Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2013-IJ-008

2013-IJ-008

Design and Implementation of a Toolkit for Usability Testing of Mobile Apps

Xiaoxiao Ma, Bo Yan, Guanling Chen, Chunhui Zhang, Ke Huang, Jill Drury, Linzhang Wang

Mobile Networks and Applications 2013

prohibited.

Mobile Netw Appl (2013) 18:81–97
DOI 10.1007/s11036-012-0421-z

Design and Implementation of a Toolkit for Usability Testing of Mobile Apps

Xiaoxiao Ma · Bo Yan · Guanling Chen · Chunhui Zhang · Ke Huang · Jill Drury · Linzhang Wang

Published online: 21 November 2012

Abstract The usability of mobile applications is critical for their adoption because of the relatively small screen and awkward (sometimes virtual) keyboard, despite the recent advances of smartphones. Traditional laboratory-based usability testing is often tedious, expensive, and does not reflect real use cases. In this paper, we propose a toolkit that embeds into mobile applications the ability to automatically collect user interface (UI) events as the user interacts with the applications. The events are fine-grained and useful for quantified usability analysis. We have implemented the toolkit on Android devices and we evaluated the toolkit with a real deployed Android application by comparing event analysis (state-machine based) with traditional laboratory testing (expert based). The results show that our toolkit is effective at capturing detailed UI events for accurate usability analysis.

Keywords Toolkit · Usability testing · Mobile application · Automated · Logging method

1 Introduction

Led by the rapid growth of the smartphone market, mobile Internet usage in the US is expected to reach 50 % total usage by 2013 [1]. The usability of the mobile applications, however, remains a thorny issue. A recent study shows that the task completion rate using mobile Web ranges from 38 to 75 % on different phones [2]. The average success rate was only 59 %, substantially lower than the roughly 80 % success rate when testing websites on a regular PC today. Another study shows that 73 % of users experienced the slow-to-load problem when using the mobile Web, and 48 % of users found mobile Web applications difficult to read and use [3].

In this paper, we focus on the usability testing of mobile applications, particularly native (instead of Web based) applications. We envision a system that can automatically collect user interface (UI) events as the user interacts with the mobile application. The collected UI data will then be uploaded to a remote server for either automated or manual usability analysis. This kind of system can complement traditional laboratory testing, and we believe it will be particularly useful to deploy for field-based usability testing. For many mobile application developers, it is often too costly to conduct extensive laboratory-based usability testing and we anticipate that the system described in this paper will be an indispensable toolkit for low-cost usability analysis. We have implemented an Android-based automatic usability toolkit. To use our usability testing system, the Android developer needs to modify the application source code by inserting statements calling our library, which captures UI events and uploads them to a central server. We have designed the system to minimize the amount of required code modification and the impact of event-uploading overhead. To evaluate this system, we conducted a traditional laboratory-based usability test on a home-built Android application, and compared it with state-machine based usability analysis using collected UI events. The results show that our usability toolkit can effectively

X. Ma (✉) · B. Yan · G. Chen · C. Zhang · K. Huang · J. Drury
Computer Science Department,
University of Massachusetts Lowell,
1 University Avenue, Lowell, MA 01854, USA
e-mail: xma@cs.uml.edu

G. Chen · L. Wang
State Key Laboratory for Novel Software Technology,
Department of Computer Science and Technology,
Nanjing University, Nanjing, 210046,
People's Republic of China

capture most of the usability problems, some of which were not even discovered by traditional laboratory testing.

In the rest of this paper, we first discuss related work in Section 2. Section 3 describes the details of the design and implementation of our toolkit. We introduce our automatic metric extraction model in Section 4. Then we discuss the user study and present the usability analysis results in Sections 5 and 6, respectively. Last we talk about some potential issues in Section 7 and conclude in Section 8.

2 Related work

Many studies have been done with event logging methods, which are compared to traditional laboratory testing methods in terms of usability problems identified. Tullis et al. [4] presented results that showed high correlations between laboratory and remote tests for task completion data and time-on-task data. The most critical usability issues with web sites were identified by both techniques, although each technique also uniquely uncovered other issues [5]. Another study by West and Lehman [6] was conducted to evaluate a method for usability testing with an automated data collection system. They found it to be an effective alternative to a laboratory-based test [5], but these studies were conducted on desktop machines instead of mobile devices.

Waterson et al. [7] conducted a remote usability study on mobile devices. They asked participants to find some information on a web site with wireless Internet-enabled digital assistants. Half of the participants ran the test in a traditional laboratory set-up while the other half performed the task with an observer present, but with an event logging tool to collect clickstream data remotely. They revealed that the event logging and analysis tool can easily gather many of the content-related usability issues, but had difficulty in capturing device-related usability problems. However, their study focused on the mobile websites rather than the mobile applications.

There have been few usability tools developed especially for mobile applications. Flurry Analytics[1] was developed to provide accurate, real time data to developers about how consumers use their mobile applications, as well as how applications are performing across different handsets. Application developers receive anonymous, aggregated usage and performance data, as well as the use of robust reporting and analysis tools. However, this tool focuses on statistical information instead of identifying usability problems.

[1] http://www.flurry.com/

3 Event logging system

In this section, we first provide an overview of the Android UI framework, which forms the foundation for our event logging system. Then we discuss the details of its implementation and how it can be integrated with Android applications.

3.1 Android UI framework

To set up the event logging system and integrate it with developers' applications, we need to have a comprehensive understanding of Android system's UI components, as well as how these components communicate with users' interaction. So here we give a brief introduction of this knowledge. The user interface of an Android application consists of *Activity* classes (terms with italic font indicates they are classes of Android Library; we use this convention throughout this paper, unless specially stated). Each screen in an Android application is a Java class that extends the Activity class, and activities use *Views* to form graphical user interfaces that display information and respond to user actions. An activity itself can be considered to be the root View for a screen and it is composed of smaller Views, which are interaction components like controls or widgets. Also, there might be other components attached to an activity, such as *Dialogs* or *Menus*, which are usually small windows that may appear in front of an activity. The top-most window in the foreground (View, Dialog or Menu) will always intercept and handle user inputs, whether it occupies a full or only partial screen.

To allow an application to react to user events such as a key press, screen touch and button click, developers have to utilize either event handlers or event listeners to accept user inputs and respond to them. An event handler is usually implemented by overriding some callback function while an event listener is a programming interface that contains a single callback method. As both event handlers and event listeners are constructed through calling underlying Android callback functions and they have the same functionality, for convenience we will use the term "Event Listener" to stand for both of them in the rest of this paper.

Usually an Event Listener is only attached to the window that registers it, and it will "consume" the events captured by it. This means if a button registers an *onClickListener* and then is clicked, the Event Listener of the button would intercept and handle the click, while the screen activity that owns the button has no idea that there was ever a click. Hence, if we want to log every single movement of users' interaction, for each Event Listener that handles an user input, we need to trace into the innermost level of the window that possesses it, and acquire our desired information that is intercepted by it.

3.2 System overview

For usability studies of websites, it is possible to build an instrumented Web browser which allows the users' interactions with the websites to be automatically logged in a fashion that is transparent to the website developer. On the other hand, this is not feasible for Android applications because the UI interactions cannot be automatically captured. Application developers must get involved in modifying the source code and capturing UI events explicitly. To minimize developers' effort, we provide an event logging toolkit that takes care of most of the work of event capturing, formatting and packing, and transmission.

Our event logging system works as follows. The developers make small modifications to the source code of their applications adding API calls and recompiling the source code with the Software Development Kit (SDK) we provide. The SDK contains the APIs for each Event Listener and the developers call the corresponding API in their own listener code. The functionality of the Application Programming Interface (API) is to log ongoing user interaction events, its timestamp and properties of the relevant windows. For example, by inserting one statement in a View's *onClickListener*, the library can retrieve information such as the View's identifier, size, owner activity, and so on.

