Evaluation of the Android Accessibility API Recognition Rate Towards a Better User Experience

Mauro C. Pichiliani(✉) and Celso M. Hirata

Department of Computer Science, Instituto Tecnológico de Aeronáutica, São José dos Campos, Brazil
{pichilia,hirata}@ita.br

Abstract. Mobile applications are based on interactive common UI elements that represents pointing targets visible on the screen. The usage of mobile applications in eyes-free scenarios or by individuals with vision impairments requires effective alternative access to visual elements, i.e. accessibility features. Previous works evaluated the accuracy of UI element's identification by accessibility APIs on desktop applications reporting that only 74 % of the targets were correctly identified, but no recent research evaluated the accuracy for similar mobile APIs. We present an empirical evaluation based on the Android accessibility API that computes the UI recognition accuracy rate on ten popular mobile applications. Our findings indicate that accessibility average recognition rate is 97 %.

Keywords: Accessibility · Android · Mobile · API · Evaluation · User interface · User experience

1 Introduction

The explosive growth of mobile devices is supported by the vast offer of applications available in online stores. The interaction model on these applications is primarily based on the touch and gestures performed on the screen representation of visual elements that compose the application's UI. However, there are special users, i.e. visually impaired, and circumstances that hinder or make the visual access to the display impossible, thus challenging application usage.

Accessibility features are one of the solutions used to allow access to visual elements and guide GUI interactions on computer platforms. Almost all operating systems provide guidelines and recommendations to make applications accessible based on existing accessibility APIs whose efficiency can be measured by their ability to recognize and identify UI elements.

This paper has the goal to evaluate the UI recognition accuracy rate of ten popular mobile applications by measuring how many UI elements are correctly identified by an accessibility API. The evaluation was based on the methodology that includes several observations of the UI elements and a special accessibility service that gatherers the location, size, type, and content data of the most common widgets found on mobile

M. Antona and C. Stephanidis (Eds.): UAHCI 2015, Part I, LNCS 9175, pp. 340–349, 2015.
DOI: 10.1007/978-3-319-20678-3_33

applications. Additionally, we also evaluated which events are captured by the API in order to create customized feedback notifications so the user can interact more consistently with the application being executed.

Identifying the properties of UI targets enables novel additions to existing technology and increases the support for development tasks. More specifically, the tasks that can benefit from accurate target identification of user interfaces include: (i) mobile accessibility services creation; (ii) automatic extraction of a task sequence; (iii) scripting of common actions; (iv) UI automation tests; and (v) support to collaboration frameworks.

The rest of the paper is organized as follows: Sect. 2 reviews the related work on evaluation of accessibility APIs. Section 3 presents the details of the mobile accessibility APIs and its applications. Section 4 discusses the methodology aspects employed to evaluate the recognition rate of UI elements along with details of the applications evaluated. Session 5 presents the results of the study. Session 6 provides a discussion and some examples of the UI interface elements that lack accessibility. Finally, Sect. 7 presents conclusions and possible developments for future work.

2 Related Work

The HCI literature has many research studies that collect data about the size, location, and other visual properties of UI interactive elements for usability evaluations [6], preference comparison [4], and pointing performance assessment [7]. Traditionally, data gathering is made on controlled laboratory settings with custom or adapted software techniques [4, 7, 8] that capture and hand code the interaction.

A recent previous work evaluated the accuracy of UI element's recognition by accessibilities APIs [5], which reports that only 74 % of the targets were correctly identified, i.e. the location and size indicated by the API matched the UI element's properties. This result is based on 1,355 targets covering real world usage of 8 arbitrary desktop applications on the Microsoft Windows XP Operating System with the Windows Automation API [11], formerly known as Microsoft Active Accessibility API (MSAA API).

On that study the authors did not evaluate the content of the element, which is the most important information needed by screen readers and other accessibility services that provide feedback to users. Additionally, the researchers did not studied the complexity of gestures and interactive UI elements found on mobile applications.

Despite the fact that there are approaches that increase the recognition rate of UI elements for desktop application [3, 5] and initiatives to define strategies, guidelines, and resources to make mobile web accessible [1], the recognition accuracy of native mobile OS accessibility APIs received little research attention.

3 Mobile Accessibility API

Accessibility APIs are designed to provide data regarding interactions on GUIs and are available on many operating systems and programming platforms. As an example, the Microsoft Active Accessibility API (MSAA API) [11] is a cross-application Windows operating system level solution for getting low-level information about targets, including push buttons, menus, textboxes, and other UI elements.

While the APIs can be extremely accurate at identifying some targets, Amant et al. [9] state, without formal evaluation, that in practice many real world targets are not supported by those accessibility APIs. In addition, not all applications support them in the same way. For example, two popular web browsers, Microsoft's Internet Explorer and Open-source Firefox, deal with content in a very different way, limiting the API's access to them [5].

