Production Number: B339

AppMonitor: A Tool for Recording User Actions in Unmodified Windows Applications

Jason Alexander, Andy Cockburn and Richard Lobb
Department of Computer Science and Software Engineering
University of Canterbury
Private Bag 4800
Christchurch 8140
New Zealand

{jason, andy, richard.lobb}@cosc.canterbury.ac.nz

+64 3 364 2987 x7755

Publication Number

B339

Running Head:

AppMonitor: A User Action Logger for Microsoft Windows

Production Number: B339

AppMonitor: A Tool for Recording User Actions in Unmodified Windows Applications

Jason Alexander, Andy Cockburn and Richard Lobb

Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand

{jason, andy, richard.lobb}@cosc.canterbury.ac.nz

Abstract

This paper 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, behavioural patterns prior to or following command use, and methods of navigating through systems and datasets. *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 subjects 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 multi-level event loggers we describe *AppMonitor’s* underlying architecture and implementation, and provide a brief example of the data generated during our four month trial with six users.

Keywords

Event logging, Client-side logging, Human-Computer Interaction

Introduction

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 and Jones 1993). Client-side logging of user actions is another technique, which has several advantages over other methods: it is highly scalable as software is easily disseminated over the web; it is low cost to the participants and experimenters because once installed the logs are generated and stored automatically; it readily allows longitudinal analysis of weeks, months or years of use; it allows fine-grain analysis of sub-second 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/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 analysing web navigation behaviour, participants have been asked to abandon their preferred proprietary web-browsers in favour of customised logging versions of open source software (e.g. Tauscher and Greenberg 1997) or to use 'roll-your-own' systems (e.g. Kellar, Hawkey et al. 2007). Clearly it is preferable that 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 which presents the features in a different manner.

In this paper 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 key-presses, 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), because video-logs require human interpretation to determine the semantics connecting their key- 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, followed by 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 four month trial period.

Related Work

Direct human observation of interaction allows an immediate and contextually rich understanding of how a user interacts with their system. The substantial disadvantage of direct human observation is the high cost 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, and 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 (www.techsmith.com) allow researchers to record the visual state of the user's screen or window while they conduct their work. The recordings are often enhanced with low-level logs of user events such as mouse movement and clicks, allowing researchers to visually emphasise 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 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 (for example, 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 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 events when they 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, then videos of the user's interaction with the system can be recreated from these log files. The primary problem with client-side logging, however, is the complexity of creating software that, essentially, '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, Hambly 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 key strokes, 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 et al. 2000; McGrenere 2002). McGrenere (2002) describes MSTracker, an "internal tool used by the usability team at Microsoft" which is unavailable to other researchers. Like *AppMonitor*, the MSTracker tool listens to the Microsoft Active Accessibility events that are generated by most Windows applications. Linton, Joy et al. (2000) describe the Organisation-Wide Learning (OWL) system. It is a macro based logger that records high level commands issued to Microsoft Word running on Apple Mac 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 this data allows Microsoft to improve their products, the data is unavailable to the user or researchers (as it is binary coded), and the events logged are not configurable. Very little information regarding this system, what it logs and the results of studies are available 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 button presses 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 button presses and key sequences, making the data they generate largely insufficient for interpreting the semantics of interface manipulation.

[1] << footnote text listed at the end of the document >>

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 minimise the impact it has on users, both during installation and subsequent use. We have found no instances where firewall, anti-virus 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 installing *AppMonitor*, 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 start-up 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 key strokes, 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. Having 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) utilising 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 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

[2] << Footnote text listed at the end of the document >>

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 message 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 co-ordinating the accompanying DLLs.

<< Insert figure 1 about here >>

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 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. that a particular menu item is selected, 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 windows 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 function `GetClassName` 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 scrutinised, 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 GUIs have one high level window, containing a series of 'child' windows (one for each toolbar, 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 key presses. 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++.

[3] << Footnote text listed at the end of the document >>

[4] << Footnote text listed at the end of the document >>

[5] << Footnote text listed at the end of the document >>

[6] << Footnote text listed at the end of the document >>

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 onto the intended application. *AppMonitor* passes all messages on, unmodified, to the intended application to ensure the application's behaviour 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 further below.

Key Events

AppMonitor could record all key presses, including regular typing events and keyboard shortcut commands. However, in our studies to date we have not recorded regular typing events in order to reduce the participants' natural concern about our ability to reconstruct their documents. 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.

---

[7] << Footnote text listed at the end of the document >>

[8] << Footnote text listed at the end of the document >>

[9] << Footnote text listed at the end of the document >>

Key presses 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 if the shift key is depressed when a modifier key is used. For example, [Ctrl]+[Left Arrow] will advance the cursor one word at a time, while [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 website[10].