The recompiled applications now can automatically record the users' UI events, and transmit the captured interaction data periodically to a central server. These events are then used for usability analysis by the evaluators. Instead of transmitting the events immediately to the remote server, the logger runs in the background as a service and puts the captured events in a memory queue. When the number of events accumulates to a predetermined amount, they are transferred to our remote server through the 3G or WiFi network. If there is no available network at the time of transmission, these events will be stored on the device's Secure Digital(SD) card or hard disk sequentially according to their availability. In every event uploading cycle, these two places will be examined, and all existing data stored there will be transmitted if the network allows. The event transmitter creates a new thread for the transporting module so that this process is separated from the UI process.

The remote receiver module simply provides a relational database, as all the UI events are saved into different tables depending on their type. Usability analysis can then be conducted either manually or automatically (see Section 6).

3.3 The logger implementation

Android has many Event Listeners, and one listener can be attached to a View, a Dialog or a Menu. As Views, Dialogs and Menus have different appearances and functionalities, they have different sets of listeners. For instance, clicking on a Menu may trigger *onMenuItemClickListener* while clicking on a View may trigger *onViewClickListener*, though both of the interactions are click events. Moreover, even if two subclasses inherit from the same parent, they may not have the same sets of listeners. Take View for example, the subclass *AbsListView* nests the interface component *onScrollListener*, which will handle the user's scrolling on this view. In comparison, most subclasses of View do not support scroll events (they discard these events) if they do not implement *onScrollListener*. Indeed, the View class hierarchy is quite complex, as it has eight direct subclasses and dozens of indirect subclasses. To assure we log user interaction events as completely as possible, we performed a thorough survey of the class hierarchy of View, Dialog and Menu, and only extracted those listeners that are related to user interaction. We then consolidated the listeners in the following ways.

View events Some listeners differ in their names or adhering classes, but they take care of the same user interaction, e.g. *onOptionItemSelectedListener*, *onMenuItemClickListener* and *onMenuItemSelectedListener* may all be responsible for selecting one item in an Option Menu. More interestingly, the three listeners can be registered at the same time, which means that they can all be triggered upon one click. Since we are interested in the user's interaction type rather than listeners' name, we decided to combine these sorts of similar listeners into one event type to avoid redundancy. Some listeners were distinguished by the type of user event they can handle but take identical parameters, such as the View's *onClickListener* and *onLongClickListener*, which deal with View's short click and long click events, respectively (Android detects the temporal length of a press, if the press is longer than a certain amount of time, it's a long-click event, otherwise, it's a click event). For the simplicity of the logging library and back-end database design, we treated them as the same events as well, but differentiated them by adding a flag parameter. Thus the View's events were consolidated into *view click*, *view key press*, *view scroll* and *AdapterView item click* (note the terms for event types are also italicized). An *AdapterView* is a View whose children are determined by an *Adapter* object acting as a bridge between the *AdapterView* and its underlying data, e.g. an *ArrayAdapter* allows an array of objects to reside in the *AdapterView*.

Dialog and Menu Events triggered by Dialog and Menu are harder to capture. Dialogs and Menus are usually managed by their adhering activities, and they may not have an identifier (anonymous). Thus in order to locate which Dialog or Menu was fired, we need to infer it from the users' action sequence, by looking at which activity, through

which action a Dialog or Menu, was initiated and how a Dialog or Menu was dismissed. It is not difficult to record the initiation event, but some attention needs to be paid to the dismiss event. Dialog has an *onDismissListener* but we cannot rely on it because it does not provide any information about how a Dialog was dismissed. Even worse, no matter how a dialog disappeared, this listener would always be triggered. For instance, if a user presses the OK button in a dialog, Android will first call Dialog's *onClickListener*, and then call *onDismissListener* as the dialog will disappear after the action. This will cause a double counting problem because the single event fires two listeners. Fortunately, we found that if a dialog was not dismissed by hitting the BACK key on the hard keyboard, at least one of the Dialog's *onClickListener* methods will be called, and it tells us which button was clicked that caused the dialog to be dismissed.

We have a different solution for Menu's dismiss event. An *onOptionMenuClosed* or *onContextMenuClosed* method will be toggled when a menu is closed, depending on the menu type. We can monitor *onMenuItemSelectedListener* to judge whether a menu was closed by selecting an item or by other means. Overall, we included *dialog key press* and *dialog button clicked* events for Dialog and *menu item select* and *menu close* events for Menu to our logging library.

System keys Android devices have BACK, MENU, HOME and SEARCH hard keys, and we name them system keys collectively as these keys function for all applications in general. Since an Android application is composed of activities, we can override *onKeyDown* listener in each activity to intercept these system keys. However, Android disabled HOME key interception to preserve its system's running state. If the developers can intercept all key events, they can easily make the system malfunction, such as preventing users from quitting their applications by intercepting the HOME key press. Thus, we have to find other ways to detect that a user clicked the HOME key. Through a class named *ActivityManager*, we can acquire the currently running tasks as well as the top activity of each task. Then we can override each activity's *onStop* method to check if its top activity equals to *com.android.launcher*, which is the activity name of the Android home screen, to decide if the activity is stopped by clicking the HOME button.

Unhandled events We believe that the above effort can already help us to record those events whose listeners were registered by the developers. This, however, is not enough. We want to collect a comprehensive set of the user's interaction behaviors, but the developers will most likely only register listeners in which they are interested. Suppose a developer does not register an *onKeyDownListener* method in one activity, and a user tried to click a button that belongs to this activity. If, for some reason, this user missed clicking on the button but happened to click on the activity itself (recall an activity is also a view), the application will still run well because that activity will discard this mis-click by default. The developers may not care about this kind of event, but these events can be important to discover usability issues. For example, if we detected many clicks around the button, we can infer that the button is hard to click for some reason. Thus we would like to capture such events as much as possible, and we name them unhandled events in general. Please be advised that orientation change is not one of unhandled events, as in most cases developers will let Android system deal with it.

In summary, the events we captured were classified by their adhering class and are listed in the Table 1. For different events, we obtained different attributes according to their own available properties. From an event that occurred in a View, we can retrieve its ID, width, height and its parent class etc., while for events that happened in a menu or dialog, they may not have such information. But for whatever window, we tried to retrieve as many event attributes as possible.

3.4 Code revision

The extent of the revision that needs to be done in the developers' code greatly depends on the hierarchical organization of the source code. Thus here we only discuss the best and the worst cases in terms of the code modification workload. In the best case, all activities in an application extend from a single root activity, so the developer just needs to insert one event recording statement (by calling our API) into the *onTouchEvent*, *onKeyUp*, *onStop* methods of that root activity. Meanwhile, the event listeners were implemented uniformly rather than redefined in each View/Dialog/Menu, so that only one recording statement in the implemented View/Dialog/Menu event listeners needs to be inserted.

In the worst case, the activities in the application have no hierarchical structure at all and the event listeners were implemented separately from each View/Dialog/Menu. Then the developers have to insert the recording statement into the *onTouchEvent*, *onKeyUp*, *onStop* methods in each of

Table 1 Event type summary

	Event type
View	Click, key press, adapter item click, scroll
Dialog	Key press, button click
Menu	Item select, close
Other	Unhandled motion event, unhandled key event, home key click, system key click, preference click

their activities, and insert the recording statement into each View/Dialog/Menu Event Listener.

In both cases, we require that the application classes extend the application class of our own library so that our library can make use of the static memory space allocated to the application.

