SimUI: Graphical User Interface Evaluation Using Playback

Nobuko Kishi

Department of Mathematics
Tsuda College
Tsuda-machi, Kodaira-shi,
Tokyo 187, Japan
Phone: +81-423-42-5160
E-Mail: kishi@tsuda.ac.jp

ABSTRACT

This paper describes a set of tools for evaluating graphical user interface designs during software development. These tools record mouse and keyboard operations by users and compare them to detect potential usability problems. The tools' two main techniques are data gathering in playback mode and multi-step matching of recorded data. A preliminary experiment showed that the tools can automate part of the usability evaluation process by detecting differences often overlooked by human observers.

INTRODUCTION

One of basic design goals of user interface design is to match a system's behavior with the users' perceived model of the system's behavior. It is assumed that the smaller the distance between the system's structure and the users' mental model, the less mental effort the users have to make when they operate the system [1, 9]. Although this claim is yet to be confirmed, current methods of usability evaluation in actual software development processes are based on this assumption. Users are asked to operate the pre-release version of a system and the evaluators try to find usability problems by observing any unexpected behavior of the users.

This method of usability evaluation has two drawbacks [3]. One is the variation caused by evaluators: human evaluators often overlook subtle phenomena that appear in users' behavior. The other is the high cost of analysis. In order to locate design problems in audio and video recording of users' behavior, it is necessary to play back the recordings several times.

To overcome these drawbacks and reduce the cost of usability evaluation, we developed a set of tools, collectively named SimUI (SIMulated User Interaction evaluation tools). These provide functions for recording, playback, and automatic analysis of user operations on applications under the OS/2 Presentation Manager. The tools' main techniques are data gathering in playback mode and multi-step matching of recorded data.

Data gathering in playback mode involves collecting data on the status of an application as well as users' operations while replaying recorded data on users' operations. This method has several advantages over the existing method, which involves data gathering while recording users' operations [7]. One is that the recording can be done in a way that is not intrusive to users. Another is that usability evaluators can analyze the data at their own pace and can repeat the analysis if necessary. However, previous approaches to data gathering in playback mode are not applicable to graphical user interfaces, because of problems with reproducibility [8]. SimUI's data gathering tool achieves better reproducibility by using process communication mechanisms.

Multi-step matching is a technique that we developed for analyzing message sequences generated by SimUI's data-gathering tool. It detects differences in the sequences from different users by repeating selection of messages based on messages type and matching of the selected message sequences. When multi-step matching is applied to two sequences: one performed by a possible user and the other named a model se-
sequence, it can reveal some of users' behaviors which should be noted by human observers during usability testing. The model sequence is made by recording operations performed by a skilled user or a user interface designer, who knows how to complete a given task efficiently.

**TOOL ORGANIZATION**

SimUI consists of:
- A recording tool
- A module modification tool
- A playback and data-gathering tool
- An analyzer.

The relationship of these tools is shown in Figure 1.

First, a user's operation sequence for a test application is recorded (Step 1). Next, the modules of the test application are modified so that the data-gathering tool can be called when the application requests the services of the OS/2 Presentation Manager, such as creation of a window. Then, the recorded data are supplied to the modified version of the application and the application's behavior is played back (Step 2). During playback, more data are recorded, so that the physical information can be translated into higher-level information at a later stage. The analyzer detects the differences between two sets of recorded data using multi-step matching (Step 3).

Finally, the detected differences are shown in an animation synchronized with playback of the recorded operations. Figure 2 shows an example of this animation. The colors of three circles at the top right change according to the degree of differences while users operations are being played back in the application window at the bottom left. The hands of three meters below the circles show the speeds of the user's operations relative to those of the model sequence.

**Message-Based Mechanism**

SimUI tools are implemented under the OS/2 Presentation Manager and we will briefly explain its mechanism for process communication before we describe SimUI tools in more detail.

