*Behavior Research Methods*
*2008, 40 (2), 413-421*
*doi: 10.3758/BRM.40.2.413*

# AppMonitor: A tool for recording user actions in unmodified Windows applications

Jason Alexander, Andy Cockburn, and Richard Lobb
*University of Canterbury, Christchurch, New Zealand*

This article describes AppMonitor, a Microsoft Windows–based client-side logging tool that records user actions in unmodified Windows applications. AppMonitor allows researchers to gain insights into many facets of interface interaction such as command use frequency, behavioral patterns prior to or following command use, and methods of navigating through systems and data sets. AppMonitor uses the Windows SDK libraries to monitor both low-level interactions, such as "left mouse button pressed" and "Ctrl-F pressed," as well as high-level "logical" actions, such as menu selections and scrollbar manipulations. The events recorded are configurable, allowing researchers to perform broad or targeted studies. No user input is required to manage logging, allowing participants to seamlessly conduct everyday work while their actions are monitored. The system currently supports logging in Microsoft Word and Adobe Reader; however, it could be extended for use with any Microsoft Windows–based application. To support other researchers wishing to create multilevel event loggers, we describe AppMonitor's underlying architecture and implementation, and provide a brief example of the data generated during our 4-month trial with six users.

Interface designers and human–computer interaction (HCI) researchers need to understand how users interact with systems in order to improve them. Several strategies are commonly used to gain such insights, including controlled experiments run in research labs, field studies, and contextual inquiry (Holtzblatt & Jones, 1993). Client-side logging of user actions is another technique that has several advantages over other methods: It is highly scalable, since software is easily disseminated over the Web; it is low-cost to the participants and experimenters, because, once it has been installed, the logs are generated and stored automatically; it readily allows longitudinal analysis of weeks, months, or years of use; and it allows fine-grain analysis of subsecond events and low-level actions. The two primary disadvantages of client-side logging are, first, that it cannot record the user's high-level goals or context; and second, that developing logging software can be prohibitively complex.

The difficulty in implementing logging software has discouraged and impaired its use. For example, in analyzing Web navigation behavior, participants have been asked to abandon their preferred proprietary Web browsers in favor of customized logging versions of open-source software (e.g., Tauscher & Greenberg, 1997) or to use "roll your own" systems (e.g., Kellar, Hawkey, Inkpen, & Watters, in press). Clearly, logging analyses should measure interaction with the users' preferred software rather than with a surrogate that supports a subset of the real system features or that presents the features in a different manner.

In this article, we describe AppMonitor, a tool that allows logging of user interaction with unmodified applications running under Microsoft Windows. Once installed, AppMonitor requires no intervention from the user. It silently records all user actions of interest (from low-level mouse movements to the selection of items in combo/dialog boxes), and periodically uploads them to a remote Web server. AppMonitor's logs provide a terse but semantically rich description of interaction. As well as capturing a description of low-level user events, such as mouse movement and button- or keypresses, it can also record semantic aspects of interaction, discriminating between the different interface components used and their modes. This capability distinguishes AppMonitor's logs from the video logs generated by screen recorders (such as TechSmith's Camtasia Studio, www.techsmith.com), because video logs require human interpretation to determine the semantics connecting their key logs and mouse logs with the video.

To assist researchers in developing their own logging software, we explain how we implemented AppMonitor. The following section reviews related work on tools supporting event logging. We then describe AppMonitor's underlying architecture and implementation and give an example of the output it generates. Finally, we provide some preliminary analyses of the total output from our six beta testers, who used AppMonitor to record their interactions with Microsoft Word and Adobe Reader for a 4-month trial period.

## RELATED WORK

Direct human observation of interaction allows an immediate and contextually rich understanding of the inter-

**J. Alexander, jason@cosc.canterbury.ac.nz**


*Copyright 2008 Psychonomic Society, Inc.*

action between user and system. The substantial disadvantage of direct human observation is the demand it makes on the experimenter's time, making it scale poorly, both to large sample sizes and to longitudinal studies. Modern high-functionality interfaces typically provide many alternative approaches to completing the same task; understanding how a population of users interacts with a system demands a large sample. Similarly, when researchers wish to understand how system use changes over time (days, months, or even years of use), direct observation is impossible, due to cost. Logging software can overcome these issues by allowing large-scale analysis on interactive system use through statistical aggregation.

This section reviews approaches for automatically gathering data on user interaction. First, we describe work on screen-capture systems, which record pictorial representations of interaction with the interface in question. Second, we examine research on client-side logging tools, which have similar objectives to AppMonitor.

**Screen Recorder Systems**

Screen recorders such as Techsmith's Camtasia Studio allow researchers to record the visual state of the user's screen or window while the user's work is going on. The recordings are often enhanced with low-level logs of user events such as mouse movement and clicks, allowing researchers to visually emphasize these activities when they replay the recording.

Screen recorders are widely used for developing instructional material for training users. However, similar system capabilities are required for interface evaluation software. Techsmith's suite of software, for example, includes a range of screen capture tools, as well as the "Morae" application, which explicitly supports usability testing.

Although they are excellent tools for supporting detailed analysis of a single user's interaction with a system, screen recorders have three main limitations. First, they are resource-intensive, consuming a large proportion of the processing power of current computers, and generating very large video data files. Second, they do not gather data on logical user interface events. Instead, they record the screen state and the location of mouse movements and mouse clicks. Researchers wishing to measure data concerning the logic of interaction (e.g., how often the zoom functions are used) will have to replay the recorded video file to interpret the users' actions. Third, the technique scales poorly. This is a direct result of the high resource requirements (users are unlikely to be willing to suffer a long-term negative effect in system performance) and the failure to capture interface semantics (it is impractical for researchers to manually extract the semantics of interaction from many long-term logs).