There are two additional challenges. First, the Android framework has already implemented some event listeners by default, such as the *onKeyDownListener* for an edit box. In this way, an edit box can accept key presses even without developers registering this listener explicitly. For such cases, we have to override the related listeners and register them so as not to miss recording the user's input. Second, the hierarchical relationship between classes can be troublesome for counting the user's interaction events accurately. While usually the user's interaction in one window will not be passed to its parent, the developers can call the super method to allow this to happen. If we add the logging function in both super and subclasses, one event that happened in the subclass may be recorded twice. Thus to avoid this double counting problem, we have to examine the application's class hierarchy and check whether at some point the developers called the super method. We considered using timestamps to verify if multiple events are triggered by a single user interaction, but function calls among these hierarchical listeners are sequential, thus the timestamps can vary when an event is passed down from one listener to another, and it's hard to define boundaries to tell when different events can be regarded as one.

In the future, we plan to provide a tool that can automatically inspect an application's source code and make appropriate changes for event logging, without the developers' involvement. We could do this by first representing a source code file as an *Abstract Syntax Tree*, and searching for functions that correspond to event listeners (as we know the name of each event listener), then extracting parameter names from these functions and finally inserting logging statements into these functions with extracted parameters.

3.5 Events not captured

So far, our logging system is able to capture all the events that the developers have already set to listen as well as those unhandled events at the activity level. But there still remain some events that are not being captured. First, we cannot log events that occurred in Android native activities, such as the Android Settings. We do not, however, really need these events as the goal of our system is to identify usability issues of the third-party applications, rather than the Android native screens. Second, we have not found any feasible way to trace keystrokes on the Android soft keyboard. Finally, we did not capture the unhandled events occurring in child Views, Dialogs and Menus of an activity. Although this is doable by registering listeners in each View/Dialog/Menu, we do not think it deserves so much effort (which involves changing source code) compared to its usefulness and potential event logging overhead.

Despite missing these events, the current system can already capture a comprehensive set of interaction events that can be used for usability analysis, as demonstrated in the next section.

4 Automated metric discovery model

Most of previous work on automated usability testing focused on task completion rate and timing analysis, few went deep into fine-grained usability problem identification or quantitative measurements. Thanks to the capability of the logging toolkit which can help collect a detailed set of user interaction events, we are able to build an automata-based model that leverages these events to perform usability analysis automatically. The core idea proposed here is the sequence comparison technique, which was described by Hilbert and Redmiles as the process of comparing a target sequence against the source sequences and producing measures of correspondence [8]. For each user task, we ask the developer to provide an *expert* sequence as the target sequence, and we collect a set of non-expert sequences from the testing users as the source sequences. We then examine how these source sequences deviate from the target sequence, and extract some quantitative metrics as the indicators of the usability problems.

In order to facilitate the specification of the expert sequence, as well as help distributing the tasks to the users, we provide developers an Android application to manage the user study tasks. This application allows the developers to name a task, describe it and set up its expert sequence. Meanwhile, the developers can configure the preference about how frequently to assign these tasks to the users. We generate a unique identifier for each task, so that when it is assigned to the users, we know which task a user takes. The expert sequence and the non-expert sequences are all transmitted and stored at our back-end server, so we can construct an automata to compare them, by extracting usability related metrics as its measurement. We need two phases to achieve this, as explained in the next two subsections.

4.1 Creating baseline state machine

We assume that there is only one best way to complete a given task, thus each task binds with a unique expert sequence provided by the developer (alternative expert sequence may exist, and we plan to allow the developers to provide multiple expert sequences for a task in the

future. Right now if we find users complete a task through a sequence other than the expert one, we will report this instance to the developer and let him/her decide whether it can be concluded to a usability problem). We map these expert sequences to one automata, as we think the automata can concisely and accurately represent each step in the expert sequence and the flows (order) between the steps. We call the automata of the expert sequence the *baseline state machine*, as this automata only involves indispensable steps to complete a task. Similar to a standard nondeterministic finite automata (NFA), the baseline state machine consists of five components.

- A finite set of states
  In the first version of our toolkit we only used activity name as the identifier of a state, but this will mix up the scenarios of whether there are menu or dialog pop-ups, as usually activity name wouldn't change when a menu or a dialog is popped up. Hence we added menu and dialog flags as two additional fields, which indicate various status of menu and dialog. For menu, the flag is set to one of None, Option, Context and Popup (these are currently available menu types); for dialog, the flag is set to an integer value that is defined by the developer. Two states are deemed as identical only if their activity name, menu and dialog flag values are the same. In order to sense the status change of the menu and the dialog, we inserted some more logging statements into the developer's code. The current toolkit receives an event every time a menu and dialog is shown or closed.
- A finite set of input symbols
  Here the user's interaction events are deemed as input symbols, since it is the user's interactions that cause the baseline state machine to transit from one state to another. As introduced in previous section, each interaction event consists of different fields, and we combine these fields all together as the identifier of an input symbol, since every field is an indispensable contributor to the input symbol. For instance, if a user clicks an item in a Preference Activity, the item's title and order need to be considered as part of the input symbol, as a different title or order implies that the user clicked on a different item.
- Transition functions
  Transition functions are a set of rules that regulate conversions between the states. The input of a transition function is a two dimensional tuple that contains the current state and an input symbol, which we call the precondition. The output of the transition function tells which state will the be reached after this transition, given the precondition. As this automata is an NFA, the set of transition functions only define the available flows between the states. If there are no transitions between two states, no flow connects each other. For simplicity, we assign each state with an integer value as its index. We assume that under the same precondition, its outcome should be decidable. Thus if we find two cases with exactly the same preconditions but return different results, we will raise an exception.
- An initial state
  The initial state is the point where an automata starts with handling input symbols, usually it is a state without particular meanings. As in our case, each state binds with an activity name, thus the initial state's activity name should be the name of home activity of the given task. Currently when we ask the developers to create an expert sequence, we will direct them to the launcher activity of their applications, and by default, both menu and dialog are closed. So for every task of an application, the initial state would always be the launcher activity of the application, with both menu and dialog flags set to be none.
- A set of final state
  A regular NFA might have more than one final states, but it is different here. We assume that as long as a task is successfully completed, the same state (the same activity and the same menu/dialog flags) would be reached, even though the intermediate states may not be the same. Hence we only have a single final state for each task.

We call the states in the baseline state machine *baseline states*, and the transition functions *baseline transitions*. The procedure of creating baseline state machine is as follows. At the beginning, we initialize all components of the state machine with empty sets, and set both menu flag and dialog flags to be none. Then we iteratively read events from the expert sequence. We have two types of events, one type can only cause the value of menu flag or dialog flag to change. In this case, we update the corresponding flag value and continue to read the next event. In the other case, we create a baseline transition. As for any two consecutive events, the former one's state index and input symbol compose the precondition of a transition function, while the latter one's state index serves as the outcome of that transition function, so the key point here is to figure out what is the state index for each event. We maintain a baseline state set, and look up whether the state of an event belongs to this set. If it does not, we add it to the state set and assign it with a new integer value. After handling all events in the expert sequence, we obtain the set of states, input symbols and transition functions of this baseline state machine. Additionally, we set the initial state's activity name to be the first event's activity name, and we assume after processing the last event, the final state is reached.

4.2 Running user sequence

The user's interaction sequences are expected to have some divergence from the expert sequence, as the users make mistakes. The users may go to some states that are out of the state set of baseline state machine, and these states are called mistaken states. Apparently, all transition functions associated with mistaken states (either as input or output) are excluded from the baseline state machine. In addition, the user's interaction sequences might contain transitions between baseline states which are not accessible in the baseline state machine (the simplest example is backtracking between baseline states). When an NFA sees an unfamiliar precondition, it will halt by default. This is not what we want, because we will not get any useful metrics for analysis. We need to keep the automata running till it processes all events in the user sequences. Thus, we enhance the baseline state machine with self-learning ability. If the baseline state machine encounters a transition that it is familiar with, it follows the transition function as usual and transits to the result state of the transition function. However, if the baseline state machine sees a transition which it never meets before, it creates a new transition and finds its own way to continue. We will explain how it works in detail in the following section.

