an example of a serialized agent.

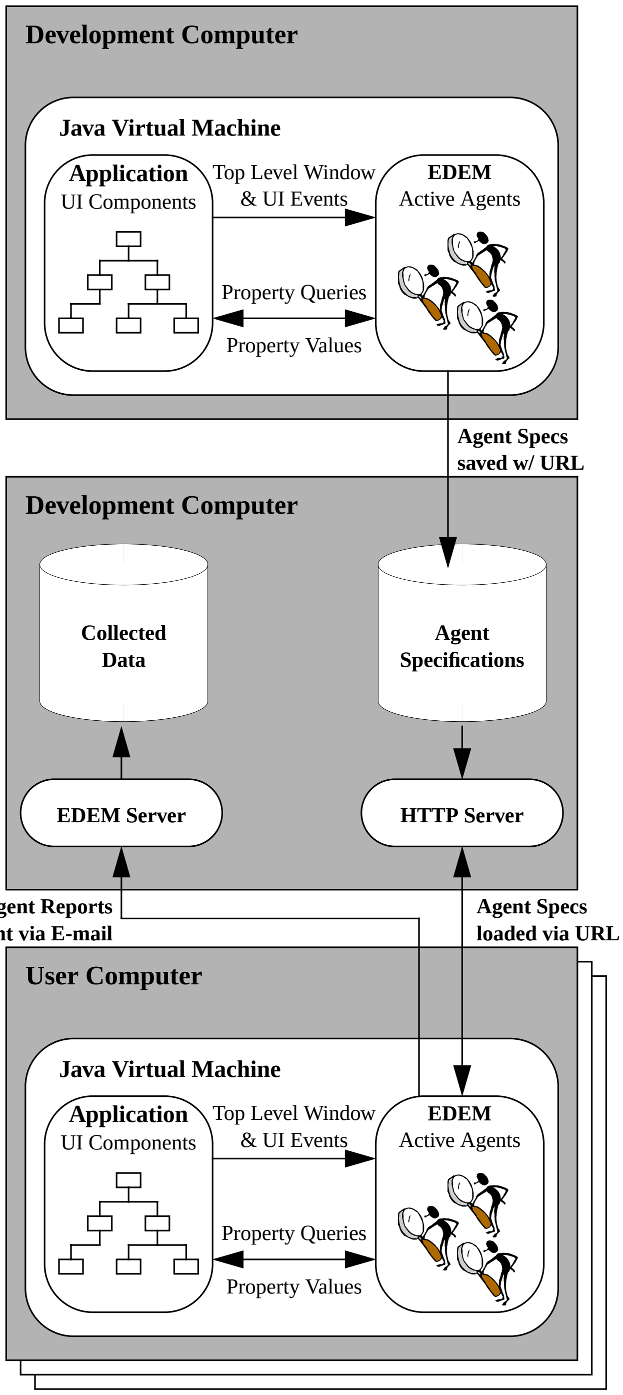


Figure 5. The EDEM architecture.

5.2 Integrating with EDEM

In our prototype Java^{TM} implementation, the top level ID of each application window to be monitored as well as each user interface event is passed to EDEM for processing. This is accomplished through the use of two simple API calls. The first call is made only once when a new application window is created. The second call is made each time the application processes a user interface event. Typically, this only requires two lines of source code to be inserted.[1] There are subtleties involved in automatically mapping the transient, implementation-dependent IDs of user interface components to persistent names for use in monitoring. We overcome this by allowing the developer to provide a name, in code, for each component that is expected to be prominent in monitoring.[2] Once this has been accomplished, the component hierarchy of the interface is detected automatically, and agents are defined in terms of user interface components and events.

Once this has been accomplished, the component hierarchy of the interface is detected automatically, and agents are defined in terms of user interface components and events. EDEM is implemented on top of an industry standard model for components [26] that standardizes how arbitrary software components make *events*, *properties*, and *methods* available to one another. Agent triggers are specified in terms of patterns of component events; agent guards are specified in terms of predicates involving component properties; agent actions may involve invocation of component methods.