**Client-Side Logging Systems**

Client-side logging systems overcome the three limitations of screen recorders. They are relatively light on system resources, because they need only capture those events that arise within the application under study. Rather than generating large video files of interaction, the logs consist of abstract descriptions (normally in raw text) of the logic of the user's action and the associated system state. As a consequence, the approach scales extremely well, with few theoretical restrictions on global dissemination over the Internet to millions of users for long-term analysis. If the logs generated by client-side logging systems are sufficiently rich, videos of the user's interaction with the system can be re-created from these log files. The primary problem with client-side logging, however, is the complexity of creating software that, in essence, "spies" on the internal state of proprietary software systems while they are running.

Several client-side event logging systems have been developed by researchers, but all have been simple keyboard and mouse loggers. These include Datalogger (Westerman et al., 1996) for Windows 3.1 and DOS; InputLogger (Trewin, 1998) for the Apple Macintosh; and RUI (Kukreja, Stevenson, & Ritter, 2006) for Windows and Mac OS X. All of these examples provide timing logs for keystrokes, mouse clicks, and mouse moves, but none provide information regarding the semantics of the application and the user's action, such as the name of the button that was pressed, the state of the scrollbar, the current interface view, and so on.

Two research projects with different goals have attempted to use client-side logging tools to study user interaction with Microsoft Word (Linton, Joy, Schaefer, & Charron, 2000; McGrenere, 2002). McGrenere describes MSTracker, an "internal tool used by the usability team at Microsoft" and unavailable to other researchers (pp. 174–175). Like AppMonitor, the MSTracker tool listens to the Microsoft Active Accessibility events that are generated by most Windows applications. Linton et al. describe the organization-wide learning (OWL) system. It is a macro-based logger that records high-level commands issued to Microsoft Word running on Apple Macintosh computers through the menus and toolbars. It cannot capture mouse or keyboard input or navigation actions such as scrolling.

Microsoft's Customer Experience Improvement Program[1] is based on client-side logging capabilities built into their Office suite of programs. When users agree to participate, logs of their actions within each application are uploaded to a Microsoft server. Although these data allow Microsoft to improve their products, the data are binary coded, and therefore unavailable to the user or researchers; accordingly, the events logged are not configurable. Very little information on this system—what it logs and the results of studies—is in the public domain.

Event logging facilities are also supported by macro recorders, which register a series of user actions and assign them to simple interface controls, such as buttonpresses or key commands. Commercial examples include "Macro Magic" (www.iolo.com/mm), "Workspace Macro" (www.tethyssolutions.com), "Smack" (www.cpringold.com), and "JitBit" (www.jitbit.com). Macro recorders generally register only low-level events, such as the coordinates of buttonpresses and key sequences, making the data they generate largely insufficient for interpreting the semantics of interface manipulation.

## DESCRIPTION OF APPMONITOR

AppMonitor is a Microsoft Windows–based program that records both high- and low-level events in unmodified Windows applications. The system currently supports logging in Microsoft Word and Adobe Reader, but little effort would be required to extend logging to other applications.

AppMonitor was designed to minimize the impact it has on users, both during installation and subsequent use. We have found no instances where firewall, antivirus, or spyware software has had to be modified to allow AppMonitor to function correctly. The logged applications do not have to be modified in any way, and after AppMonitor is installed, all aspects of its use are automated—users do not have to manually start and stop recording, upload, or copy log files. AppMonitor is launched at system startup and placed into the system tray. Logging begins automatically as the applications of interest are opened, and log files are uploaded by the system to a remote Web server at regular intervals.

The set of logged events within each application is fully configurable by the researchers. AppMonitor can record keystrokes, mouse events, logical events (such as menu selections), application-level events (such as window resizing), and document navigation events, such as opening and closing documents and changes in zoom level or scrollbar position.

Feedback from potential beta testers revealed the importance of allowing users to interactively view the data gathered. Users can view a real-time display of the recorded events by double-clicking on the system-tray icon. After we had added this feature, volunteers readily agreed that using AppMonitor raised few (or no) privacy concerns (see the Key Events section for a further discussion).

## SYSTEM ARCHITECTURE

AppMonitor is written in C/C++ using Visual Studio, with one dynamic link library (DLL) utilizing C# bindings. It was built for Microsoft Windows XP, Microsoft Word 2003, and Adobe Reader 7.[2]

AppMonitor has three parts: First, there is the main application program AppMonitor.exe, which is responsible for the majority of the system's functionality. Second, there is the event-hooking DLL, Hooker.dll, which is loaded into the memory space of the monitored applications. Third, there is a small secondary DLL, MSWordStat.dll, that has the sole purpose of determining the length of a document opened in Microsoft Word.

Figure 1 shows the high-level architecture of the AppMonitor system, with the arrows representing messages passing between layers and applications. The main AppMonitor application runs on top of the Windows XP platform and is responsible for the majority of the functionality in the system, including displaying the real-time list of events, keeping track of open applications and documents, monitoring their state, and coordinating the accompanying DLLs.

The hardware layer coordinates the transmission of input device signals (keyboard and mouse interrupts) to the operating system (Windows XP), which normally directs these messages to the appropriate application. However, AppMonitor places our event interception software (Hooker.dll) between the operating system and the monitored applications. Hooker.dll intercepts the mouse and keyboard events, records them if necessary, and passes them, unmodified, to the application. This is the method for obtaining low-level mouse events such as "left mouse button depressed" or "mouse moved" and low-level keyboard events such as "Ctrl-F pressed." These events are