To be able to extract usability related metrics while handling the user sequences, the baseline state machine must be precisely aware of the user's position towards the final state at every moment. On one hand, if the user is situating on a baseline state, the baseline state machine needs to know what is the expected transition that leads to the next baseline state. On the other hand, if the user is deviating from the expert sequence, the baseline state machine needs to tell how far the current mistaken state is away from the baseline transition. Here we first introduce several auxiliary variables which can be obtained by the baseline state machine.

- Precondition queue of each state
  By verifying whether a transition is a baseline transition, we do not only check whether this transition's result state is correct, but also examine whether its precondition is the same with the baseline transition. If the transition's precondition is different than the baseline transition's, we do not consider this transition as the same as expert sequence even if it leads to the expected state. Thus for each baseline state, we need to remember through what precondition it is directed to. Moreover, a baseline state might be visited more than once in the expert sequence. To distinguish between visits, we maintain a precondition queue for each state, indicating the corresponding precondition of each visit.
- Regressive pointer and non-regressive pointer
  We introduce the two types of pointers to help understand where a user is situated in a baseline state machine at each move. The Non-regressive pointer, as implied by its literal meaning, will never retrograde. We can see from Fig. 1, when the user moves from state q_2 to q_1, the non-regressive pointer stays pointing to state q_3, because we use it to indicate the state reached by the user that is closest to the final state at present. On the other hand, regressive pointer implies the very next baseline state that the user should try to move to from the present state. For instance, when the user is at state q_2, the regressive pointer points to state q_3, as the next baseline state is q_3. And when the user moves back from state q_2 to q_1, the regressive pointer also falls back from q_3 to q_2, since if the user wants to reach the final state from state q_1, q_2 is the very next state he/she has to go through. Both the regressive and non-regressive pointer are useful in calculating usability related metrics, and we will talk about those metrics later in this section.
- State stack
  Ideally a user can not bypass any baseline state in the transition process of a task, but an application's design or implementation may have flaws, plus the user's behaviors are not predictable, thus the ideal case might not always be true. We deem such scenarios as exceptions because it disobeys the principles of the baseline automata. We cannot eliminate these exceptions but we can detect them. Therefore we employ a state stack to trace the user's interaction sequence. If there are any unexpected situations (bypassing is only one type) in the user sequence, we will report it to the developer and

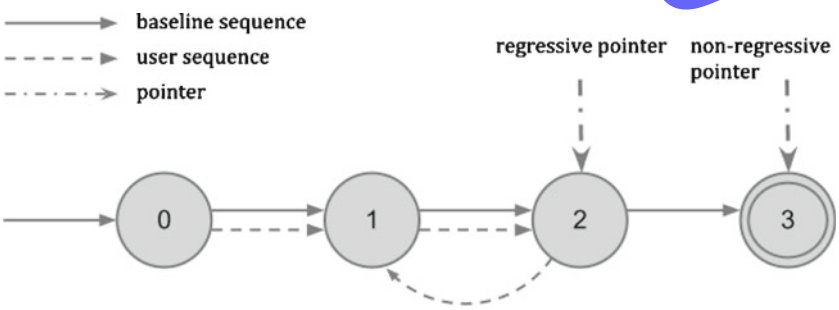


Fig. 1 Regressive pointer and non-regressive pointer

let the developer decide whether it's a problem. The mechanism here is to push the current state into the *state stack* if a move is not a backtracking, but pop out the state at the top of the stack if a move is. In the end, we compare this state stack with the expert sequence. If there are no exceptions, the state stack would be the same as the expert sequence. Otherwise, the state stack will have at least one state less or one state more than the expert sequence, and these two situations are considered as *bypass exception* and *redundant exception*, respectively.

With the help of these variables, we categorize transitions into the following status, based on the incoming state index and the outgoing state index of a transition.

- Hit
  The user is following precisely the expert sequence and is moving to the next unreached baseline state, both non-regressive pointer and regressive pointer will move forward.
- Correct
  The user is following a baseline transition, but this transition does not help the non-regressive pointer to move forward. This is either because the precondition of this transition is not expected or the user has went through this transition before.
- Setback
  The user is moving from a baseline state to a previous baseline state. In this case, the regressive pointer must be relocated.
- Deviating
  The user is traversing to a mistaken state from a baseline state.
- Returning
  The user is returning to a baseline state from a mistaken state.
- Roaming
  The user is traversing between mistaken states.
- Mistaken
  Either the transition function or the outgoing state does not exist. The baseline state machine will create the transition function or the state, mark it as mistaken and add it to its corresponding set.

According to the type of transitions, we can adjust the value of those auxiliary variables and calculate a set of metrics that are pertinent for usability analysis. These metrics are explained as follows, and we will illustrate them further in the user study section. To understand the cause of a usability problem and locate it accurately, we granulate the statistics of each metric to state level. That is, we check at which baseline state does an instance occur, and count it onto the corresponding baseline state. If the users do not situate on a baseline state, we use the baseline state that is pointed by the regressive pointer for such cases.

- Backtracking
  Backtracking is defined as returning to a state which has previously been traversed. In practice, we only count how many times *BACK* key is pressed as the backtracking number. If a user goes back to a previous state without clicking BACK key, we do not regard it as backtracking. As we observe that in some applications, there are functions that intend to direct users back to previous states. If we strictly follow the definition of backtracking as described above , we may bring in many false backtracking instances.
- Correct Flow Ratio
  This metric is defined as the ratio of the number of baseline transitions and the number of mistaken transitions in each baseline state. If the status of a transition is *Hit* or *Correct*, we deem that the user makes a right choice as the transition follows the baseline transition. We call it a *correct flow*. Otherwise, it is an *incorrect flow*. The correct flow ratio is an aggregated value across all users. For each user, we only count the number of correct flows and incorrect flows. We take a ratio of total correct flows and incorrect flows as the result of this metric. This metric, to some extent, reflects the users' confidence of making a correct choice at each baseline step.
- Most Misleading State, Misleading Coefficient
  Most misleading state is the mistaken state that attracts most of the incorrect flows, and we think this state affects the user's decision most of making a right choice. Only the states which can be reached in one hop from the current state will be considered. We ignore the portion of incorrect flows that caused by backtracking, as if a user makes a backtracking from a baseline state, s/he would go to a baseline state. Hence, we only consider the situations when the user's transition status is *Deviating*. We hold a mistaken state set and record the occurrence of each state, and after we complete processing the user sequence, we take the mistaken state that appears most as the most misleading state. The misleading coefficient is simply the ratio of the occurrence of the most misleading state and the correct flow number, it reveals how much confusion this state brings to the users.
- First Time Choice
  We are particularly concerned about whether the users can make a correct choice at the first time they are asked to choose, as we think the first time choice is not influenced by the user's prior selections on this interface,

but a pure reflection of the users' intuition. So that we regard this metric as a criteria of clarity of an interface.

- Number of Actions between States
  In the expert sequence, we use exactly one action to transit from a baseline state to another. But whenever a user makes a mistake, he will spend at least one more actions between that transition. The more mistakes a user makes, the more number of actions he consumes. Hence we use this metric to judge how many mistakes a user makes between two expert states.

In the expert state machine, we do not have an activity name for the final state, as it is a virtual state. So how could we determine whether the user has reached the final state? To make sure we catch the proceeding to the final state, we keep an eye on the preceding state of the final state. As long as a user performs the correct transition on the preceding state, we regard this user reaches the final state, and we use this to decide if a user successfully finished a task.

When we succeed in processing the user sequence, we will obtain the metrics explained above. In later sections, we will give some concrete examples to illustrate how we use these metrics to identify usability problems and how effectively they are in a practical user study.