5.3 Filtering and Abstraction

While separating probes from application code is important in allowing monitoring code and applications to evolve independently, we do not enforce a separation between the collection of data — typically preformed by probes — and filtering and abstraction — typically performed by usability analysts after data have been collected. This is because in order to do Internet-scale collection, data needs to be filtered close to the source to avoid undue network traffic. This does not affect application deployment because our architecture allows event processing to be modified incrementally as new data needs arise without impacting application code, as described above (Figure 5).

Filtering is accomplished by allowing event abstraction to occur within probes. 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. Agents themselves conform to the component standard described above and can therefore monitor one another in the same way they monitor user interface components. 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. Other agents can then detect patterns of these higher level agent events just as they can detect patterns of lower level user interface component events. This allows a multi-level event model to be constructed in which higher level, abstract events are specified in terms of combinations of lower level events. Only a selected subset of these events is ultimately reported via email upon application completion. A multi-level event model for usability data collection has been implemented using this approach and is described in [13].

The main contributions of these aspects of our approach include the following. First, by pushing event abstraction mechanisms into probes and closer to the source, event data can be compiled before being sent across the network. Second, by allowing higher level events to be specified in terms of lower level events, data can be collected and analyzed at multiple levels of abstraction.[3]

6 EVALUATION

It is important to evaluate to what extent the data collected by agents is subsequently useful in design improvements. It is also important to verify that the benefits of collecting usability data outweigh the costs of authoring and maintaining agents. To date our approach has been applied as part of a research demonstration project conducted by Lockheed Martin C2 Integration Systems in the context of a large-scale logistics and transportation information system based on the Global Transportation Network (GTN).[4] Please refer to the "Usage Scenario" section for a description.

Our initial experience with the Lockheed demonstration project 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 personal 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.

7 CHALLENGES

We are also addressing a number of other challenges that must be overcome before the potential of Internet-scale usability data collection can be realized. These challenges range from technical to social, including: agent representation, authoring, and maintenance; data storage and analysis; integration of expectations into the development process; privacy; and finally, non-disruptive techniques for requesting user feedback to augment automatically collected data.

1.This is not necessary on platforms where user interface components and events can be observed as well as *queried* from a separate process connected to the windowing system. Most windowing systems do not support this functionality, however.

2.A non-robust mapping can be generated automatically. Requiring the developer to provide aliases for components is the most robust and maintainable way to accomplish this mapping. The details as to why this is the case are beyond the scope of this paper.

3.Related work in distributed system monitoring and debugging is discussed below.

4.The GTN is a system that gathers, integrates, and distributes transportation-related information and acts as the central clearinghouse of transportation information for the Department of Defense. The system will eventually become the U.S. Transportation Command's primary command and control system and a fully integrated component of the Department of Defense's command and control infrastructure.

With respect to agent representation and authoring, we are investigating existing tools and techniques for constructing state-based [30], rule-based [10], and mode-transition-based [1] specifications. With regard to agent maintenance, we have identified mitigating factors that minimize the impact of maintenance issues [13]. With regard to data storage and analysis, we are investigating existing techniques for managing and processing temporal and sequential data [8][9].

With regard to integrating expectations into the development process, we are investigating relationships between expectations and usability requirements, cognitive walkthroughs, use cases, and other artifacts that already exist in the development process. With regard to privacy, users should always be notified prior to use that monitoring will take place. Since we do not collect arbitrary low-level data for unspecified purposes, but rather, higher level information for specified purposes, it is easier to justify collection, and users can be given discretionary control over what is reported. For example, upon exiting, users may be given the option to review a description of the data that has been collected, an explanation of the purposes for collection, as well as the collected data itself before allowing data to be reported. Users may also be given an option to deactivate monitoring altogether if privacy or security concerns are significant. In beta test situations, however, consent to allow data collection may be included as one of the terms of the license agreement. Finally, with regard to non-disruptive collection of user feedback, we have investigated various scheduling and control mechanisms to limit agent execution and filter agent requests for user attention [21].