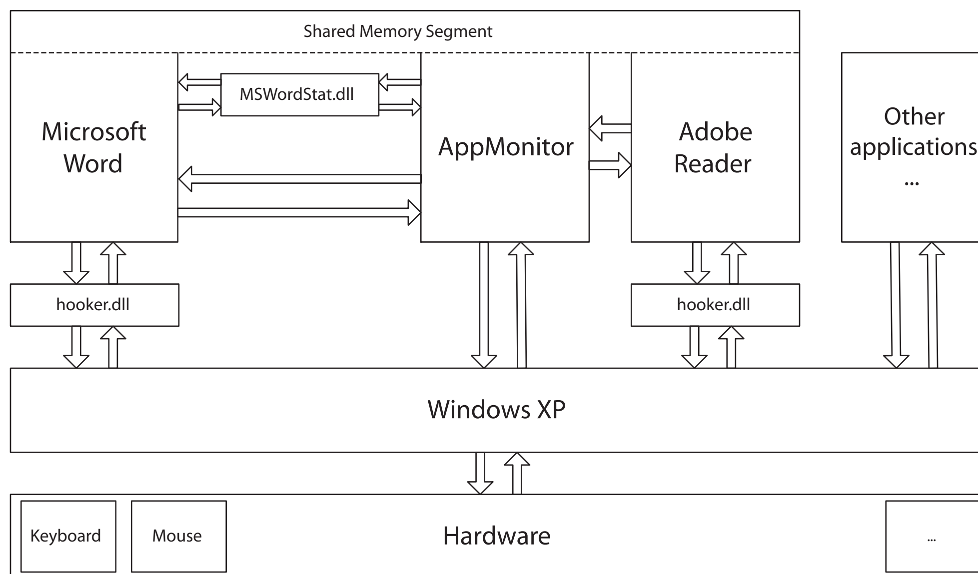


**Figure 1. AppMonitor architecture.**

subsequently communicated to AppMonitor using a segment of memory that is shared between AppMonitor and the applications. The main AppMonitor executable also communicates with the applications directly, using the Active Accessibility Interface. This communication allows AppMonitor to determine the interface semantics of the associated low-level event (e.g., the selection of a particular menu item, or the distance that the scrollbar is moved). AppMonitor also polls the monitored applications to determine whether the scrollbars or zoom of each document has changed. Polling is necessary because these interface controls change state not only through direct user manipulation, but also indirectly (and belatedly, because of threading) due to activities such as window resizing, changes of view, or keystrokes to add or delete text. Finally, the MSWordStat DLL is used to determine the length of Microsoft Word documents as they are opened.

The following sections further describe how AppMonitor works, giving details on how it uses the window hierarchy to determine interface semantics within applications and how both low- and high-level events are monitored.

### The Window Hierarchy

The *window hierarchy* describes the internal interface structure of all running desktop windows. Every application has at least one top-level window in the hierarchy, even when iconified. AppMonitor uses a Windows software development kit (SDK)[3] function called EnumWindows to gain identifiers to all of the top-level windows. Each window is tested with the GetClassName function to determine its parent application. Microsoft Word windows are instances of the class OpusApp, and Adobe Reader windows are instances of the class AdobeAcrobat. In the current version of AppMonitor, only windows belonging to Microsoft Word or Adobe Reader are further scrutinized, but extension to other applications should be relatively straightforward.

Two tools greatly assist programmers in comprehending the internal window structure of applications: Spy++ and Accessibility Explorer. Most Microsoft Windows–based graphical user interfaces (GUIs) have one high-level window containing a series of "child" windows (one for each toolbar and scrollbar, etc.) to create the interface seen by the user. Spy++[4] allows the developer to view the window handles, class names, and descriptions of all open windows and their "children." It also displays system-generated messages, such as mouse clicks, movements, and keypresses. Several similar tools exist to aid Microsoft .NET programmers using Windows Forms, such as ManagedSpy.[5]

Accessibility Explorer[6] allows programmers to view the accessible object tree, which gives important additional details of the constituent components of an application beyond that available with Spy++; for example, Spy++ describes a scrollbar as a single window, but Accessibility Explorer additionally discriminates between the scroll thumb, scroll gutter, and scroll-arrow components.

### Low-Level Event Logging

Microsoft Windows applications are event-driven, meaning that active components await input to be passed to them via a message queue.[7] AppMonitor intercepts low-level events using the Windows Hook mechanism,[8] which allows a program to be notified whenever another application is about to receive a message via its message queue. A "hook" is installed by using the SetWindowsHookEx function, passing it the address of a callback function (to be called when a message is received). The hook[9] callback function is responsible for passing the message on to the intended application. AppMonitor passes all messages on, unmodified, to the intended application to ensure that the application's behavior does not change.

AppMonitor's hook procedures are stored in a DLL (Hooker.dll), which is loaded into the address space of the logged application (Word or Reader). Keyboard and mouse events are both recorded as described below.

**Key events**. AppMonitor could record all keypresses, including regular typing events and keyboard shortcut commands. However, in order to reduce the participants' natural concern about our ability to reconstruct their documents, we have not recorded regular typing events in our studies to date. Consequently, we have only logged the following keyboard events: key combinations that include a modifier key (either Ctrl or Alt); the arrow keys; the function keys; the navigation keys (Page Up, Page Down, etc.); Enter; and Tab.

Keypresses are recorded in the log files using their virtual key code (e.g., Figure 3, Line 34). Key combinations that include a modifier are logically ORed with the modifier's key code, as described in Table 1. Note that the Shift key by itself is not considered to be a modifier. However, it is useful to record whether the Shift key is depressed when a modifier key is used; [Ctrl]+[Left Arrow], for example, will advance the cursor one word at a time, whereas [Ctrl]+[Shift]+[Left Arrow] will advance the cursor one word at a time while highlighting the text. A full list of key codes is available on Microsoft's Web site.[10]

**Mouse events**. The mouse hooking mechanism allows us to discriminate among all low-level mouse actions: movement, button depressions and releases, double clicks, scrollwheel use, and so on. For each mouse event, AppMonitor records the interface object beneath the cursor. For mouse movement events, we additionally record the cursor's screen coordinate.

Specialized mouse events can also be recorded. For example, we are particularly interested in scrolling and zooming actions, so we explicitly encode Ctrl–scrollwheel events, which Word and Reader both use for advanced zooming capabilities.

