

Agents for Collecting 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 use is critical in software engineering. Prototyping, beta testing, and usability testing are widely employed to refine system requirements, to 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 agent-based approach and prototype system that makes large-scale collection of usage data over the Internet a practical possibility.

Keywords

Application usage monitoring, Internet-scale usability data collection, remote usability testing, expectation-driven event monitoring, expectation agents

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 be used as 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 number 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 for 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 gathered manually by beta testers themselves. Since data are collected manually, and because beta testers pay the price of bug reporting while vendors receive most of the benefit, both the quality and quantity of data is limited. Fragmentary reporting leads to difficulty in reproducing

and analyzing problems, and typically only the most obvious or unrecoverable errors 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 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, based on application usage monitoring, that makes ongoing, large-scale use a practical possibility. The specific contributions of our approach include: (a) explicit treatment of “usage expectations” in the development process to improve design and focus data collection, (b) a flexible agent-based approach to monitoring that allows instrumentation to evolve dynamically as monitoring needs change, without requiring modifications to application code, and (c) flexible event processing embedded within instrumentation to provide distributed filtering and multiple levels of abstraction in collected data.

In the following section, we describe the state of the practice in application usage monitoring and explain why current techniques cannot be used on a large scale over the Internet. Next we describe a scalable agent-based approach in which agents collect data on behalf of developers over the Internet. Finally, we discuss the current status and evaluation of a prototype implementation followed by related work and conclusions.

2 CURRENT 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][20], usage monitoring involves instrumented applications or windowing systems that 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 video and 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.

Readers interested in further details regarding data storage and analysis may wish to consult other papers regarding existing techniques [6][8][11][13][27][28]. This paper focuses specifically on data collection, since data collection impacts choice of subjects (e.g. laboratory subjects vs. actual users), study setting (e.g. usability labs with specially configured workstations vs. normal working conditions), and study duration (e.g. short experiments vs. ongoing evaluation). Unfortunately, existing approaches are not intended for ongoing, large-scale use with actual users under normal working conditions.

One problem is that most current approaches do not appropriately separate instrumentation from application code. As a result, it is difficult to evolve instrumentation as monitoring needs change without affecting the application being monitored. In order to modify the type, format, or amount of data that is captured, the application typically needs to be modified and re-delivered. An obvious strategy to deal with this problem has been to collect information directly from the windowing system. This, however, results in a deluge of low-level event data that must be filtered before any meaningful analysis can be performed.

To avoid modifying instrumentation that is intermingled with code, or perhaps as a result of inserting probes directly into the windowing system, experimenters are often left with little choice but to collect as much data as possible — at very low levels of abstraction — and to defer all processing and analysis until after data have been collected. This presents serious problems for Internet-scale use. The volume of interaction 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 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 [31].

3 EXPECTATION-DRIVEN EVENT MONITORING

We propose an approach to large-scale application usage monitoring based on the notion of “usage expectations”. In the following subsections, we discuss the importance of usage expectations in the development process, provide an overview of our approach, and describe a simple usage scenario.

3.1 Expectations in the Development Process

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* [7]. When the environment in which a system is deployed or its users behave in unexpected ways, various problems can ensue. Such problems typically result in sub-optimal user and system performance, and can, in safety- or security-critical systems, lead to much more serious 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 or in use cases. 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.

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 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 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://”.

3.2 The EDEM Approach

Expectation-driven event monitoring (EDEM) is an agent-based approach to application usage monitoring, 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/or usability experts): (1) identify usability 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 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

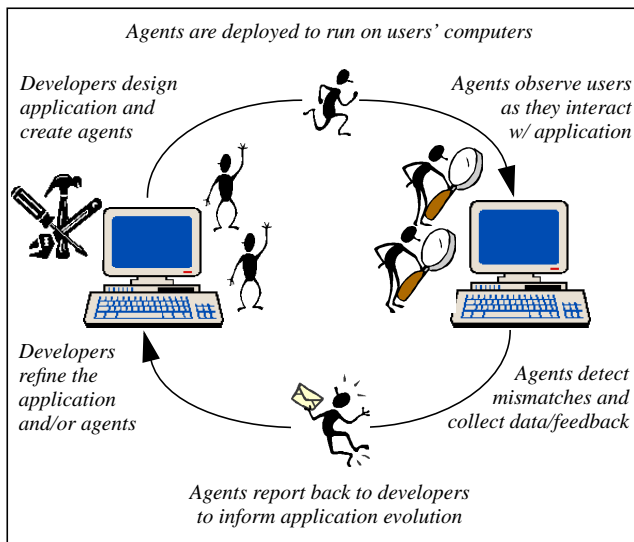


Figure 1. A software development process augmented with agents for collecting usability data.