It should be noted that developers cannot anticipate all areas where usability may break down, thus automatic detection of expectation violations is only part of a complete usability engineering solution. Our system has been designed so that users can determine for themselves when undetected breakdowns have occurred, and use the same reporting mechanisms to send information back to developers including program state, event history, as well as comments. Nonetheless, this approach is intended to be used in conjunction with existing usability engineering and evaluation techniques. It is not intended as a replacement.

8 RELATED WORK

8.1 Application Usage Monitoring

As described above, current approaches to application usage monitoring do not address issues of large-scale use. Monitoring code is typically intermingled with application code and too much low-level information is collected. The strengths of current approaches involve techniques for synchronizing event data with video data and observers' notes [15], and techniques for analyzing data once they have been collected [9][11][15]. While EDEM is primarily intended for use in situations where video equipment and human observers are not present, integration with existing video synchronization techniques and post-facto analysis tools is planned as future work.

Some experimenters have already begun to explore remote usability evaluation using the Internet [12]. However, data filtering and reporting 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 typically unaware of when their actions violate developers' expectations. EDEM's automatic mismatch detection is thus extremely important in collecting data under general circumstances.

8.2 Software Process Event Monitoring

Numerous researchers have investigated techniques for capturing software process event data for the purpose of: analyzing and improving the software process [29], validating the process with respect to a formal model [7], generating a formal model based on process events [7], or applying metrics to help guide the process (e.g., to automatically apply analysis tools when changes to code increase the likelihood of interface or control faults based on software metrics and historical data) [23].

While differing substantially in intent, EDEM bears some similarity to systems such as Amadeus [23] and YEAST [17] that detect process events and take pre-specified actions in response. However, many critical process events are difficult to detect automatically, including communication, coordination, and decision making events [29]. As a result, process event data is somewhat less amenable to automatic collection than is user interaction data. EDEM could, however, be used as a tool for collecting process-related events in so far as those events can be specified in terms of user interaction events occurring within software tools supporting the process in question.

Future work may involve the use of EDEM to do pattern discovery in addition to pattern validation [7]. This involves generating models to characterize patterns in event data as opposed to simply detecting when particular patterns have been satisfied or violated. This, however, would require either more network band-width and server disk-space for data transmission and storage, or alternatively, more sophisticated processing within the agents (i.e. probes) themselves. In our prototype implementation, we have attempted to be sensitive to utilization of network band-width, server disk-space, as well as the use of client processing resources. However, if network band-width and server disk-space are not serious issues in a given experimental situation, then pattern discovery may be performed on servers with the help of separate analysis tools once data have been collected.

8.3 Distributed System Monitoring and Debugging

Work in the area of distributed system debugging has also led to approaches with characteristics similar to those found in EDEM. Event-based behavioral abstraction (EBBA) is an approach to distributed system debugging in which models of expected program behaviors are created and compared to actual behaviors exhibited by the program [4]. TAOS is a specification-based testing system that applies a similar approach [20]. EDEM can be viewed as a "debugging" or "testing" tool for user interface designs that compares models of expected use to actual use. However, because these debugging and testing tools are primarily designed for use in development situations as opposed to ongoing use on client machines after deployment, they are significantly heavier-weight than EDEM in terms of memory, storage, and processing requirements.

Work in the area of distributed system monitoring has also addressed some of the issues addressed by EDEM. Our approach is similar to the Generalized Event Monitoring (GEM) approach presented in [18] in that it distributes event filtering and abstraction mechanisms as close as possible to the sources of events, as opposed to performing filtering and abstraction after distribution of event data.

8.4 Event Frameworks

While differing in focus, our monitoring framework is related to other frameworks that have been proposed for event-based software integration [3] and internet-scale event observation and notification [22].