### High-Level Event Logging—WinEvents

High-level event logging involves recording "WinEvents" that are generated by an application when its logi-

**Table 1**
**Keyboard Modifier Masks**

| Modifier | Logical Mask |
|---|---|
| Ctrl | 0x100 |
| Alt | 0x200 |
| Shift | 0x400 |

cal state changes. This allows AppMonitor to record some of the semantics of interaction that cannot be inferred from low-level logs, such as “scrolling starting” or “menu item selected.” This is made possible by exploiting the WinEvent Hook functionality, which is part of the Active Accessibility application programming interface (API).

Active Accessibility[11] is an API within the Windows SDK that was designed to ease the construction of interfaces for physically handicapped people. For example, a programmer might use the Active Accessibility API to write a tool that allows a quadriplegic person to activate a menu using voice commands.

To hook WinEvents, a program calls the SetWinEventHook API function, passing the location of an associated callback function to be run when the event occurs. Like the mouse and keyboard hooks, the WinEvent hook procedure must also be placed inside a DLL—in our case, Hooker.dll.

**Polling**

Much of the interface state information, such as the scrollbar position and current document zoom, can only be determined by directly querying specific window objects inside an application. AppMonitor employs a polling mechanism to continually query each document in each application for any changes in state. Polling is necessary because certain interface components, such as scrollbars, can be updated without direct user intervention; for example, as a side effect of changing the zoom state. In our studies, we used a polling interval of 200 msec, which is a trade-off between increased CPU demands at short intervals and failure to register pertinent events with long ones.

**Events Unexposed by Previous Techniques**

In our experience, almost all of the events that may be of interest to researchers are exposed by the low-level, high-level, or polling data capture techniques. Sometimes, however, a researcher will be interested in system information not available through these channels. In such situations, a programmer may be able to find other means of determining the required information.

In deploying AppMonitor to help us understand how users navigate through their documents, we wanted to know each document’s length, in pages. Adobe Reader exposed this information directly, but Microsoft Word did not. We therefore wrote a special dynamically linked library (MSWordStat.dll) using the Microsoft Office interoperability API.

**AppMonitor Portability and Extendability**

We built AppMonitor to observe interaction with Microsoft Word 2003 and Adobe Reader 7 when they are running under Microsoft Windows XP. This section provides guidance for researchers who wish to update or extend AppMonitor to newer software versions and new applications.

AppMonitor is portable to any Microsoft Windows operating system that implements the technologies described in the System Architecture section. It can also be extended to monitor any Microsoft Windows application that implements the Microsoft Active Accessibility Interface. Doing so requires an understanding of, and ability to modify, the following aspects of AppMonitor’s operation.

AppMonitor recognizes when applications of interest are opened by regularly traversing the window hierarchy and comparing the root application class names with those that are to be logged. Once an application of interest is “discovered,” the window handle is used in the construction of an internal model for that application. To extend AppMonitor to a new application, researchers would need to determine the internal application class name and add custom code for model instantiation.

Microsoft Word and Adobe Reader required a finer grained model, one at document level rather than at application level. This therefore also required AppMonitor to continually inspect the application’s internal window hierarchy to determine whether new documents had been opened. Monitoring a new application will require the researcher to determine the granularity of model required, implement the model (see below), and write the application-specific code to allow these models to be instantiated at the correct time.

The internal models maintain window handle references and state information about the application or document under scrutiny. The model must maintain a reference to the root window handle of the application to ensure that it is only recognized as a new instance once. The model should also maintain state information and function implementations for any application-specific monitoring that is to occur. Generally, these functions will be called as part of the polling process, inspecting the internal state of the application, possibly through external DLLs.

Each application may also need custom keyboard or mouse “hooks” to be written inside Hooker.dll if additional computation or extended logging capabilities are required. For example, when the mouse wheel is moved, our mouse callbacks determine whether the Ctrl key is simultaneously pressed, and if so, record “CtrlScrollWheel” (instead of simply “ScrollWheel”), because this is an important method of zoom control in document navigation systems.

The AppMonitor system uses the Active Accessibility Interface to record interaction with menus, buttons, dialog boxes, and so forth. These should not need to be modified to record actions in other applications.

## EVENT CONFIGURATION, LOG FILES, AND EXAMPLE

AppMonitor provides researchers with a configuration GUI for tailoring the set of events to be logged when the software is distributed to participants (as shown in Figure 2). To ensure that a consistent set of data is collected, the configuration interface is disabled during longitudinal studies.

All events from a particular user are stored in a single log file on the local machine. This file is automatically uploaded to a remote Web server whenever it reaches a threshold size or after a week has passed since the last upload. A CGI script on the server receives the log file and then