including notifying users and/or developers of mismatches, reporting system state and/or event history to developers for debugging purposes, providing guidance or suggestions to users, or collecting feedback directly from users [10].

3.3 Usage Scenario

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 helps developers collect valuable usage data, consider the following usage scenario. A group of developers are designing a form-based application to help customer representatives at a hypothetical phone company take customer orders over the phone. See [7] for a more detailed scenario. The user interface of the order form must be well-suited to users' tasks and users must be aware of how to most efficiently use the interface in order to maximize the number of orders that can be taken in a given amount of 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 order form application is ready for deployment. Figure 2 shows the customer information section of the order form. Developers are interested in verifying that their expectations, particularly those relating to efficiency, are actually met by users of the prototype. In particular, they are interested in verifying that users complete input fields in the order expected, and that the ZIP field in the customer information section is used to automatically complete the City and State fields, as expected.

Figure 2. Customer information section of a hypothetical phone service order form.

Figure 3. A simple agent editor.

Figure 3 shows an agent editor that allows developers to specify expectation agents without writing code. In Figure 3 (upper-left) the developer expresses interest in detecting when the user begins editing the State field in the order form and adds this event to an agent (lower-right) that will “fire” whenever the user edits the City or State fields while the ZIP field is empty. This agent then runs on users' computers monitoring user interactions and reporting data back to developers when expectations are violated by actual usage. Collected data is stored in a database and standard plotting and analysis tools are used to analyze results.

The main contribution of our approach, as it has thus far been described, is our 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. 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. In the following section we describe how our approach allows monitoring to evolve without affecting the deployment of applications being monitored, and how agents provide distributed event filtering and abstraction.

4 IMPLEMENTATION

4.1 Expectation Agents

Expectation agents are currently represented as instances of a simple Java™ class with attributes describing 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 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.

Agent triggers are specified in terms of event patterns of the following form:

- “A or B or ...” (Disjunction)
- “A and B and ...” (Conjunction)
- “A then B then ...” (Sequence)
- “(A and B) with no intervening C” (Conjunction with Exclusion)
- “(A then B) with no intervening C” (Sequence with Exclusion)

Where variables A, B, and C are filled in by specifying:

- a component from the user interface plus an event (e.g. `LOST_EDIT:City(Field)` which occurs when the `City(Field)` component is edited and then another component is edited), or
- another agent and agent event (e.g. `FIREED:AddressCompleted` which occurs when the `AddressCompleted` agent has fired)

Agent guards are specified in terms of condition patterns of the following form:

- “A or B or ...” (Disjunction)
- “A and B and ...” (Conjunction)

Where variables A and B are filled in by specifying:

- a component from the user interface and some expression involving its properties (e.g. `value = `':Zip(Field)` or `value > 10:NumPhones(Field)`), or
- another agent and some expression involving its properties (e.g. `enabled = false:AddressStarted` or `count > 100:ZipBeforeCityState`)

An optional time limit may also be specified to require that the agent’s event pattern be satisfied within a specified interval.

```
edem.kernel.Agent[
  name="Enter ZIP field first",
  eventPattern="A or B or ...",
  events=Vector[2,
    edem.kernel.EventRecord[
      name="City(Field)", type="component", event="GOT_EDIT"],
    edem.kernel.EventRecord[
      name="State(Field)", type="component", event="GOT_EDIT"]],
  conditionPattern="A or B or ...",
  conditions=Vector[1,
    edem.kernel.ConditionRecord[
      name="Zip(Field)", type="component", condition=edem.kernel.Condition[
        predicate="=",
        key="value",
        value="",
        negate="false"]]],
  timeLimit="",
  action=edem.kernel.Action[
    message="Enter ZIP before City/State. City/State can be completed automatically.",
    interruptUser="false",
    feedback="true",
    log="true",
    logTime="false",
    logValues="false",
    summary="true",
    summaryCount="true"],
  repeat="true",
  enabled="true"]
```

Figure 4. ASCII version of the agent specified in Figure 3.