Barret et al. [3] provides an excellent semi-formal, object-oriented framework for characterizing event-based tool integration (EBI). The most notable difference between the EBI framework and our model is that we explicitly differentiate between the system of interest and probes. In the EBI model, the tools being integrated are analogous to the system being monitored in our model, and wrappers are analogous to our probes. However, tools and wrappers are treated as a single logical entity in their model ("Participants"), and wrappers (i.e. probes in our model) are thus not seriously considered as potential loci of flexible and dynamically reconfigurable event processing and distribution activities ("Message Transform Functions" and "Delivery Constraints" in their terminology).

Rosenblum and Wolf [22] provides a good overview of several interrelated design dimensions that must be considered in designing any Internet-scale event observation and notification facility. Their framework includes an object model, an event model, a naming model, an observation model, a time model, and a resource model. In terms of these dimensions, our framework primarily focuses on issues involved in the observation and resource models. However, in terms of the roles and activities introduced in our framework, the Rosenblum/Wolf framework focuses primarily on issues associated with event notification and processing in the realm of distributors and consumers, without addressing in detail how these roles and activities interrelate with the remaining roles (namely probes) and activities (namely observation and actions) identified in our framework.

9 CONCLUSIONS

The main contributions of this paper are an approach (based on usage expectations) and an architecture (based on agents that perform distributed event filtering and abstraction) that together make large-scale collection of usability data on the Internet a practical possibility. By treating usage expectations explicitly in the development process, we provide a principled way of focusing data collection. By separating probes from application code, we provide an architecture that allows event monitoring to evolve flexibly and independently of applications being monitored. Finally, by embedding event abstraction mechanisms within our probes, we allow events to be filtered in a scalable way, reducing network band-width requirements, and allowing testing to address events at multiple levels of abstraction.

ACKNOWLEDGMENTS

The authors would like to thank J. Robbins and P. Oreizy for sharing their insights as well as the members of the Lockheed Martin C2 Integration Systems team, Teri Payton, Lyn Uzzle, and Martin Hile, for their patience, cooperation, and thoughtful feedback. Thanks are also due to A. Girgensohn, F. Shipman, A. Lee, and A. Turner who worked on precursors to this work and continue to provide insight and support. Finally, the authors acknowledge R. Balzer and the other participants in an informal SIG meeting on monitoring held during the Second Annual Evolutionary Design of Complex Systems (EDCS) Program Workshop and Demo Days, July, 1997, for providing inspiration for some of the ideas developed in the monitoring framework presented here.

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

REFERENCES

1. J.M. Atlee and J. Gannon. "State-based Model checking of event-driven system requirements". IEEE Transactions on Software Engineering, Jan. 1993.
2. R.M. Baecker, J. Grudin, W.A.S. Buxton, and S. Greenberg, eds. Readings in Human-Computer Interaction: Toward the Year 2000. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1995.
3. D.J. Barrett, L.A. Clarke, P.L. Tarr, and A.E. Wise. "A Framework for Event-Based Software Integration". ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 5, No. 4, Oct. 1996.
4. P.C. Bates. "Debugging heterogeneous distributed systems using event-based models of behavior". ACM Transactions on Computer Systems, Vol. 13, No. 1, Feb. 1995.
5. N. Bevan and M. Macleod. "Usability Measurement in Context". In Jacob Nielsen ed.: Usability Laboratories, Special Issue of Behaviour and Information Technology, Vol. 13, No. 1 & 2, Apr. 1994.
6. J. Chen, Providing Intrinsic Support for User Interface Monitoring. In proceedings of Human-Computer Interaction - INTERACT'90.