The OS/2 Presentation Manager is based on a message-passing model [5]. Since all applications and the system run independently under OS/2, they communicate with each other by using messages. Messages can be generated by an input device, such as a keyboard or a mouse, or they can originate within the system as means of managing resources among processes.

A message is used to send information or to request that an action be carried out by a receiving process. A queue is a storage area used to hold messages. After a sending process has put a message into a queue, the receiving process can retrieve and process it at an appropriate time controlled by the scheduling mechanism. There is one special queue, called a system queue, that collects hardware input. OS/2 copies this input into an appropriate message queue, which is created by an application process. For example, when a key is pressed or released, the Presentation Manager decodes the key and stores the information about the keystroke in the system queue. This keyboard message is later routed to the message queue of the window with the input focus and then retrieved by an application process.

Hooks are functions, provided by OS/2, to allow applications to monitor and modify the message stream in OS/2. They can be installed in the system queue or in an application queue, depending on the types of
message to be monitored or modified. In other platforms, such as X Windows, the term 'events' is often used for messages. The role of events is similar to that of messages. It is therefore possible to implement mechanisms similar to hooks in other platforms.

**Recording**

Recording is done by a hook installed in the system queue. The hook records only the messages generated by the keyboard and the mouse. Each time the hook receives a message from the system queue, it tells a shared process, created by the recording tool, to record the message in binary form in a file. We call this file a journal file. Each message stored in the journal file contains the following information:

- Window handle
- Message identifier
- Two message parameters
- Message time
- Mouse position.

The window handle is an identifier of the window for which the message is intended. The message identifier is an integer value that indicates the type of message. The message parameters are the data to be used in processing the message. The meaning and type of the message parameters depend on the message identifier. The message time contains the time at which the message is created. The mouse position is the location of the mouse pointer when the message was created.

**Modifying the Application**

Before the recorded data is played back, the application module is modified to make it call the data-gathering tool when a new window is created. This modification is implemented by replacing the names of dynamic libraries with those of the data-gathering tool's libraries.

We chose to modify the application codes rather than the window system because we need to know only the target application in order to record its status. If we had modified the window system, processes of other applications would also generate redundant information.

**Playback**

The playback tool creates an independent process, which reads a record of a message from the journal file and places it in the system queue (Figure 3). When a message is read, a pause time is calculated by extracting the message time of the previous message from that of the current message. The playback process waits at least as long as the pause time before placing the message in the system queue. The synchronization of the playback process and other processes is maintained naturally, because the playback process waits with the same priority as other processes. For example, when the system load becomes heavy, the playback process slows down.

**Data Gathering**

During playback, data is gathered by the data-gathering tool, which is called from the application modules and the playback routines. Unlike the recording tool, which records only the messages in the system queue, the data-gathering tool records the following information about an application process:

- Identifiers (window handles) of newly created windows, and their attribute information
- Messages posted to the application process via the application message queue
- Messages sent directly to the application process.

The data gathered at this stage are stored in textual form, of which a sample is shown in Listing 1. In the gathered data, keywords start with lowercase letters, and identifiers start with uppercase letters. A text string // starts a comment, which continues to the end of the line.

Lines with the keyword send are generated when a system queue receives a message from a playback process. Lines with the keyword receive are generated when a message queue of the application receives a message. Identifiers with WH are textual representations of message identifiers, which indicate the type of messages.

When a new window is created, a line with the keyword map is generated to record the relationship between the window's attributes and its physical address; this address is known as its window handle. Since a standard window consists of several child windows, several lines with map are also generated to record the mapping between their physical addresses and their attributes. Identifiers with HWBD and with numbers, such as HWBD5000:3A04, are generated from the physical address of the window resource.
Analyzer

Although the analyzer can collect some standard statistics, such as the number of keystrokes, its unique function is automatic generation of the difference between two sets of data by means of multi-step matching.