<< Insert Table 1 about here >>

Mouse Events

The mouse hooking mechanism allows us to discriminate between all low-level mouse actions: movement, button depressions and releases, double clicks, scroll-wheel 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 co-ordinate.

Specialised 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 logical 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).

[10] << Footnote text listed at the end of the document >>

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 200ms, 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 that is 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.

[11] << Footnote text listed at the end of the document >>

AppMonitor Portability and Extendability

We built *AppMonitor* to observe interaction with Microsoft Word 2003 and Adobe Reader 7 when running under Microsoft Windows XP. This section provides guidance for researchers who wish to update or extend *AppMonitor* to newer software versions and to 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 recognises when applications of interest are opened by regularly traversing the window hierarchy, 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 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 it is only recognised 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 mousewheel is moved, our mouse callbacks determine whether the 'Ctrl' key is simultaneously pressed, and if so, record 'CtrlScrollWheel' (instead of simply 'ScrollWheel'), as 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 etc. 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 show in Figure 2). To ensure a consistent set of data is collected, the configuration interface is disabled during longitudinal studies.

<< Insert Figure 2 about here >>

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 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.

<< Insert Figure 3 about here >>

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:

[12] << Footnote text listed at the end of the document >>

Operating System event codes. These are identified by being fully capitalised in the log file and are generated by the Windows SDK (for example,Figure 3, line 4, “WM_LBUTTONDOWN”). These events originate from both the high and low level data collection techniques.

AppMonitor Pseudo-event codes. These are the mixed case event codes and are generated by AppMonitor (for example, 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 application that has generated the event. These allow *AppMonitor* to distinguish between documents that are open concurrently in both the same application and in a different application.

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 us 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$^{\text{ths}}$ of the gutter length. Finally, it has a dynamic scroll position of 0.

The second line 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) having used the ‘Line down’ control (line 10) shows that the scrollbar’s down arrow was used. Lines 12–16 record mouse scroll wheel actions. Each of the “ScrollbarsChanged” events record the position of the scrollbar at a particular point in time, allowing post processing 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 21–22). Lines 25–27 show scrollbar updates caused by displaying the thumbnail panel. This feedback can be discriminated 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/paste action using Control-C (hex-code 0x143, constituting the virtual key code for the ‘c’ character (0x43) logically ‘ORed’ with a mask represented in the Ctrl modifier (0x100)) on lines 34–35, the scroll thumb (lines 36–43) and Control-V (lines 46–47).

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

<< Insert figure 4 about here >>

To ease data analysis, *AppMonitor’s* log files follow a BNF syntax definition, as described in Figure 4. In this definition, non-terminals 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 four months. Our original objective was to characterise how people navigate within documents using tools like the scrollbar (thumb, gutter, arrows), rate-based scrolling, the scroll-wheel, zooming, split windows, thumbnails, ‘Find’, and so on. We quickly realised, however, that with little additional work *AppMonitor* could be enhanced to support much broader characterisation of interaction, and so we generalised 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.

<< Insert Table 2 about here >>

This section provides examples of data gathered during beta testing. Some results for Microsoft Word are summarised 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 (mean 92, sd 75; Table 2, line 1), and between 39 and 177 PDF documents in Adobe Reader (mean 119, sd 52; line 2).

Keyboard commands. We logged 45532 occurrences of 127 different keyboard commands (which we interpret as any keyboard action that does not enter/delete text, including cursor arrow keys and page-up/down, etc.). There were few keyboard command events in Adobe Reader (except for page-up/down). Keyboard command use in Word is summarised on lines 3 and 4 of Table 2. Participant 5 had the largest keyboard 'vocabulary', using 74 different keyboard commands. Only eight 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 1718 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 two participants. Button use in Word is summarised on lines 5-6 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 law (1949) 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 (Greenberg and Witten 1993; Findlater and McGrenere 2004).

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 these do not sum to 100%, as minor techniques are not shown). The mouse scroll wheel 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 the interfaces that we use everyday. The results described here can provide insight into how *AppMonitor's* logs allow statistical characterisation 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 characterise user's document navigation and imply whether particular navigation tools are under- or over-utilised.

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 paper presented 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 characterising how users navigate within documents.

Acknowledgements

Many thanks to the anonymous reviewers for their informative comments, and to the study participants. This research was partially funded by New Zealand's Royal Society Marsden grant, number 07-UOC-013.