5 User studies

We conducted a user study to evaluate whether the proposed event logging toolkit is effective and helpful in identifying usability problems. One Android application called AppJoy [9], which has been deployed to the Google Market, was used as our subject application. It was developed by our group; we have the source code so it is convenient for us to integrate it with our logging library. We recruited participants to use this application and asked them to execute certain tasks assigned by us. Afterwards, we examined the logged events for usability analysis.

5.1 AppJoy overview

The explosive growth of the mobile application market has made it a significant challenge for users to find interesting applications in crowded App Stores. While the application stores allow the users to search for applications by keywords or browse top applications in different categories, it is still difficult for the users to find interesting applications that they like. In light of this problem, existing industry solutions often use users' application download history or their ratings on some applications to predict users' preferences, such as Amazon's book recommendations. However, "downloading" is actually a weak indicator of users' fondness for an application, particularly if the application is free and users just want to try it out. Using application ratings, on the other hand, suffers from tedious manual input and potential data sparsity problems.

AppJoy makes personalized application recommendations by analyzing users' statistical usage of their installed applications. It can also allow users to browse popular applications according to their location, and track their application usage.

5.2 Participant briefing

Participants were recruited through posters. We recruited 12 participants in total, all of them were undergraduate or graduate students of our school. We asked the participants to fill in a demographic information survey before the study. The questions included gender, major, own cellphone platform, familiarity with Android platform and previous experience in usability testing, and so on.

Among the participants, 7 were from the Computer Science department while 5 were not, 7 were male and the other 5 were female. All of them were between 20 and 35 years old. 2 participants owned an Android phone, 3 participants owned an iPhone and 7 participants did not have a smartphone. One iPhone user and one non-smartphone user also used an Android phone before, so in addition to the two participants who owned an Android phone, we had 4 Android phone users, but none of them were Android developers. 4 participants had previous experience with usability tests. None of the participants had used AppJoy before.

5.3 AppJoy tasks

All participants were given the same Android device—Motorola Droid with Android version 2.2.2. AppJoy was preloaded onto the device and the device was connected to a university WiFi network. Every participant was assigned the following tasks one by one in the same order:

1. Browse recommended applications in AppJoy and "dislike" the first application the user is not interested in.
2. Browse the newest applications in AppJoy and install the first one whose rating is greater than 4 stars and the number of downloads is greater than 50,000.
3. Clear search history in AppJoy settings.
4. Search applications used by people in Boston, and point out the first application that was installed on this device.
5. Use AppJoy to uninstall the application that has been installed in task 2.
6. In AppJoy, look up the usage time of AppJoy.

As we did not code a special function to indicate the completion of a task, we used Android's SCREEN_ON

and SCREEN_OFF broadcasting events as the separator between tasks during the test. Participants were asked to give the device back to one of the evaluators after completing each task, and the evaluator turned off and turned on the screen twice and reset the application to its homepage before the next task.

6 Evaluation results

In this section we answer the question of whether the UI events collected via the Android UI framework can indeed be used for usability analysis. The laboratory-based usability testing method known as formal usability testing is one of the most widely used approaches by usability practitioners for performing usability analysis [10, 11]. Thus we performed a laboratory-based usability test and compared it to a quantified state-machine based analysis using the collected events.

When the participants were executing tasks, we asked them to "think aloud" [12] and had several evaluators sitting beside them to take notes. At the same time, all of the user's interaction events with the AppJoy were simultaneously logged and the data was transmitted to our server. In this way, we were able to get a fair comparison for the two different methods, since they were compared using the same participants, the same Android device, at the same time, within the same testing environment and the same user behaviors.

During the test, we lost two participants' data due to the misconfiguration of the event logging system. So when comparing the two methods, we only considered information collected from the remaining 10 participants. The two people we lost data from were participant 4 and 5, one of them is male and the other is female. Neither of them majored in computer science and neither of them had prior Android experience. When presenting the evaluation in this section, we do use all 12 participants' data except for the comparison results.

6.1 Traditional laboratory-based usability testing results

When the participants were executing tasks, we asked them to "think aloud" and had 3 evaluators taking notes beside them. The evaluators were all very familiar with AppJoy and one of them is the lead developer. In order to get a better understanding of wrong moves the participants made when executing specific tasks, we talked with them about the difficulties they encountered during the test, and what caused their confusion. We found that these conversations with participants were indeed valuable for us to judge the exact cause of a usability problem. After the experiment was over, the evaluators discussed and consolidated usability problems identified based on their notes, the participants' survey and their verbal feedback. Then we rated the severity of each usability problem according to Nielsen's severity rating criteria [13], and summarized them in Table 2.

Some of the problems were apparently caused by the AppJoy design, which we call AppJoy problems. Some of the other issues could not be categorized as AppJoy problems because AppJoy just leveraged some components of the Android framework that caused the user confusion. For instance, some participants did not know how to view the notifications on Android, as they tapped on the notification bar instead of dragging. Also, there was one participant who said that he/she did not know how to scroll the view on the screen, and he/she moved his/her finger in the opposite direction. For these problems, we say that they were generated by the users' unfamiliarity with some of the Android conventions.

In addition, we have two problems not included in Table 2. One problem was that the participants had trouble finding the AppJoy setting. The reason for this problem was unclear, and we could not arbitrarily say whether this was because the participants did not know that pressing the Menu button can trigger application settings as an Android convention or they did not believe that the Menu is the right place to find the AppJoy settings. Although the confusion was mostly from the Non-Android participants, one Android participant also spent a lot of effort before getting to the right place. The other problem was that the participants frequently touched the AppJoy's caption bar by tapping, dragging or scrolling. This problem is not negligible as 5 participants had this issue in 6 tasks. However, we cannot simply blame either AppJoy or Android for this as we do not see that any application or the Android framework itself defined the functionality of the caption bar. Thus we left the two problems described above uncategorized.

6.2 Event logging method result

As we described before, in this method we first identify the baseline sequence for each task, and then we examine how users' interaction sequences deviated from the baseline sequence.

First we draw the baseline state machine each task. To be concrete, we use task 1 as example. Figure 2 shows the

Table 2 Usability problems identified by laboratory testing

	AppJoy problem	Android convention
Cosmetic	5	2
Minor	4	2
Major	4	2
Catastrophe	1	0

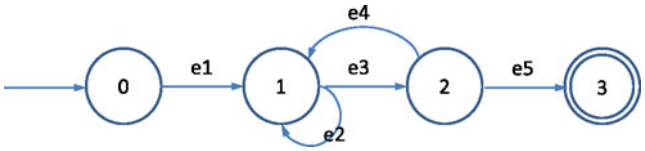


Fig. 2 Task 1 baseline state machine

baseline finite state machine for task 1, and Table 3 lists the associated activities corresponding to the states, and the user actions corresponding to the transition functions. As we can see, there are four states in this state machine with state 0 being the initial state and state 3 being the final state. Because this state machine represents the baseline sequence, we name the states in the state machine as fundamental states. Events e1, e3, e5 are the three imperative actions to complete this task, while e2 and e4, though not required, are also considered to be "correct" actions. Recall the task was asking the participants to dislike one application they were not interested in, so the participants may freely browse the applications by scrolling in the activity, or by checking the detailed information of an application and then going back. Events e2 and e4 correspond to these two actions, respectively.

Then we manually processed users' interactions sequences. We counted two kinds of behaviors that can, to some extent, indicate participants' confusion. One is backtracking (sometimes called "regressive behavior" by usability evaluators), and the other is engaging in unhandled events as we mentioned before.