This simplified trigger and guard notation facilitates a form-based authoring environment. Arbitrary patterns can be specified by composing agents (as explained below). Figure 4 shows an ASCII representation of the agent specified in Figure 3.

4.2 Monitoring Architecture

In our prototype Java 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 library 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 agents have been defined, they are serialized and stored in ASCII format in a file that is associated with a

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, however, the details as to why this is so are beyond the scope of this paper.

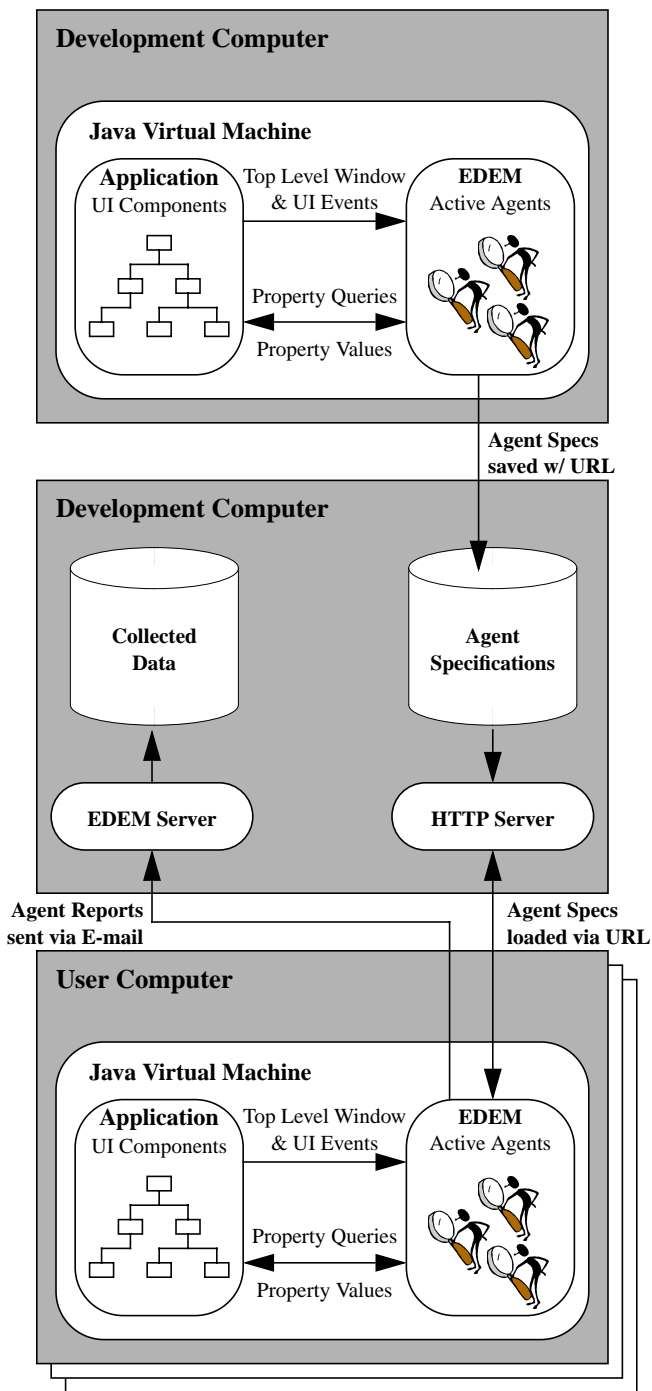


Figure 5. The EDEM architecture.

URL on a development computer. 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 E-mail protocol is used to send agent reports back to development computers. An EDEM server is used to store agent data reports for later analysis. Agents may therefore be modified, added, and deleted incrementally without requiring modifications to the application being monitored. Figure 5 shows a high-

level view of the EDEM architecture.