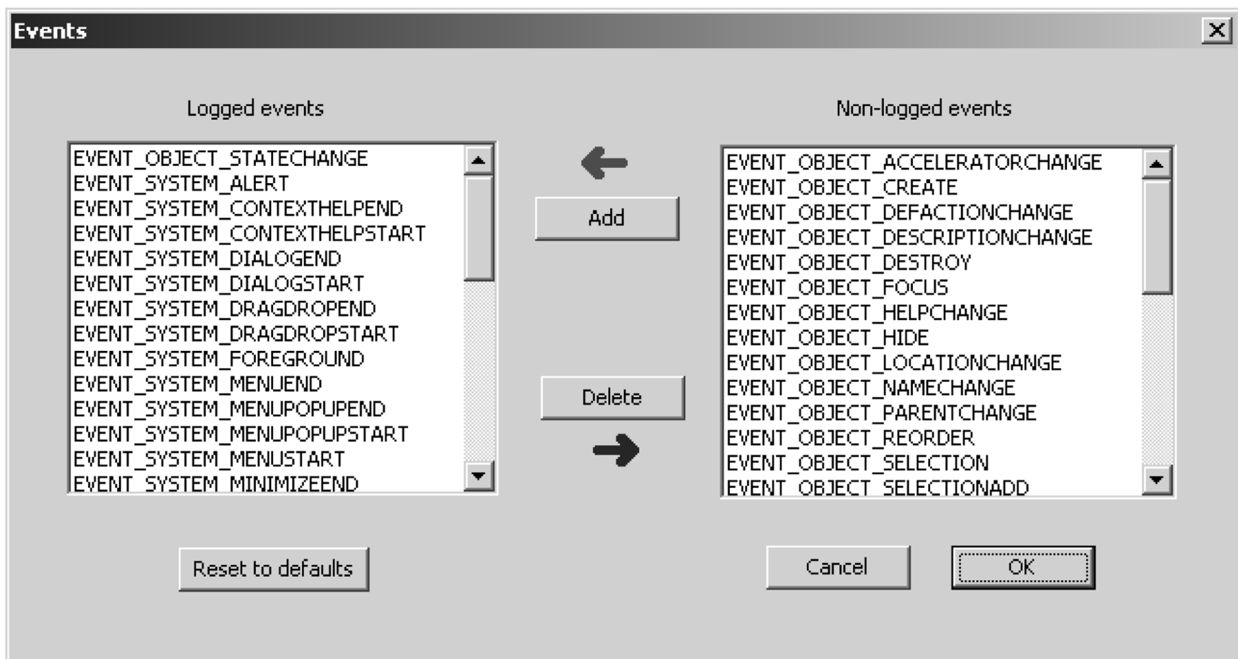


**Figure 2. AppMonitor configuration dialog.**

instructs the host machine to truncate the file it holds. This method of file transfer occurs over the HTTP port 80 (as used for Internet browsing) and so prevents the need for system administrators to allow traffic on other ports before AppMonitor can correctly function. Figure 3 shows a sample log file produced by AppMonitor when Microsoft Word is being used. Note that we have added line numbers for clarity of explanation. The log file demonstrates many of the capabilities of AppMonitor, as described below.

Every line in the log file begins with a date and timestamp (down to millisecond accuracy),[12] followed by an event code, one or more identifying window handles, and possibly further information about the event. There are two types of event codes:

1. Operating system event codes. These are identified by being fully capitalized in the log file and are generated by the Windows SDK (e.g., Figure 3, Line 4, "WM_LBUTTONDOWN"). These events originate from both the high- and low-level data collection techniques.

2. AppMonitor pseudoevent codes. These are the mixed-case event codes and are generated by AppMonitor (e.g., Figure 3, Line 1, "NewDocument"). These events originate from all other data collection techniques—for example, polling.

The window handles (hexadecimal numbers) uniquely identify the document and the application that has generated the event. These allow AppMonitor to distinguish between documents that are open concurrently in two applications.

Line 1 shows that AppMonitor has detected a document being opened in Microsoft Word. The code VE(0,255,0,62,0) describes the "Vertical East" scrollbar. The numerical values in the brackets describe the current state of the scrollbar, using the convention: (minimum trough value, maximum trough value, scrollbar static position, document thumb size, scrollbar dynamic position). The static scroll position is not updated when the user drags the scroll thumb; however, the dynamic position is updated, allowing one to determine the position of the scroll thumb even if scrolling is under way. In this case, we can see that the scroll trough extends from 0 (the top) to 255 (the bottom), with the static position of the thumb controller currently 0. The thumb size is 62/255 of the gutter length. Finally, it has a dynamic scroll position of 0.

Line 2 describes the state of the document at the time of opening. The document "AppMonitor.doc" has been opened in Microsoft Word's print layout view, with an initial zoom of 150%. Line 3 shows that the document is five pages long.

Lines 4–10 show the user scrolling from the top of the document to a position 5% of the way through the document (Line 8). The fact that the button comes up on the "grip" (Line 9) following use of the "Line down" control (Line 10) shows that the scrollbar's down arrow was used. Lines 12–16 record mouse scrollwheel actions. Each of the "ScrollbarsChanged" events records the position of the scrollbar at a particular point in time, allowing postprocessing to determine the speed and direction(s) of the scroll action.

Lines 17–20 show the user posting the "View" menu and then selecting the "Thumbnails" menu item (Lines 21 and 22). Lines 25–27 show scrollbar updates caused by displaying the thumbnail panel. This feedback can be distinguished from end-user scrolling, because it is not accompanied by a scroll action.