We define backtracking as redundantly returning to the state that has already been traversed. We use "redundant" only to exclude situations where backward transitions are also considered to be appropriate responses, such as e4 in task 1. Basically if a user goes back to a state which he/she has visited before, that is backtracking, no matter whether he/she traces back from a baseline state or a mistake state. Additionally, if a user goes back from one activity to another, then immediately back to an even earlier state, we count this circumstance as two backtracking events rather than one. Usually backtracking reflects a user's confusion. When a backtracking event happens [14], it means that the user has picked the option that he/she thought to be most probably right, but apparently he/she did not reach the desired state via that option. For example, to find the usage time of AppJoy, many participants went to AppJoy settings first. After realizing there is no such information, they stepped back to the home page of AppJoy.

Similarly, unhandled events are user behaviors that occurred beyond the developers' expectations, since the developers did not even register listeners for those events. Although some events were triggered by the users' unintentional touches, most of these events reflected the users' intentional purpose. If a user performed a lot of such actions, we can infer that this user might not know where to navigate to the next step, as he/she was trying actions either randomly or exhaustively, hoping to hit something correct by chance or by systematically attempting to activate all plausible interface actions in turn. We list the number of instances that occurred for the above two behaviors by task in Table 4.

Simply from these numbers, we can infer that tasks 3 and 6 were the two most difficult jobs for the participants, and tasks 1 and 2 were relatively easy. However, having only these numbers is insufficient for us to analyze usability problems; thus we examined these events more closely.

As we recorded the participants' every single move from one activity to another, we represent these transitions as well as the volume of these transitions graphically, in a form of traffic flow chart. We present task 1's traffic flow chart in Fig. 3 and list its states and transition functions in Table 5. Note here we only show those states and transition functions that were not included in Table 3 (in other words, mistake states and transition functions).

We use the width of an edge to represent the volume of the transitions. Blue edges represent incoming flows, while green edges and orange edges represent outgoing flows to a baseline state or a mistake state accordingly. Additionally, we use dashed red edges to indicate backtracking flows. The volume of traverses along with each transition function is marked above each edge. Note that the number of correct outgoing flows from one baseline state and the number of incoming flows to the next baseline state have different implications. The former means how many times the participants made the right choices from one state, and the latter means how many situations occurred in which the participants were asked to make a choice. For example, if one participant went to some mistake state from state 1, and then returned back to state 1, the number of incoming flows to state 1 would be greater than the number of correct outgoing

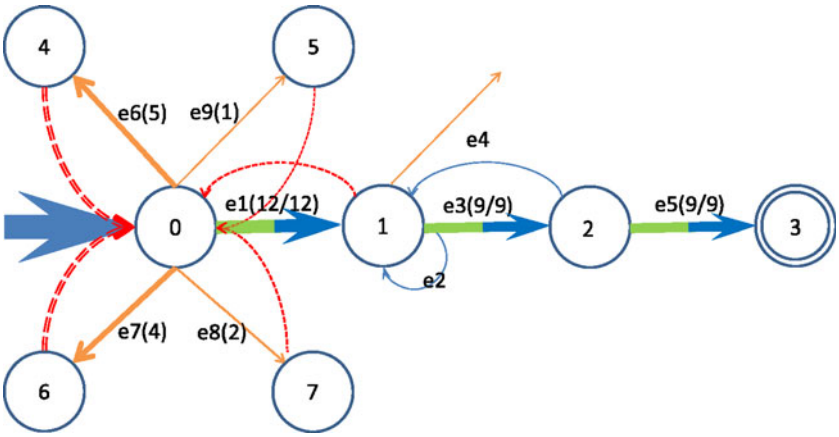
Table 3 States and transitions for baseline state machine of Task 1

	Activity		Event
a0	Home page	e1	Click *my recommendations*
a1	My recommendation	e2	Scroll in *my recommendations*
a2	Application detail	e3	Click in an application
a3	Dislike dialog	e4	Click *back* button
		e5	Click *dislike* button

Table 4 Backtracking and unhandled events in each task

	T1	T2	T3	T4	T5	T6
Backtracking	14	13	36	35	14	70
Unhandled motion	1	4	11	1	1	21
Unhandled key	7	3	4	1	30	4

Fig. 3 Task 1 traffic flow chart



flows from state 0. Thus we have two figures above every baseline transition function, with the first one representing correct outgoing flows from one state and the second one representing incoming flows to the next state. Coincidently, the two figures for all baseline transitions turn out to be equal in this task.

As shown in Table 6, the participants traversed to the home page of AppJoy 24 times, of which they went to state 1 12 times, while they went to different mistake states another 12 times. Comparatively speaking, from state 1 to state 2, among the 12 incoming flows, 9 times the participants progressed immediately to state 2, backtracking occurred two times during this stage, and one participant failed to find the next move. From state 2 to state 3, all of the 9 incoming flows traveled to the correct state. The correct flow ratios for the three stages were 50, 75 and 100 %, respectively. Clearly the users were less confused at state 2 than at state 0. We calculated the success ratio of flows in each step as a measurement for detecting usability problems, because this can be considered to be a measure of the users' confusion at each step. Also, the amount of flow that entered the final state is actually the number of participants who successfully completed a task. On the other hand, the number of participants who failed to complete a step is also an important indicator of usability problems. We calculated these metrics for each task and summarized them in Table 6.

Note that the step in which backtracking occurred is determined by the next baseline state rather than the current baseline state, as we think users' difficulties in locating the next baseline state is the main reason that causes backtracking in the current baseline state. For instance, the 12 backtracking events that happened during the transition between state 0 and state 1 were counted as backtracking events that occurred at state 1.

Many usability problems can be discovered by reviewing Table 6. Too much backtracking, such as the amount that occurred at step 1 of task 6, indicates that the desired information is located at a different place than anticipated by participants, or it was not visible to users. Low correct flow ratio, in our case less than 50 %, is another sign of potential usability problems. For instance, the ratio of correct flows for step 1 of task 3 was only 28.6 %, which is congruent with our previous discussion regarding participants having problems finding AppJoy settings. Also, if the number of participants who cannot complete a step exceeded a certain threshold, there is possibly a usability problem. For instance, 4 out of 10 participants failed to find the usage time at step 2 of task 6. Although we are unaware of the cause, we would strongly recommend that the developer of AppJoy inspect that component to show usage time clearer more clearly.

By taking a step further to examine which mistake state attracted most of the incorrect traffic, we may possibly predict the reason for that problem. If many participants went often to the same mistake state, that state must be very confusing to the participants. Hence we call that state the "most misleading state." For example, at step 3 of task 4, 18 out of the 20 incorrect flows went to the detailed page of location-based applications; obviously this is the most misleading state. We guess the underlying reason of this problem is that the users sought to find something at the summary screen of the location-based applications, but that information was not sufficiently visible.

An interesting phenomenon emerged in task 5. Three participants failed to progress through step 1 but surprisingly all of the participants successfully completed this task. This means that some participants avoided one fundamental state and reached the objective state through another route.