In sum, expectation agents act as reconfigurable instrumentation that can be incrementally modified to collect data about application usage as needed. This architecture provides a general solution for allowing instrumentation to evolve flexibly in a large-scale, distributed system, without requiring the systems being monitored to be modified when monitoring needs change.

4.3 Event Filtering and Abstraction

While separating instrumentation from application code is important in allowing instrumentation to evolve without impacting application deployment, we do not enforce a separation between the collection of data (typically preformed by instrumentation) and filtering and abstraction of the data (typically performed manually after data have been collected). This is because Internet-scale 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 application code, as described above.

Filtering is accomplished by allowing event abstraction to occur within agents. 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 are implemented on top of an industry standard component model, the JavaBeans™ specification [30], 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.

Because agents themselves, like the components they monitor, conform to the JavaBeans specification, they too can be monitored in the same way that user interface components can be monitored. 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). Other agents can then detect patterns of these higher level agent events. 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 [10].

Agent output is logged during execution and sent back to development computers via E-mail when the application of interest is exited. The main contributions of this aspect of our approach include the following. First, by pushing event abstraction into "instrumentation" and closer to the source, event data can be filtered and abstracted 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.

5 EVALUATION

It is important to evaluate to what extent the data collected by expectation 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 within the context of a research demonstration project conducted by Lockheed Martin C2 Integration Systems for the Global Transportation Network (GTN) project¹.

Our initial experience with GTN suggests that the time, effort, and expertise required to integrate with EDEM and to author agents is not extensive, and that significant data can nonetheless be captured. Also, because our approach allows agents to be deployed dynamically, 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. Further investigation within the context of GTN and evaluations with more quantifiable results are planned for the future.

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 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.

With respect to agent maintenance, we have already identified mitigating factors that minimize the impact of maintenance concerns [10]. With regard to data storage and analysis, we are investigating existing techniques for managing and processing temporal and sequential data [5][6]. 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, 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. 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 [23].

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

1. 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.

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.

6 RELATED WORK

6.1 Application Usage Monitoring

As mentioned above, current approaches to application usage monitoring do not address issues of ongoing, large-scale use. *Observation* is achieved by inserting programmatic probes directly into application code or by tapping into the windowing system's event queue. *Distribution* is achieved by writing collected data directly to a file or other stream for later processing. *Filtering and abstraction* is typically performed manually by usability analysts after distribution and before *analysis*. In expectation-driven event monitoring, observation, filtering, and abstraction all take place within expectation agents. Expectation agents act as dynamically reconfigurable probes that perform filtering and abstraction prior to distribution, thereby reducing network bandwidth requirements, and allowing data to be collected at multiple levels of abstraction. Agents may be updated dynamically as monitoring needs change. Figure 6 illustrates this contrast.

The strengths of current approaches involve techniques for synchronizing event data with video data and observers' notes [11][32], and post-facto analysis [6][8][11][32]. While EDEM is primarily intended for use in situations where video equipment and human observers are not present, integration with existing video synchronization techniques as well as post-facto analysis tools is planned as

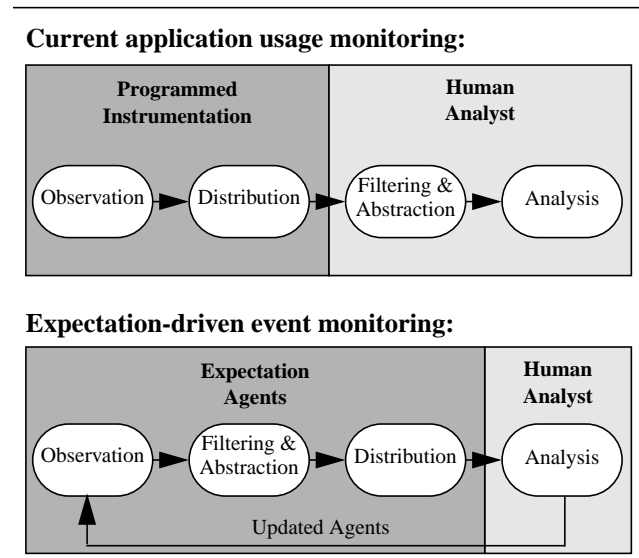


Figure 6. EDEM contrasted with current application usage monitoring.

future work.