While some mobile Operating Systems only provide ready to use accessibility services, i.e. screen reader or virtual magnifier glass, others provide a complete set of API and resources to develop accessibility services based on low-level hooks that capture OS events.

Among the three main mobile Operating Systems, Apple's iOS, Microsoft's Windows Phone, and Google's Android, the last is the only one that provides accessibility APIs [2] that gives developers a complete set of features to improve application usage to users who have special needs. With the APIs, all common visual controls have accessibility by default and developers can make their own application more accessible, or make their own accessibility services that provide enhancements for other applications.

Giving permissions from users, developers can access the on screen view hierarchy, accessibility events and enhance user experiences accordingly with the APIs. Those APIs act as a delegate between applications and the system exposing interface state so that an accessibility service can be aware of any interaction and interface changes triggered by the inputs. Developers can also leverage accessibility APIs to fire interaction events [10].

Besides the creation of accessibility services, the access to on screen interactions and elements provide technological support to frameworks that extract the sequence of tasks for scripting and resources to suits that automate UI testing without needing a real person to interact with an application. Another category of application that benefits from accessibility APIs are collaborative frameworks and groupwares that require instrumentation to capture and replay events on multi-synchronous collaboration scenarios found on the CSCW (Computer Supported Cooperative Work) area.

4 Evaluation Methodology

The evaluation methodology used to calculate the recognition rate of UI elements on mobile applications is based on the previous work [5] that automatically collects and compares screenshots of the interface elements with their real counterparts during real world tasks performed on selected desktop applications.

The approach that evaluates UI elements recognition rate may be biased due to the lack of developer's effort to implement accessibility properties on the desktop

application's interfaces being tested. However, unlike the desktop scenario, all standard controls inserted on the UI of Android applications already have special properties, such as the *contentDescription*, filled with textual data that is passed over to events trigged by the accessibility API. This standard accessibility description reduces the bias probability on our rate recognition evaluation.

However, in many cases the default value for the *contentDescription* do not provide enough information to understand the element's content or function. For instance, picture containers used as buttons often lack a description that can be used to inform and assist the user interact with the UI on eyes-free scenarios.

The 10 applications selected for the evaluations were chosen by the popularity criteria, i.e. number of users shown on the Google Play store at February 16th, 2015. Table 1 shows the applications evaluated along with the total number of elements on all activities, which are similar to the window concept in desktop application.

Table 1. The 10 mobile applications evaluated.

Application	App. Version Date	Popularity (users)	Activities (windows)	Interactive elements	Non-interactive elements	Total UI elements
Facebook	02/14/2015	25,941,692	17	94	36	130
WhatsApp Messenger	02/10/2015	23,218,150	18	70	14	84
Clean Master	02/15/2015	21,386,769	13	105	30	135
Instagram	02/11/2015	19,379,612	15	69	29	98
Messenger (Facebook)	02/13/2015	14,197,755	26	112	23	135
Viber	02/15/2015	6,425,682	25	109	43	152
Skype	02/09/2015	6,177,881	27	160	26	186
YouTube	02/13/2015	5,818,561	18	77	23	100
Twitter	02/09/2015	4,799,044	14	64	26	90
Maps	02/11/2015	4,598,545	11	63	13	76

The total UI elements values of Table 1 were computed by manually highlighting the elements on top of the application's screen while performing real world tasks such as registering, typing a post, and changing an application setting. The UI elements were classified as interactive elements that trigger events, i.e. edit texts, buttons, hyperlinks, list items, or as non-interactive elements, i.e. text labels, layout grids, and images. Since the evaluated applications have elaborated, dynamic, and reused activities, the following criterion was used to account for the total number of elements.

4.1 Dynamic Content

A common pattern found on applications while analyzing their IU was the dynamic generation of control's elements from user data. For instance, several applications filled interactive lists with list items that correspond to the user's friends. When we found this pattern we account for only one element on the dynamic control.

4.2 Dialog Messages

Many applications used dialog pop-up windows that showed notifications or questions asking the users to choose options by touching buttons. We account for non-interactive (text) and interactive elements (buttons) when we found these dialog windows.

4.3 Web Pages

All evaluated applications were native, i.e. they did not require an internet web browser. However, when certain options are accessed the applications started a new web browser session and redirected the user to a specific web page. Since the web page is not officially part of the application, all the UI elements found on the page were not considered.

4.4 Common Android Activities

The Android platform allows developers to reuse common activities and controls via the definition of an intent that requests actions to be performed by a shared application or service once the correct permission was granted. The camera capture intent is a conventional example: developers register the intent to call a common shared activity that takes a photo or record a video instead of building their own logic to interact with the device's camera. These common activities are not part of the application and, therefore, their UI elements did not add up to the total number of elements.

4.5 Reused Elements on Distinct Activities

The interface of most applications reused basic navigation controls, such as back buttons, menus and clickable images, on more than one activity. If the element is the same, but it is located on distinct activities, it is accounted for only once.