7. J.E. Cook. “Process Discovery and Validation through Event-Data Analysis”. Ph.D. Thesis, Technical Report CU-CS-817-96, University of Colorado, Sep. 1996.
8. S. Fickas and M. Feather. “Requirements Monitoring in Dynamic Environments”. In Proceedings of the Second IEEE International Symposium on Requirements Engineering, York, England, Computer Society Press, Mar. 1995.
9. C. Fisher and P. Sanderson. “Exploratory sequential data analysis: exploring continuous observational data”. Interactions, Vol.3, No. 2, Mar. 1996.
10. 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. Monterey, CA, USA, 1994.
11. M.L. Hammontree, J.J. Hendrickson & B.W. Hensley. “Integrated Data Capture and Analysis Tools for Research and Testing on Graphical User Interfaces”. In Proceedings of CHI’92, Monterey, CA, USA, May 1992.
12. 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, ACM Press, 1996.
13. 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.
14. D.M. Hilbert and D.F. Redmiles. “Agents for Collecting Application Usage Data Over the Internet.” To appear in Proceedings of the Second International Conference on Autonomous Agents (Agents’98), Minneapolis/St. Paul, MN, USA, May 1998.
15. D.E. Hoiem and K.D. Sullivan. “Designing and Using Integrated Data Collection and Analysis Tools: Challenges and Considerations”. In Jacob Nielsen ed.: Usability Laboratories, Special Issue of Behaviour & Information Technology, Vol. 13, No. 1 & 2, Apr. 1994.
16. J. Kay and R.C. Thomas. “Studying Long-Term System Use”. Communications of the ACM, Vol. 38 No. 7, Jul. 1995.
17. B. Krishnamurthy and D.S. Rosenblum. “Yeast: A General Purpose Event-Action System”. IEEE Transactions on Software Engineering, Vol. 21, No. 10, Oct. 1995.
18. M. Mansouri-Samani and M. Sloman. “An Event Service for Open Distributed Systems”. In Proceedings of the Joint International Conference on Open Distributed Processing (ICODP) and Distributed Platforms (ICDP), Toronto, Canada, May 1997.
19. J. Nielsen. Usability Engineering. Academic Press, AP Professional, Cambridge, MA, USA, 1993.
20. D. J. Richardson. “TAOS: Testing with Analysis and Oracle Support”. In Proceedings of the 1994 International Symposium on Software Testing and Analysis, Aug. 1994.
21. J.E. Robbins, D.M. Hilbert, and D.F. Redmiles. “Using Critics to Analyze Evolving Architectures”. In Proceedings of the Second International Software Architecture Workshop. San Francisco, CA, USA, Oct. 1996.
22. D.S. Rosenblum and A.L. Wolf. “A Design Framework for Internet-Scale Event Observation and Notification”. In Proceedings of the Sixth European Software Engineering Conference/ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering, Zurich, Switzerland, Sep. 1997.
23. R.W. Selby, A.A. Porter, D.C. Schmidt, and J. Berney. “Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development”. In Proceedings of the Thirteenth International Conference on Software Engineering, 1991.
24. A.C. Siochi, R.W. Ehrich. “Computer Analysis of User Interfaces Based on Repetition in Transcripts of User Sessions”, ACM Transactions on Information Systems. Vol. 9, No. 4, Oct. 1991.
25. A.C. Siochi & D. Hix. “A Study of Computer-Supported User Interface Evaluation Using Maximal Repeating Pattern Analysis”. In Proceedings of CHI’91, New Orleans, LA, USA, ACM Press, Apr.-May 1991.
26. Sun Microsystems. “JavaBeans^{TM} API Specification, Version 1.01”. Jul. 1997. (URL: http://java.sun.com/beans/).
27. R.N. Taylor and J. Coutaz. “Workshop on Software Engineering and Human-Computer Interaction: Joint Research Issues”. In Proceedings of the International Conference on Software Engineering ‘94, Sorrento, Italy, May 1994.
28. P. Weiler. “Software for the Usability Lab: A Sampling of Current Tools”. In Proceedings of INTERCHI’93, Amsterdam, The Netherlands, ACM Press, Apr. 1993.
29. A.L. Wolf and D.S. Rosenblum. “A Study in Software Process Data Capture and Analysis”. In Proceedings of the Second International Conference on Software Process, 1993.
30. J. Wing. “A Specifier’s Introduction to Formal Methods”. IEEE Computer, Sep. 1990.