Some experimenters have begun to explore remote usability evaluation using the Internet [9]. 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. Expectation agents are thus indispensable in detecting mismatches without having to depend on proactive users.

6.2 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 [35], validating the process with respect to a formal model [4], generating a formal model based on process events [4], or applying metrics to help guide the process (e.g., to automatically apply analysis tools when changes to code increase the likelihood of faults based on software metrics and historical data) [25].

While differing substantially in intent, EDEM bears some similarity to systems such as Amadeus [25] and YEAST [15] 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 [35]. 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 detecting process-related events in so far as those events are expressible in terms of user interactions 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 [4]. This involves generating models to characterize unanticipated patterns in event data as opposed to simply detecting when particular patterns have been satisfied or violated. This, however, will require either more network band-width and server disk-space for data transmission and storage, or alternatively, more sophisticated processing within expectation agents 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.

6.3 Distributed System Debugging and Monitoring

Work in the area of distributed system debugging has 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 [3]. TAOS is a specification-based testing system that applies a similar approach [22]. EDEM can be viewed as a “debugging” or “testing” tool for user interfaces 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 users’ computers after deployment, they require significantly more memory, storage, and processing resources than EDEM.

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 [19] in that it distributes event filtering and abstraction mechanisms as close as possible to event sources, as opposed to performing filtering and abstraction after distribution of event data.

6.4 Agents

In contrast to interface agents that act primarily as “user assistants” [17], expectation agents act primarily as “developer assistants” by monitoring application use, detecting when environmental conditions and/or user behavior violate developers’ expectations, and reporting data back to developers for evaluation purposes.

6.4.1 Agent Representation

There are numerous techniques that might be used to represent expectation agents. State-based representations are well suited for expressing expectations about sequences of events 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 occurring in the user interface. 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 programming [34].

Rule-based representations [7] 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 [1].

We have chosen what is basically a mode-transition-based representation, in which agents are specified in terms of triggers, guards, and actions. A similar approach was used in early work on agents by Malone and colleagues [18]. In their approach, agents are represented using a trigger, query, and action. Queries are roughly equivalent to guards in our approach.

Because we are interested in defining agents to act upon specific patterns of user interface events, we use a symbolic representation as opposed to a statistical representation such as used by Sheth and Maes [26]. Statistical techniques and other data mining approaches may eventually be used to do post-hoc analysis of already collected data for the purpose of discovering patterns as described above.

6.4.2 Agent Mobility

Expectation agents must move from development computers to user computers prior to execution. Stamos and Gifford [29] introduced the concept of remote execution in which servers are viewed as programmable processors. EDEM can be thought of as a programmable processor in which expectation agents are the programs. Agents are authored and maintained on development computers and transported to user computers to execute remotely and report data back to developers. Several researchers have begun to look at sophisticated techniques for achieving agent mobility [12][14][24]. Rus and colleagues [24] describe an approach in which agents sense network status and navigate adaptively based on reactive plans. There are also a number of commercially (and freely) available mobile agent platforms [21], of which Telescript [33], introduced by General Magic, Inc. in 1994, was perhaps the first.

Because expectation agents are published in a well-known location and only transported once per execution, their requirements for mobility are fairly straightforward. As a result, we have avoided dependencies on special-purpose mobile agents platforms and opted instead for a more standard and ubiquitous transport mechanism. Expectation agents are associated with URL's on development computers and downloaded to user computers via standard hypertext transfer protocol (HTTP).

6.4.3 Agent Interface

A large portion of the research community in agents works on the issue of interfacing software agents to their human users. Great emphasis is given to anthropomorphism [16][17]. We believe this approach is critical for many classes of users and can greatly improve the chances of agents being adopted. However, in our current research, we have developed a more literal, low level, and "precise" interface between the agents and their primary users. We believe this approach to be appropriate because of the sophistication of the users, namely, software developers, and because of the need for precision in defining expectations. Anthropomorphism may eventually be incorporated as a mechanism for interacting with end users of the applications being monitored, to request feedback

from them, or to provide suggestions based on developers expectations [10].