Figure Captions

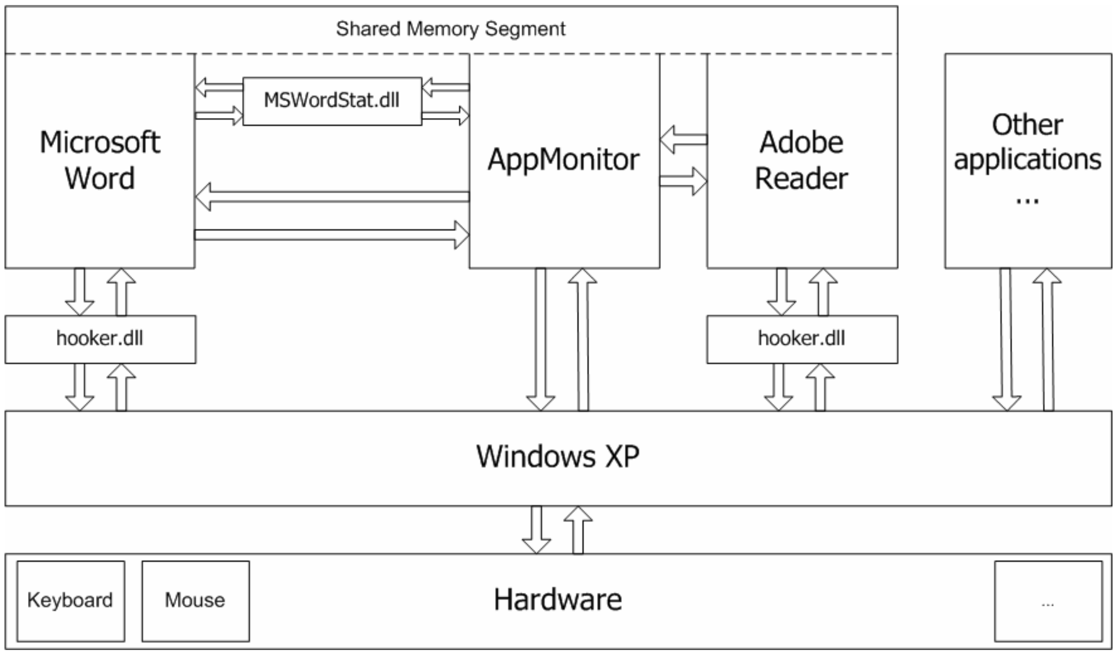
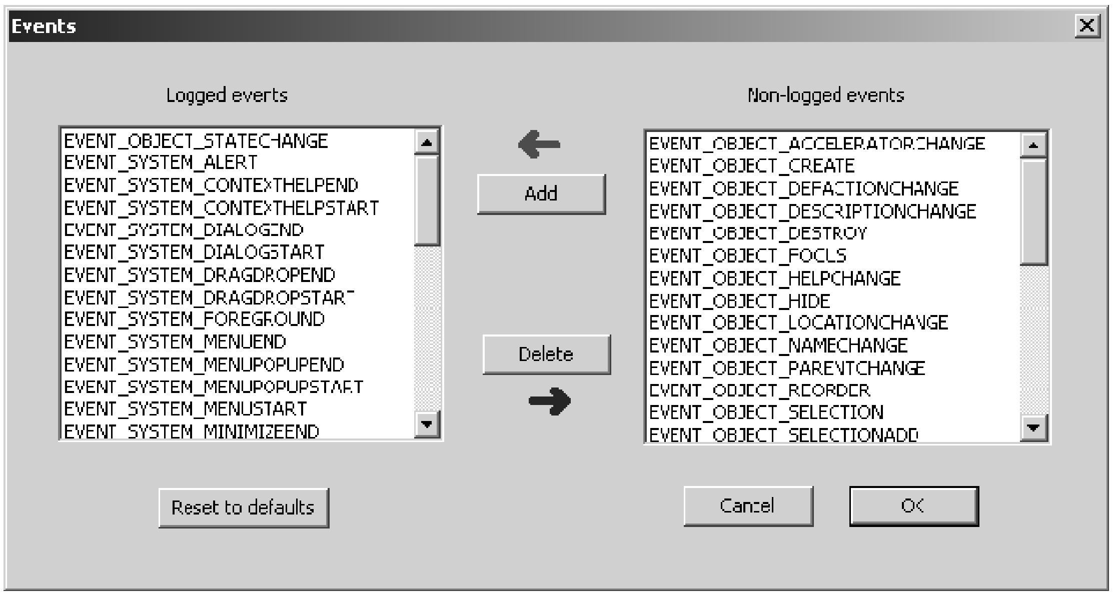