Lines 28–33 show the user changing the zoom level from 150% to 100% using the Zoom combo-box (Line 28). On Lines 34–47, the user carries out a copy, scroll, and paste action using Control-C (hex-code 0x143, constituting the

```
1   17/08/2006 14:21:42.265: NewDocument 0x30029c (0x2f0274) [MicrosoftWord] VE(0,255,0,62,0) HS(0,1355,0,1355,0)
2   17/08/2006 14:21:43.437: DocumentProperties 0x30029c (0x2f0274) "AppMonitor.doc - Microsoft Word" PrintLayoutView 150.00%
3   17/08/2006 14:21:50.657: PageStatusChanged 0x30029c (0x2f0274) 1 of 5
4   17/08/2006 14:22:23.805: WM_LBUTTONDOWN 0x30029c  VSCROLLBAR[0]
5   17/08/2006 14:22:23.805: EVENT_SYSTEM_SCROLLINGSTART 0x30029c
6   17/08/2006 14:22:24.366: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,2,15,0)
7   17/08/2006 14:22:24.766: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,6,15,0)
8   17/08/2006 14:22:25.167: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,11,15,0)
9   17/08/2006 14:22:25.247: WM_LBUTTONUP 0x30029c  grip
10  17/08/2006 14:22:25.247: EVENT_OBJECT_STATECHANGE 0x30029c  Line_down
11  17/08/2006 14:22:25.247: EVENT_SYSTEM_SCROLLINGEND 0x30029c
12  17/08/2006 14:22:25.367: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,11,15,11)
13  17/08/2006 14:22:32.367: WM_MOUSEWHEEL 0x30029c  Scrollwheel
14  17/08/2006 14:22:32.377: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,13,15,13)
15  17/08/2006 14:22:32.708: WM_MOUSEWHEEL 0x30029c  Scrollwheel
16  17/08/2006 14:22:33.178: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,16,15,16)
17  17/08/2006 14:22:41.140: WM_LBUTTONDOWN 0x30029c  View/menu_item
18  17/08/2006 14:22:41.140: EVENT_SYSTEM_MENUSTART 0x30029c  AppMonitor.doc_-_Microsoft_Word/Menu_Bar
19  17/08/2006 14:22:41.150: EVENT_SYSTEM_MENUPOPUPSTART 0x30029c  View
20  17/08/2006 14:22:41.240: WM_LBUTTONUP 0x30029c  View/menu_item
21  17/08/2006 14:22:44.835: WM_LBUTTONDOWN 0x30029c  Thumbnails/menu_item
22  17/08/2006 14:22:44.975: WM_LBUTTONUP 0x30029c  Thumbnails/menu_item
23  17/08/2006 14:22:44.975: EVENT_SYSTEM_MENUPOPUPEND 0x30029c  View
24  17/08/2006 14:22:44.995: EVENT_SYSTEM_MENUEND 0x30029c  AppMonitor.doc_-_Microsoft_Word/Menu_Bar
25  17/08/2006 14:22:45.116: ScrollbarsChanged 0x30029c (0x2f0274) HS(0,1284,114,634, 114)
26  17/08/2006 14:22:45.196: ScrollbarSetChanged 0x30029c (0x2f0274) VE(0,255,16,15,16) HS(0,1284,114,634,114) (0,255,0,224,0)
27  17/08/2006 14:22:45.396: ScrollbarsChanged 0x30029c (0x2f0274) VC(0,255,0,224,0)
28  17/08/2006 14:23:09.100: WM_LBUTTONDOWN 0x30029c  Zoom:/combo_box[150%]
29  17/08/2006 14:23:09.100: EVENT_SYSTEM_MENUSTART 0x30029c  AppMonitor.doc_-_Microsoft_Word/Menu_Bar
30  17/08/2006 14:23:10.632: WM_LBUTTONUP 0x30029c  100%/list_item
31  17/08/2006 14:23:10.652: EVENT_SYSTEM_MENUEND 0x30029c  AppMonitor.doc_-_Microsoft_Word/Menu_Bar
32  17/08/2006 14:23:10.833: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,0,22,0) HS(0,856,68,634,68)
33  17/08/2006 14:23:10.833: ZoomChanged 0x30029c (0x2f0274) 100.00%
34  17/08/2006 14:23:38.422: WM_KEYDOWN 0x30029c  0x143
35  17/08/2006 14:23:38.593: WM_KEYUP 0x30029c  0x143
36  17/08/2006 14:23:42.779: WM_LBUTTONDOWN 0x30029c  VSCROLLBAR[0]
37  17/08/2006 14:23:42.789: EVENT_SYSTEM_SCROLLINGSTART 0x30029c
38  17/08/2006 14:23:43.279: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,2,22,2)
39  17/08/2006 14:23:43.680: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,73,22,73)
40  17/08/2006 14:23:44.080: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,146,22,146)
41  17/08/2006 14:23:44.281: ScrollbarsChanged 0x30029c (0x2f0274) VE(0,255,157,22,157)
42  17/08/2006 14:23:44.491: WM_LBUTTONUP 0x30029c  VSCROLLBAR[61]
43  17/08/2006 14:23:44.501: EVENT_SYSTEM_SCROLLINGEND 0x30029c
44  17/08/2006 14:23:46.945: WM_LBUTTONDOWN 0x30029c  Microsoft_Word_Document/ client
45  17/08/2006 14:23:47.075: WM_LBUTTONUP 0x30029c  Microsoft_Word_Document/client
46  17/08/2006 14:23:49.909: WM_KEYDOWN 0x30029c  0x156
47  17/08/2006 14:23:50.079: WM_KEYUP 0x30029c  0x156
48  17/08/2006 14:24:06:864: WM_LBUTTONDOWN 0x30029c  Close/push_button
49  17/08/2006 14:24:07.805: WM_LBUTTONUP 0x30029c  Close/push_button
50  17/08/2006 14:24:07.915: ScrollbarSetChanged 0x30029c (0x2f0274)
```

**Figure 3. Sample AppMonitor log.**

virtual key code for the "c" character [0x43] logically ORed with a mask represented in the Ctrl modifier [0x100] on Lines 34 and 35, the scroll thumb (Lines 36–43) and Control-V (Lines 46 and 47).

Finally, the user closes the application using the close button at the top right corner of the window (Lines 48–50).

To ease data analysis, AppMonitor's log files follow a BNF syntax definition, as described in Figure 4. In this definition, nonterminals are enclosed in angle brackets, and terminals are in italics. (Note: A "HookerEventCode" is any of the "operating system event codes" described earlier in this section.)

## PRELIMINARY RESULTS

Before beginning a comprehensive field study, six beta testers used our software for a period of 4 months. Our original objective was to characterize how people navigate within documents using tools like the scrollbar (thumb, gutter, arrows), rate-based scrolling, the scrollwheel, zooming, split windows, thumbnails, Find, and so on. However, we quickly realized that, with little additional work, AppMonitor could be enhanced to support much broader characterization of interaction, and so we generalized its capabilities.

AppMonitor's logs are detailed, and they will contain a superset of the information required for any individual research question. They are a rich resource for answering specific and detailed research questions about interaction. We have used computer programs written as Python scripts to extract pertinent events and statistically aggregate their occurrences for each user and across users.

This section provides examples of data gathered during beta testing. Some results for Microsoft Word are summarized in Table 2. Our primary objective here is to exemplify the use of AppMonitor's data, rather than to present a detailed analysis of any particular aspect of interaction with Microsoft Word.