Before the multi-step matching, the analyzer translates some identifiers into symbolic names; Listing 2 shows a translation of Listing 1. Identifiers with physical addresses, such as HWD5000:3A04 in Listing 1, are translated into symbolic names, such as WindowHWD3. The numbers in symbolic names are assigned in order of the times at which the windows were created. Since most windows are created in a similar order, this translation allows windows created in different playback to be compared. If the order of windows does not match, the mapping table can be customized manually. After the translation, multi-step matching is performed by using an existing tool for generating differences between texts.

Such tools are often available as standard utilities of operating systems. They compare the contents of two files and report the differences between them, such as missing lines and redundant lines. We used an OS/2 version of a tool described in [2], because it has an input filter that excludes from comparison any lines that contain text strings such as message identifiers.

Multi-step matching is done by repeating the text difference generation with different filters. These contain lists of message identifiers, which correspond to the types of message to be excluded. Currently, the types of message are categorized as follows:

1. Command-Level Messages
2. Message-Level Messages

Messages at the command level are sent from a window, which is a user interface control, to its owner, which is usually a main window of an application process. They are used to report to the application process the command, rather than the button, that the user has selected. A line with the message identifier WH_COMMAND in Listing 2 shows an example of a message at the command level.

Messages at the message level are also sent from a user interface control to a main window. They are used to report which user interface control has been selected or deselected. They are often used when the application needs to be informed of which method the user has used to enter a command: hitting a key or selecting a menu item with the mouse. A line with the message identifier WH_MOUSESELECT in Listing 2 shows an example of a message at the message level.

Messages at the event level are sent from the system queue to an application process, to report information relevant to hardware devices, such as the location of the mouse pointer and the scan code of a key. A line with the message identifier WH_MOUSEMOVE in Listing 2 shows an example of a message at the event level.

This classification of messages is inspired by the layered models of human-computer interaction.
posed in several publications [4, 6]. In these models, the divisions of layers clarify the differences in the functions that the systems need to offer. For example, three commonly known layers are the physical, the syntactic, and the semantic. Our version of layers is obtained from the actual implementation of the system’s interprocess communication, but we believe that some human-computer interface models can be emulated with our layers.

Figure 4 shows three windows generated by an analyzer for interactive multi-step matching. The central window shows data on model operations and the right-hand window shows a user’s operations. The left-hand window shows the results of multi-step mapping. Its three letter columns on the left show the results of command-level, message-level, and event-level matching, from left to right. Each column contains one of the following letters:

M (Match) The two lines, the model’s and the user’s, match.

C (Change) The two lines, the model’s and the user’s, are different.

D (Delete) The line in the model operations is omitted from the user’s operations.

I (Insert) The user’s operation contains an extra line.

(Blank) The lines are still filtered out and the match has not yet been tested.

The two multi-digit numbers in the columns to the right of these letters show the line numbers of the data being matched, whose contents are shown in the central and left-hand windows. These lines are also colored differently according to the result of the match. This result can animated, as illustrated in Figure 2, by displaying the different colors in synchronization with the playback of the user’s recorded operations.

**EXPERIMENT**

We conducted a small-scale experiment to confirm an assumption that we made about our analysis technique, namely, that the differences between the message sequences detected by multi-step matching bear some relationship to the differences noticed by human observers.

First, as a small test application, we recorded three operation sequences of a simple task: to change the colors of the application’s window by selecting a color name from a pull-down menu. The first operation sequence, named Data A, was recorded as a model sequence. It efficiently performs the given task, changing the colors three times. The second operation sequence, Data B, performs the same task, but the operation has some pauses because the user hesitates. The third operation sequence, Data C, performs a similar task, but the order of the selected colors is different from that in the other two sequences. The last operation sequence, Data D, also performs a similar task, but extra commands such as ‘help’ are used several times. The matching rates among the four data sets are shown in Table 1.

Next, three subjects were shown the playback of five sequences: Data A as a model sequence, and Data

<table>
<thead>
<tr>
<th>Table 1: Data Used in the Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Matching Rate (%)</strong></td>
</tr>
<tr>
<td>Command Level</td>
</tr>
<tr>
<td>Message Level</td>
</tr>
<tr>
<td>Event Level</td>
</tr>
</tbody>
</table>
Some subjects noticed differences at the message level, such as redundant selections of user interface controls (Data B).