Table 5 States and transitions for all user actions of Task 1

	Activity		Event
a4	Most recent	e6	Click *most recent*
a5	My downloads	e7	Click *my downloads*
a6	Location-based search	e8	Click *location-based search*
a7	Top-rated	e9	Click *top-rated*

Table 6 Traffic flow metrics based on number of mistakes

Metrics	T1	T2	T3	T4	T5	T6
Step 1						
No. of backtracks (*)	12	10	29	11	11	61
Mistake state no.	4	3	12	6	4	8
Correct/incorrect flows	12/12	12/10	16/40	12/7	9/12	14/58
Correct flow ratio	50 %	54.5 %	28.6 %	63.2 %	42.9 %	24.1 %
Fail to pass no.	1	0	1	1	3	0
Step 2						
No. of backtracks	2	3	7	5	2	9
Mistake state no.	0	0	1	2	1	3
Correct/incorrect flows	9/3	13/3	9/7	11/6	11/2	6/8
Correct flow ratio	75 %	81.3 %	56.3 %	64.7 %	84.6 %	42.9 %
Fail to pass no.	0	0	0	1	0	4
Step 3						
No. of backtracks	0	0	0	19	1	N/A
Mistake state no.	0	0	0	2	0	N/A
Correct/incorrect flows	9/0	10/0	9/0	8/20	10/1	N/A
Correct flow ratio	100 %	100 %	100 %	28.6 %	90.9 %	N/A
Fail to pass no.	0	0	0	0	0	N/A
Step 4						
No. of backtracks	N/A	N/A	0	N/A	N/A	N/A
Mistake state no.	N/A	N/A	0	N/A	N/A	N/A
Correct/incorrect flows	N/A	N/A	9/0	N/A	N/A	N/A
Correct flow ratio	N/A	N/A	100 %	N/A	N/A	N/A
Fail to pass no.	N/A	N/A	0	N/A	N/A	N/A

*No. of backtracks = number of backtracking incidents

We provided this information to the developer of AppJoy who confirmed that the way it was designed had some problems.

For unhandled motion events, we inspected the activities in which unhandled motion events occurred and their physical positions on the screen. Among 39 unhandled motion events across all tasks, 30 of them (76.9 %) were clicks or moves at the caption bar in different activities; this was the case for all 11 events in task 3 and 17 out of 21 events in task 6. Although we didn't classify it as a usability problem, this phenomenon reflects the participants' frustration. As these behaviors happened frequently in the two tasks that participants had trouble dealing with. Maybe only when the users could not find other ways to complete a task, they touched the caption bar as a last resort. For the remaining 9 unhandled events, 8 of them were clicks or moves on blank pages between two activities; these actions were not noticed in laboratory-based usability testing. Even though we cannot conclude that lengthy loading time between two pages caused the users to perform such actions, at least this is an interesting finding that we did not expect: that such events can be captured by the event logging library. The last unhandled motion event was a mis-click on the side of one button that one participant intended to click, but this was a rare case.

For unhandled key events, we inspected the key code for each click and its related activity. Our library does not log the key code for keystrokes on the Android keyboard except for the delete key out of concern for the users' privacy. However, none of these unhandled key events were from keystrokes on the Android keyboard. Actually, only four keys were pressed: the volume down, volume up and camera focus keys on the right side of the device, and the search key below the screen (we did not classify the search key as a system key because most third party applications do not respond to this key). Except for the search key presses, other unhandled key events were not observed by the evaluators during the experiment. We speculate that these keys were probably pressed without the users' conscious purpose as these keys can be easily touched by mistake in daily usage. However, the search key presses constitute a different case, as 36 out of 49 unhandled key events are from search key presses, including all of the 30 unhandled key events in task 5. Recall that in task 5 we asked participants to uninstall the application they just installed in task 2; we can infer that in task 5, the users were trying to search the application

directly with the search key because they knew the application name, but apparently AppJoy did not handle this key event. This confirms a usability problem identified in laboratory testing that AppJoy should add some mechanisms to search or at least sort the recommended applications. Furthermore, 3 search key presses were from task 3 and 3 were from task 6, yet these two tasks did not involve looking for a special application. We guess that the participants intended to search the functionalities of AppJoy setting and usage time by doing so.

The above statistics only dealt with the number of total occurrences of each event but ignored the differences between the participants. The number may be difficult to interpret if a minority of the participants made a large number of mistakes. In fact, if only a few people had problems with a user interface component, this component's design is probably satisfactory. To avoid the difficulties of interpretation, we calculated how many participants backtracked and how many participants entered the most misleading state in each step of a task. In this way, we can alleviate the above issue: if the backtracking number for all participants was large but the number of participants who experienced backtracking was small, it means that a few participants were confused. Similarly, only if many participants backtracked from a particular state can we conclude that the state was really misleading. The two metrics related to the number of participants are summarized in the Table 7.

Regarding backtracking, the data in Tables 6 and 7 shows fairly uniform behavior across participants, because more backtracking across all participants corresponds to more participants navigating with backtracking. Regarding the correct flow ratio and most misleading state, while the data again shows fairly uniform behavior, there are some minor differences that can be observed. For instance, at step 3 of task 4, the number of incorrect flows was higher across all users, compared to that at step 1 of task 5, but fewer participants went to the most misleading state. This means the problem in task 5 is more general across the users, hence that problem should be considered to be more critical. In summary, we can look at the data in both of the tables to rate the severity level of usability problems.

Besides the above measurements, developers or evaluators can almost "replay" the users' behaviors if they have time to manually review the logged events. Though time-consuming, this approach can help to detect additional usability problems, even for some subtle issues that were overlooked during the laboratory-based usability testing. For example, there was a button that overlapped with recommended applications which made it hard to be seen, so that only one participant clicked that button and none of the evaluators observed this phenomenon during the experiment. But by examining the logged sequence, we noticed this event and confirmed that this was a usability problem.

6.3 Comparison results

Finally, we summed up all usability problems identified through the event logging toolkit, and compared the number with that discovered by laboratory-based usability testing in Table 8. Note that in this comparison we excluded the two participants' data that was lost with the event logging method.

Usability problems identified by laboratory-based usability testing are shown in parentheses (cited from Table 2) for comparison. From the rest of the 10 participants, we identified exactly the same number of usability problems as from all 12 participants in laboratory-based usability testing. We can easily see that the laboratory-based testing method can identify more cosmetic problems. All 5 cosmetic usability issues observed through laboratory testing were not discovered by the event logging method. But the event logging method is effective for identifying critical usability issues, including major and catastrophic usability problems. All

Table 7 Traffic flow metrics based on number of participants making mistakes

Metrics	T1	T2	T3	T4	T5	T6
Step 1						
No. of backtracks	4	6	4	2	6	8
No. to the MMS (*)	3	2	5	2	7	8
Step 2						
No. of backtracks	0	0	2	1	2	3
No. to the MMS (*)	0	0	0	2	1	2
Step 3						
No. of backtracks	0	0	0	4	1	N/A
No. to the MMS	0	0	0	4	0	N/A
Step 4						
No. of backtracks	N/A	N/A	0	N/A	N/A	N/A
No. to the MMS	N/A	N/A	0	N/A	N/A	N/A

*MMS = Most Misleading State

Table 8 Usability problems identified by auto logging method

	AppJoy problem	Android convention
Cosmetic	0(5)	1(2)
Minor	4(4)	1(2)
Major	5(4)	1(2)
Catastrophe	1(1)	0(0)

critical usability problems discovered through laboratory testing were found by the event logging method, and by manually reviewing the participants' behavior sequences, we found out one more major problem that was overlooked in the experiment. However, the shortcoming of the event logging method is, for most of the issues it identified, that although it can point out the location of a problem, it cannot tell the cause of that problem. This is an issue common to all event logging methods because they lack the information that can be gleaned from listening to participants' verbalized thoughts and observing participants' facial expressions and other nonverbal signals.

Compared to laboratory testing, the event logging method found fewer problems that were introduced by the users' unfamiliarity with Android conventions. We expected this because the library cannot record the users' interactions outside of AppJoy. But because the objective of this library is to find usability problems in third-party applications rather than in the Android framework itself, we do not consider this to be a big issue.

7 Discussion

We have only tested the event logging toolkit on one application, which is of course far from enough to conclude that it can be effective to help evaluators, developers and designers identify usability issues on all Android applications. We will integrate this library into more Android applications to validate its usefulness in the future. One thing to note is that our toolkit does not help applications developed without the Android UI framework, such as games based on OpenGL.