### Number of Documents

Our six testers interacted with between 26 and 223 Word documents ($M = 92$, $SD = 75$; Table 2, Line 1), and between 39 and 177 PDF documents in Adobe Reader ($M = 119$, $SD = 52$; Line 2).

### Keyboard Commands

We logged 45,532 occurrences of 127 different keyboard commands (which we interpret as any keyboard action that does not enter or delete text, including cursor arrow keys and Page Up/Page Down, etc.). There were few keyboard command events in Adobe Reader (except for Page Up/Page Down). Keyboard command use in Word is summarized on Lines 3 and 4 of Table 2. Participant 5 had

```
<LogfileLine> ::= <DateAndTime> : <Event>
<Event> ::= <InternalEvent> | <HookerEvent>
<DateAndTime> ::= <Date><Time>
<Date> ::= <Day>/<Month>/<Year>
<Day> ::= <Integer>
<Month> ::= <Integer>
<Year> ::= <Integer>
<Time> ::= <Hours> : <Minutes> : <Seconds>
<Hours> ::= <Integer>
<Minutes> ::= <Integer>
<Seconds> ::= <Integer>.<Integer>        // seconds.milliseconds
<InternalEvent> ::=  <AppMonitorEvent> | < DocumentState > | <Debug> | <LogSent> | <DocumentProperties>
<AppMonitorEvent> ::= AppMonitorStarted | AppMonitorExit | EventOptionsChanged
<DocumentState> ::= <NewDoc> | <DeadDoc> | <ScrollbarSetChange> | <ScrollbarChange> | <PageStatusChange> | <ZoomChange>
<Debug> ::= Debug <String>
<LogSent> ::=  ResumeAfterSendingFile <Filename>
<DocumentProperties> ::= DocumentProperties  <WindowHandle> (<WindowHandle>) <String> <ViewType> <Zoom>
<NewDoc> ::= NewDocument <WindowHandle> (<WindowHandle>) <ApplicationName> <ScrollbarStatusList>
<DeadDoc> ::= DeadDocument <WindowHandle> (<WindowHandle>)
<ScrollbarSetChange> ::= ScrollbarSetChanged <WindowHandle> (<WindowHandle>) <ScrollbarStatusList>
<ScrollbarChange> ::= ScrollBarsChanged <WindowHandle> (<WindowHandle>) <ScrollbarStatusList>
<PageStatusChange> ::= PageStatusChanged <WindowHandle> (<WindowHandle>) <Integer> of <Integer>
<ZoomChange> ::= ZoomChanged <WindowHandle> (<WindowHandle>) <Zoom>
<ApplicationName> ::= [MicrosoftWord] | [AdobeReader]
<ScrollbarStatusList> ::= <ScrollbarStatus> | <ScrollbarStatus> <ScrollbarStatusList>
<ScrollbarStatus> ::= <ScrollbarID> <ScrollbarState>
<ScrollbarID> ::= VE | VC | VU | VL | VX | HS | HC | X | VT | HT | VB | HB | VR | HR
<ScrollbarState> ::= ( <ScrollbarMin>,<ScrollbarMax>,<ScrollbarStatic>,<Thumbsize>,<ScrollbarDynamic> )
<ViewType> ::= UnknownView | NormalView | PrintLayoutView | OutlineView | ReadingLayoutView | WebLayoutView | SinglePageView | ContinuousView |
          ContinuousFacingView | FacingView
<Zoom> ::= <PercentToken>
< HookerEvent > ::= < HookerEventCode>  [<WindowHandle> [<EventInformation>]]
< HookerEventCode > ::= EVENT_SYSTEM_SOUND | EVENT_SYSTEM_ALERT | ... | EVENT_OBJECT_ACCELERATORCHANGE
<WindowHandle> ::= <HexadecimalInteger>
<ScrollbarMin> ::= <Integer>
<ScrollbarMax> ::= <Integer>
<ScrollbarStatic> ::= <Integer>
<Thumbsize> ::= <Integer>
<ScrollbarDynamic> ::= <Integer>
<EventInformation> ::= <Name> | <Name> / <Name>
<Name> ::= any token
<Filename> ::= string token   // A token delimited by double-quotes
<String> ::= string token       // A token delimited by double-quotes
<HexadecimalInteger>::= a token that represents a hexadecimal number, e.g. 0x125a4fc
<Integer> ::= a token made up of digits only
<PercentToken> ::= a token in which the last character is '%' and all other characters are digits
```

**Figure 4. BNF definition of log structure.**

the largest keyboard "vocabulary," using 74 different keyboard commands. Only 8 keyboard commands were used by all participants: Enter; Tab; the left, right, and down arrows; Ctrl-C; Ctrl-V; and Ctrl-Z.

## Interface Buttons

The Close, Maximize/Restore, and Minimize window buttons accounted for 486, 66, and 259 events, respectively. Seventy-two different toolbar buttons were used, producing 1,718 button selections. The set of buttons used across participants was dissimilar, except for the Zooming controls in Reader, which were heavily used by all but 2 participants. Button use in Word is summarized on Lines 5 and 6 of Table 2.

## Zipf's Law of Item Frequency

Regression analysis showed that all users strongly adhered to a Zipfian distribution of button use (Line 7 of Table 2). Zipf's (1949) law states that the frequency of any word in human natural language is roughly inversely proportional to its rank in a frequency table, and previous research has demonstrated that Zipf's law also holds for command and menu item frequency (Findlater & McGrenere, 2004; Greenberg & Witten, 1993).

## Scrolling Actions

Lines 8 and 9 of Table 2 describe the two most commonly used scrolling techniques as a percentage of the total distance scrolled in all documents (note that these do not amount to 100%, because minor techniques are not shown). The mouse scrollwheel and the scroll thumb account for between 68% and 95% of the total distance scrolled by all of our beta testers.