- None found any differences in the playbacks of the same recording (Data A).

CONCLUSIONS

By developing SimUI tools, we confirmed experimentally that usability evaluation can be partly automated by software tools like SimUI.

Data gathering in playback mode, one of the key techniques in SimUI, was found to be an effective means of ensuring the reproducibility of users' operations on graphical user interfaces. SimUI's use of the playback mode takes advantage of multiprocess communication and eliminates the need for video recording hardware.

SimUI's multi-step matching detected differences in users' operations, that were sometimes overlooked by human observers. The results of the small-scale experiment were encouraging as regards the relationship between the differences detected by SimUI and those observed by human evaluators, although SimUI tends to detect more differences, some of which are not related to usability problems.
FUTURE WORK AND DISCUSSION

When multimedia environments become easily available, it will be possible to enhance SimUI's recording tool in several respects. One possibility is to synchronize the system with audio-visual equipment so as to record the user's facial expressions and comments. Another is to have eye-camera recording to observe eye-hand coordination.

Our immediate interest, however, is to give more specific information about users' operations to the people who design user interfaces. We would like to provide them with information such as “The location of this button is causing some problems” and “The command sequence for this task causes more errors than those for other tasks.” To achieve this, we plan to conduct various types of experiment and to enhance the analyzer by using algorithmic and heuristic approaches.

References


Listing 1: Sample Text Generated by Data Gathering Tool

map HWND5000:3A04 to WindowHWND3 id 32776
map HWND4FB0:3928 to MenuHWND1 in $(HWND4F90:3808)
map HWND4FA0:37A4 to MenuHWND2 in $(HWND4F90:3808)
map HWND4FF0:3974 to MenuHWND3 in $(HWND4F90:3808)

receive WM_UPDATEFRAME by $(HWND5000:3A04)
receive WM_FORMATFRAME by $(HWND5000:3A04)

send WM_MOUSEMOVE from SysQ x 220 y 198//time 3229343
receive WM_BUTTON1DOWN by $(HWND5000:3E2C) x 46 y 13
send WM_BUTTON1UP from SysQ x 220 y 198//time 3229531
receive WM_MOUSEMOVE by $(HWND5000:3E2C) x 45 y 12
send WM_MOUSEMOVE from SysQ x 220 y 198//time 3229625
receive WM_BUTTON1UP by $(HWND5000:3E2C) x 45 y 12
receive WM_MENUSELECT by $(HWND5000:3A04) id -1
receive WM_MENUEND by $(HWND5000:3A04)
receive WM_COMMAND by $(HWND4F90:3808) id 262

Listing 2: Sample Text Translated by Analyzer

map HWND5000:3A04 to WindowHWND3 id 32776
map HWND4FB0:3928 to MenuHWND1 in WindowHWND2
map HWND4FA0:37A4 to MenuHWND2 in WindowHWND2
map HWND4FF0:3974 to MenuHWND3 in WindowHWND2

receive WM_UPDATEFRAME by WindowHWND3
receive WM_FORMATFRAME by WindowHWND3

send WM_MOUSEMOVE from SysQ x 220 y 198//time 3229343
receive WM_BUTTON1DOWN by WindowHWND6 x 46 y 13
send WM_BUTTON1UP from SysQ x 220 y 198//time 3229531
receive WM_MOUSEMOVE by WindowHWND6 x 45 y 12
send WM_MOUSEMOVE from SysQ x 220 y 198//time 3229625
receive WM_BUTTON1UP by WindowHWND6 x 45 y 12
receive WM_MENUSELECT by WindowHWND3 id 262
receive WM_MENUSELECT by WindowHWND3 id -1
receive WM_MENUEND by WindowHWND3
receive WM_COMMAND by WindowHWND2 id 262