On the other hand, even after conducting just one test, we can already demonstrate that the proposed event logging toolkit can detect some subtle actions that are difficult to observe in laboratory testing, such as some quick moves and the unhandled events discussed above. Meanwhile, it can provide strong quantitative measurement and lots of statistical data describing users' interactions with the application. So it can at least complement traditional laboratory-based usability testing.

The SDK we are using has been upgraded to be compatible with Android Version 2.3.x (API level 10). As Android API means to be backward-compatible, hence to be consistent with future releases of Android API, we just need to add support for new listeners and UI-related methods, if there is any, and this is expected to be a small amount of effort. Android 4.x introduces a variety of new features, and alters the UI architecture to some extent, we plan to develop another SDK that fits into Android 4.x API. In concern with scalability to other mobile platforms, such as iOS, Adobe AIR on Android, PhoneGap, we believe as long as these platforms allow us to log event-based user interactions, we are able to use the uniform mechanism (as what we proposed in this paper) to do the analysis.

We tested the application in a WiFi network environment, which neglects possible networking problems that could happen under poor network conditions. Although we know that AppJoy sometimes has trouble connecting to the server under the 3G network, we did not identify this problem through this experiment. Hence we can see context information is needed to locate usability issues under some conditions, and this is precisely the weak point of laboratory-based usability testing. We will include context information retrieved from sensors of the Android device in the next version of the event logging toolkit. Because we can collect the users' interaction data in a real world environment, we can determine usability problems under different conditions through the toolkit. We anticipate that this approach will be a big advantage for an event logging method as it is more suitable for field-based usability testing.

Another interesting question we need to further address is whether our usability toolkit induces new usability problems to the applications being tested. From our design perspective, this toolkit is not expected to affect the performance of the subject application, as we simply add some logging statements into the original source code, and these statements are supposed to be executed pretty fast. Our main concern would be network and energy issues. We have a separate thread posting logged data through the network, which might compete bandwidth with other threads in this application. The impact could be nontrivial if the application is network-intensive. However, we can alleviate the impact by storing the events locally and only upload when the WiFi network is available. In addition, uploading UI events would aggravate battery consumption, especially if the network is intermittent. To study the performance impact, we will do quantitative measurements in the future.

The automated metric discovery model was not tested in a real user study, but we believe that it is as capable as what we can do with manual effort, for which we have validated that the automated model is able to accurately extract the usability related metrics. The automated model outweighs manual analysis for its significant savings on time and labor. More importantly, it is less error prone and may discover exceptional sequence which can be hardly observed in a manual way, such as *bypass* situation discussed in previous section. As a future work, we plan to test this model in real user studies. Also, we are currently researching on how to generate usability problem report in an automatic manner, and this might require a lot of empirical knowledge learned from the practical user studies.

8 Conclusion

It is challenging to conduct usability testing for mobile applications. In this paper, we present a UI event logging toolkit that can be embedded into Android applications. The toolkit requires minimum source code modification by the developers and automatically uploads fine-grained UI events to a central server. By testing a deployed Android application, a state-machine based sequence analysis is evaluated using the logged events and compared to traditional laboratory-based usability testing. The results show that the proposed toolkit is effectively capturing detailed interaction events, which can provide accurate and quantitative analysis of usability problems. In summary, the event logging toolkit can discover most usability problems comparable to those uncovered by the laboratory method, and also reveal some unexpected issues. In future work, we will extend the toolkit so it can be deployed for field-based journal usability testing.

Acknowledgement This work was partly supported by the National Science Foundation under Grant No. 1016823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Appendix

In above sections, we only dealt with the number of usability problems, without specifically pointing out what these usability problems are. Here we summarize the usability problems identified from the user study and categorize them by their severity levels.

- Cosmetic
    1. At the home activity of AppJoy, the users tried to trigger a menu by long-clicking an item, but the long-click listener was not implemented.
    2. At location-based search activity, if the search result is empty, the activity shows a blank screen, instead of cueing "no result found."
    3. At recommended application activity, there is an "install" button which misled the users to think that by clicking this button, the installation process will be finished (actually there are several steps).
    4. The position of menu items are not consistent.
    5. At *Help* web page, some links do not function.
- Minor
    1. At location-based search activity, users clicked on an non-editable box to alter location.
    2. At location-based search activity, recommended free applications did not provide information about whether they have been installed.
    3. At my downloads activity, several users complained the lack of search functionality.
    4. Some recommended applications cannot be found in Android Market.
- Major
    1. After the users completed installing an application, the "install" button did not disappear so that they thought the installation was not successful.
    2. The *Help* web page was outdated.
    3. The most recent activity was supposed to contain only applications that were not downloaded (according to the developer's design), but that was not the fact.
    4. At the detailed information activity, the text font was too small so that some important information was easily overlooked.
- Catastrophe
    1. The meaning of recommendation options at the home activity is not clear, and the users cannot well understand it across all tasks.

The severity level of a usability problem are mostly related with the frequency of its occurrence and the impact if it happens. We gathered up the notes taken by the evaluators, and summed up in which task and through which participant did we notice each usability problem in Table 9.

Table 9 Occurrence of usability problems

Usability problem	Which task	Which participant
Cosmetic 1	1	1, 4
Cosmetic 2	5	7
Cosmetic 3	2	8, 11
Cosmetic 4	N/A	Evaluator
Cosmetic 5	6	11
Minor 1	4	1, 12
Minor 2	4	3, 8
Minor 3	5	2, 7
Minor 4	2	7
Major 1	2	2, 12
Major 2	6	7, 11
Major 3	5	All
Major 4	6	9, 10, 11
Catastrophe 1	All	All

One cosmetic usability problem was discovered by the evaluators while the users performing the tasks, none of the users actually triggered the usability problem, so we did not specify a particular participant number here. Some usability problems occurred in almost all tasks or across almost all users, so we note *All* here for it.

References

1. Kerr R (2009) US mobile internet usage to near 50 % in 2013. In: Vator news
2. Budiu R, Nielsen J (2009) Usability of mobile websites: 85 design guidelines for improving access to web-based content and services through mobile devices. Nielsen Norman Group Research Report
3. Gomez Inc (2009) Why the mobile web is disappointing end-users. Equation Research Report
4. Tullis T, Fleischman S, McNulty M, Cianchette C, Bergel M (2002) An empirical comparison of lab and remote usability testing of web sites. In: Usability professional association conference, Orlando
5. Bastien JMC (2010) Usability testing: a review of some methodological and technical aspects of the method. Int J Med Inform 79:e18–23
6. West R, Lehman K (2006) Automated summative usability studies: an empirical evaluation. In: Proceedings of the SIGCHI conference on human factors in computing systems, ser. CHI '06. ACM, New York, pp 631–639
7. Waterson S, Landay JA (2002) In the lab and out in the wild: remote web usability testing for mobile devices. In: Conference on human factors in computing systems, pp 296–297
8. Hilbert DM, Redmiles DF (2000) Extracting usability information from user interface events. ACM Comput Surv 32:384–421
9. Yan B, Chen G (2011) Appjoy: personalized mobile application discovery. In: Proceedings of the 9th international conference on mobile systems, applications, and services, ser. MobiSys '11. ACM, New York, pp 113–126. Available: http://doi.acm.org/10.1145/1999995.2000007
10. Rosenbaum S, Rohn J, Humburg J (2000) A toolkit for strategic usability: results from workshops, panels, and surveys. In: Proceedings of the ACM CHI 2000 conference on human factors in computing systems, New York, pp 337–344
11. Usability Professionals' Association (2008) UPA 2007 Salary Survey
12. Ericsson KA, Simon HA (1980) Verbal reports as data. Psychol Rev 87:215–251
13. Nieslen J (2007) Severity ratings for usability problems. Retrieved June 4th from UseIt. Available: http://www.useit.com/papers/heuristic/severityrating.html
14. Akers D (2009) Backtracking events as indicators of software usability problems. Ph.D. dissertation