Figure 1: ***AppMonitor*** **architecture**



```
EVENT_OBJECT_STATECHANGE
EVENT_SYSTEM_ALERT
EVENT_SYSTEM_CONTEXTHELPEND
EVENT_SYSTEM_CONTEXTHELPSTART
EVENT_SYSTEM_DIALOGEND
EVENT_SYSTEM_DIALOGSTART
EVENT_SYSTEM_DRAGDROPEND
EVENT_SYSTEM_DRAGDROPSTART
EVENT_SYSTEM_FOREGROUND
EVENT_SYSTEM_MENUEND
EVENT_SYSTEM_MENUPOPUPEND
EVENT_SYSTEM_MENUPOPUPSTART
EVENT_SYSTEM_MENUSTART
EVENT_SYSTEM_MINIMIZEEND
```

```
EVENT_OBJECT_ACCELERATORCHANGE
EVENT_OBJECT_CREATE
EVENT_OBJECT_DEFACTIONCHANGE
EVENT_OBJECT_DESCRIPTIONCHANGE
EVENT_OBJECT_DESTROY
EVENT_OBJECT_FOCUS
EVENT_OBJECT_HELPCHANGE
EVENT_OBJECT_HIDE
EVENT_OBJECT_LOCATIONCHANGE
EVENT_OBJECT_NAMECHANGE
EVENT_OBJECT_PARENTCHANGE
EVENT_OBJECT_REORDER
EVENT_OBJECT_SELECTION
EVENT_OBJECT_SELECTIONADD
```

Figure 2: *AppMonitor* configuration dialog

Modifier	**Logical Mask**
Ctrl	0x100
Alt	0x200
Shift	0x400

Table 1: Keyboard modifier masks

```
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

```
<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

		Beta Tester					
		1	2	3	4	5	6
Document counts							
1.	Word	62	55	26	139	223	49
2.	Reader	177	123	85	122	39	169
Microsoft Word							
3.	KB vocabulary	60	27	24	63	74	32
4.	KB command use count	14849	491	2041	8004	9086	11061
5.	GUI button vocabulary	38	12	11	26	30	22
6.	GUI button use count	472	54	88	451	398	255
7.	Menu Selection, Zipf's Law R^2	0.93	0.96	0.93	0.98	0.99	0.97
8.	Mouse wheel, % distance	71	12	47	50	58	41
9.	Scroll thumb, % distance	7	79	35	23	10	54

Table 2: Sample results from beta testers

Footnotes

1. http://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. http://msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/windowing/windows.asp

4. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcug98/html/_asug_home_page.3a_.spy.2b2b.asp

5. http://msdn.microsoft.com/msdnmag/issues/06/04/managedspy/

6. http://www.microsoft.com/downloads/details.aspx?familyid=3755582a-a707-460a-bf21-1373316e13f0&displaylang=en

7. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/messagesandmessagequeues/aboutmessagesandmessagequeues.asp

8. http://msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/windowing/hooks.asp

9. http://msdn2.microsoft.com/en-us/library/ms997537.aspx

10. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/WinUI/WindowsUserInterface/UserInput/VirtualKeyCodes.asp

11. http://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.

References

Findlater, L. and McGrenere, J. (2004). A Comparison of Static, Adaptive, and Adaptable Menus. SIGCHI conference on Human factors in Computing Systems, Vienna, Austria, ACM Press 89–96.

Greenberg, S. and Witten, I. H. (1993). "Supporting Command Reuse: Mechanisms for Reuse." International Journal Man-Machine Studies **39**(3): 391–425.

Holtzblatt, K. and Jones, S. (1993). Contextual Inquiry: A Participatory Technique for System Design. Participatory Design: Principles and Practice. D. Schuler and A. Namioka. Hillsdale, N.J, Lawerence Earlbaum**:** 180–193.

Kellar, M., Hawkey, K., Inkpen, K. M. and Watters, C. (2007). "Challenges of Capturing Natural Web-based User Behaviours." International Journal Human-Computer Interaction, In Press.

Kukreja, U., Stevenson, W. E. and Ritter, F. E. (2006). "RUI—Recording User Input from Interfaces under Windows and Mac OS X." Behavior Research Methods **38**(4): 656–659.

Linton, F., Joy, D., Schaefer, H.-P. and Charron, A. (2000). "OWL: A Recommender System for Organization-Wide Learning." Educational Technology & Society **3**(1): 62–76.

McGrenere, J. (2002). The Design and Evaluation of Multiple Interfaces: A Solution for Complex Software. Computer Science. Toronto, University of Toronto. **Doctor of Philosophy**.

Tauscher, L. and Greenberg, S. (1997). "How People Revisit Web Pages: Empirical Findings and Implications." International Journal of Human-Computer Studies **47**(1): 97–138.

Trewin, S. (1998). "InputLogger: General-Purpose Logging of Keyboard and Mouse Events on an Apple Macintosh." Behavior Research Methods, Instruments and Computers **30**(2): 327–331.

Westerman, S. J., Hambly, S., Alder, C., Wyatt-Millington, C. W., Shryane, N. M., Crawshaw, C. M. and Hockey, G. R. J. (1996). "Investigating the Human-Computer Interface using the Datalogger." Behavior Research Methods, Instruments and Computers **28**(4): 603–606.

Zipf, G. (1949). Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. Reading, Mass., Addison-Wesley.