7 CONCLUSIONS

The main contributions of this paper include an expectation-driven approach to 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 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 instrumentation within expectation agents, we allow monitoring to evolve flexibly without impacting the deployment of applications being monitored. Finally, by embedding event abstraction mechanisms within expectation agents, we allow events to be filtered in a scalable way, reducing network bandwidth requirements, and allowing data collection to address events at multiple levels of abstraction.

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 who continue to provide insight and support.

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.

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, S. Greenberg, eds. *Readings in Human-Computer Interaction: Toward the Year 2000*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1995.
3. P.C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, Vol. 13, No. 1, Feb. 1995.
4. 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.
5. 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.
6. C. Fisher and P. Sanderson. Exploratory sequential data

- analysis: exploring continuous observational data. *Interactions*, Vol.3, No. 2, ACM Press, Mar. 1996.
7. 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 '94*. Monterey, CA, USA, 1994.
 8. 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*, ACM Press, Monterey, CA, USA, May 1992.
 9. H.R. Hartson, J.C. Castillo, J. Kelso, W.C. Neale. Remote Evaluation: The Network as an Extension of the Usability Laboratory. In *Proceedings of CHI'96*, ACM Press, 1996.
 10. 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.
 11. D.E. Hoiem, 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.
 12. D. Johansen, R. van Renesse, and F. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, 1995.
 13. J. Kay and R.C. Thomas. Studying Long-Term System Use. *Communications of the ACM*, Vol. 38 No. 7, Jul. 1995.
 14. K. Kotay and D. Kotz. Transportable Agents. In *Proceedings of the Workshop on Intelligent Information Agents, 1994*.
 15. B. Krishnamurthy and D.S. Rosenblum. Yeast: A General Purpose Event-Action System. *IEEE Transactions on Software Engineering*, Vol. 21, No. 10, Oct. 1995.
 16. B. Laurel. Interface Agents: Metaphors with Character. In *The Art of Human-Computer Interface Design*, Addison-Wesley Publishing Company, Reading, MA, 1990, pp. 355-365.
 17. P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, vol.37, (no.7), July 1994, pp.30-40, 146.
 18. T.W. Malone, K.Y. Lai, and C. Fry. Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '92)*. (Toronto, Canada) ACM, New York, Oct. 31-Nov. 4, 1992, pp. 289-297.
 19. 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.
 20. J. Nielsen. *Usability Engineering*. Academic Press, AP Professional, Cambridge, MA, USA, 1993.
 21. ObjectSpace, Inc. ObjectSpace Voyager and Agent Platforms Comparison. ObjectSpace white paper, 1997.
 22. D.J. Richardson. TAOS: Testing with Analysis and Oracle Support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, Aug. 1994.
 23. J.E. Robbins, D.M. Hilbert, and D.F. Redmiles. Extending Design Environments to Software Architecture Design. To appear in *The International Journal of Automated Software Engineering. Special Issue: The Best of KBSE'96*.
 24. D. Rus, R. Gray, and D. Kotz. Transportable Agents. In *Proceedings of Autonomous Agents 1997* (Marina Del Rey, California, USA), 1997.
 25. 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.
 26. B. Sheth and P. Maes. Evolving agents for personalized information filtering. In *Proceedings of the Ninth Conference on Artificial Intelligence for Applications* (Orlando, FL) IEEE Computer Society Press, Los Alamitos, CA, March 1-5, 1993. p.345-52.
 27. A.C. Siochi and 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.
 28. A.C. Siochi and 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.
 29. J. Stamos and D. Gifford. Remote Execution. In *ACM Transactions on Programming Languages and Systems*, 12(4):537-565, October 1990.
 30. Sun Microsystems. JavaBeans™ API Specification, Version 1.01. Jul. 1997. (URL: <http://java.sun.com/beans/>).
 31. R.M. 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.
 32. P. Weiler. Software for the Usability Lab: A Sampling of Current Tools. In *Proceedings of INTERCHI'93*, Amsterdam, The Netherlands, ACM Press, Apr. 1993.
 33. J.E. White. Telescript Technology: the Foundation for the Electronic Marketplace. General Magic white paper, General Magic, Inc., 1994.
 34. J. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, Sep. 1990.
 35. 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.