Our aim as HCI researchers is to improve interfaces that are used every day. The results described here can provide insight into how AppMonitor's logs allow statistical characterization of interface use. For example, the frequency of use of menu items and buttons raises questions about item placement and shortcut facilities, and the scrolling analysis allows us to model and characterize a user's document navigation and imply whether particular navigation tools are under- or overutilized.

**Table 2**
**Sample Results From Beta Testers**

| | Beta Tester | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| Document Counts | | | | | | |
| Microsoft Word | 62 | 55 | 26 | 139 | 223 | 49 |
| Adobe Reader | 177 | 123 | 85 | 122 | 39 | 169 |
| Microsoft Word | | | | | | |
| KB vocabulary | 60 | 27 | 24 | 63 | 74 | 32 |
| KB command use count | 14,849 | 491 | 2,041 | 8,004 | 9,086 | 11,061 |
| GUI button vocabulary | 38 | 12 | 11 | 26 | 30 | 22 |
| GUI button use count | 472 | 54 | 88 | 451 | 398 | 255 |
| Menu selection, Zipf's law $R^2$ | .93 | .96 | .93 | .98 | .99 | .97 |
| Mouse wheel,% distance | 71 | 12 | 47 | 50 | 58 | 41 |
| Scroll thumb,% distance | 7 | 79 | 35 | 23 | 10 | 54 |

## CONCLUSION

Automated logging of user events is an important facility for HCI research, enabling detailed longitudinal usage analysis that would be prohibitively expensive to conduct through other methods. The complexity of software development has been a substantial barrier to the widespread use of client-side logs; but, although still complex, various APIs and tools have eased development. This article presents a high-level overview of the techniques used to develop and test AppMonitor, an event logging system. We intend to stimulate others to develop and deploy logging systems, and to help them in doing so. AppMonitor's software is available by contacting the authors. We have now widely deployed AppMonitor, and our future work will focus on characterizing how users navigate within documents.

### AUTHOR NOTE

We thank the study participants, and the anonymous reviewers for their informative comments. This research was partially funded by New Zealand's Royal Society Marsden Grant 07-UOC-013. Correspondence concerning this article should be addressed to J. Alexander, A. Cockburn, and R. Lobb, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch 8140, New Zealand (e-mail: jason@cosc.canterbury.ac.nz; andy@cosc.canterbury.ac.nz; richard.lobb@cosc.canterbury.ac.nz).

### REFERENCES

Findlater, L. & McGrenere, J. (2004). A comparison of static, adaptive, and adaptable menus. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 89-96). Vienna: ACM Press.

Greenberg, S., & Witten, I. H. (1993). Supporting command reuse: Mechanisms for reuse. *International Journal of Man–Machine Studies*, **39**, 391-425.

Holtzblatt, K., & Jones, S. (1993). Contextual inquiry: A participatory technique for system design. In D. Schuler & A. Namioka (Eds.), *Participatory design: Principles and practice* (pp. 180-193). Hillsdale, NJ: Erlbaum.

Kellar, M., Hawkey, K., Inkpen, K. M., & Watters, C. (in press). Challenges of capturing natural Web-based user behaviours. *International Journal of Human–Computer Interaction*.

Kukreja, U., Stevenson, W. E., & Ritter, F. E. (2006). RUI—Recording user input from interfaces under Windows and Mac OS X. *Behavior Research Methods*, **38**, 656-659.

Linton, F., Joy, D., Schaefer, H.-P., & Charron, A. (2000). OWL: A recommender system for organization-wide learning. *Educational Technology & Society*, **3**, 62-76.

McGrenere, J. (2002). The design and evaluation of multiple interfaces: A solution for complex software. Unpublished doctoral dissertation, University of Toronto.

Tauscher, L., & Greenberg, S. (1997). How people revisit Web pages: Empirical findings and implications. *International Journal of Human–Computer Studies*, **47**, 97-138.

Trewin, S. (1998). InputLogger: General-purpose logging of keyboard and mouse events on an Apple Macintosh. *Behavior Research Methods, Instruments, & Computers*, **30**, 327-331.

Westerman, S. J., Hambly, S., Alder, C., Wyatt-Millington, C. W., Shryane, N. M., Crawshaw, C. M., & Hockey, G. R. J. (1996). Investigating the human–computer interface using the Datalogger. *Behavior Research Methods, Instruments, & Computers*, **28**, 603-606.

Zipf, G. (1949). *Human behavior and the principle of least effort: An introduction to human ecology*. Reading, MA: Addison-Wesley.

### NOTES

1. www.microsoft.com/products/ceip/en-en/default.mspx.
2. AppMonitor has also been successfully tested on Microsoft Windows Vista. See the AppMonitor Portability and Extendability section for a discussion on modifications required to log different versions of Microsoft Word and Adobe Reader.
3. msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/windowing/windows.asp.
4. msdn.microsoft.com/library/default.asp?url=/library/en-us/vcug98/html/_asug_home_page.3a_.spy.2b2b.asp.
5. msdn.microsoft.com/msdnmag/issues/06/04/managedspy/.
6. www.microsoft.com/downloads/details.aspx?familyid=3755582a-a707-460a-bf21-1373316e13f0&displaylang=en.
7. msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/messagesandmessagequeues/aboutmessagesandmessagequeues.asp.
8. msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/windowing/hooks.asp.
9. msdn2.microsoft.com/en-us/library/ms997537.aspx.
10. msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/WinUI/WindowsUserInterface/UserInput/VirtualKeyCodes.asp.
11. msdn2.microsoft.com/en-us/library/ms697707.aspx.
12. AppMonitor timestamps all registered events. However, the operating system may introduce a slight delay between an event occurring and AppMonitor receiving notification (for example, under high-load conditions). Hence, millisecond timestamp accuracy cannot be guaranteed.

(Manuscript received May 8, 2007;
revision accepted for publication September 18, 2007.)