The recognition of UI elements by and accessibly API was evaluated on an Intel tablet (codename Medfield) with the 4.0.4 Android version (Ice Cream Sandwich) that allowed the inspection of the element's properties. Contrary to previous work [5] that used image comparison techniques to identify the location and size of UI elements, we created a simple accessibility service that gathered the elements data properties when an event callback was triggered.

The accessibility service assumed the form of a class that extended the *AccessibilityService* class. Its main feature is the *onAccessibilityEvent()* callback method invoked when an accessibility event is triggered from a user interaction with the UI. Inside the event the *AccessbilityEvent* class was used to obtain the UI element represented by the *AccessibilityNodeInfo* class, which queries the view layout hierarchy (parents and children) of the target component that originated the accessibility event.

Once the instance of *AccessibilityNodeInfo* was obtained its public fields and methods allowed the access to the element's state and visual properties. The internal resource ID used to identify the element inside the application is also provided along

with the value of the *contentDescription* property. However, the object received as a parameter of the callback method *onAccessibilityEvent()* is an instance of the *AccessibilityNodeInfo* class that do not correspond to an instance of the real control inserted on the activity.

The object instance of the *AccessibilityNodeInfo* class allows access to properties including the element's location, size, class, text and *contentDescription*. The *contentDescription* value is the element's content that is commonly used on accessibility services such as TalkBack, which is the default screen reader shipped with the Android OS.

5 Results

The location, size, and type provided by the accessibility layer are accurate and represent the correct properties of the element when an event is triggered. To calculate the recognition rate we focused on the API capability to trigger events that allows the access to the UI interactive element's properties and also the location, size, type, and content data recognition rate.

We evaluated the recognition rate only on interactive elements that allows the inspection of the *AccessibilityNodeInfo* object inside the *onAccessibilityEvent()* method. All non-interactive elements are obtained querying the view layout hierarchy once an event was triggered. While evaluating the API with this approach we found that when certain interactive elements were touched they did not generate any event at all. Although this lack of accessibility event generation is rare on the evaluated applications, we accounted for this fact and provided the Event Trigger Rate metric.

The most common type of event that triggered the *onAccessibilityEvent()* method was the TYPE_VIEW_CLICKED, which is a numeric constant that classify the event as a touch in any area of a View. Within this event an *AccessibilityNodeInfo* object can invoke the *getPackageName()* and *getClassName()* methods to identify precisely the type of target's element being clicked.

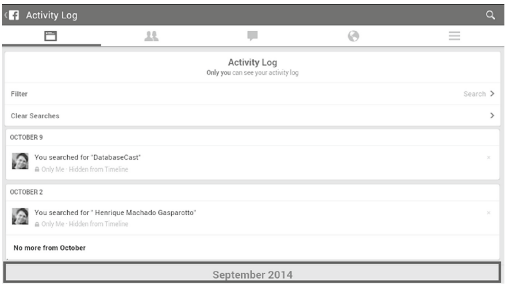
The methods *getBoundsInParent()* and *getBoundsInScreen()* allows the identification of the element's position relative to its container and on the real screen, respectively. The position of the element is obtained as a *Rect* object that has four coordinates, thus providing means to calculate the element's size.

The recognition of the content was evaluated by the presence of textual information that allows its identification. The methods *getText()* and *getContentDescription()* of the *AccessibilityNodeInfo* object returns the content information assigned manually by the developer or automatically by the accessibility API. Therefore, the Content Rate metric was calculated based on the presence of textual information provided by any of those two methods. Table 2 presents the values of the Event Trigger Rate and Content Rate metrics from the 10 Android applications evaluated. The Overall Recognition Rate is the average value of the two previous mentioned rates.

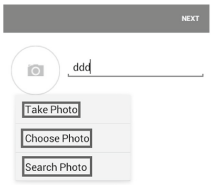
The evaluation of the recognition rate accounted for 923 interactive elements found on 185 activities across 10 applications. The average Event Trigger Rate was 99.69 % (*s.d.* 0.53) and the average Content Rate was 93.65 % (*s.d.* 3.35). The average Overall Recognition Rate was 96.87 % (*s.d.*1.88).

Table 2. Rates of the applications' trigger events and element content.

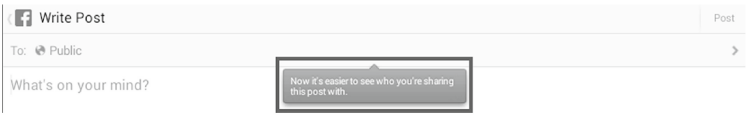
Application	Event Trigger Rate (%)	Content Rate (%)	Overall Recognition Rate (%)
Skype	100	98.75	99.37
WhatsApp Messenger	99.8	98.57	99.18
Viber	99.8	98.16	98.98
Instagram	100	94.20	97.1
Maps	100	93.65	96.82
YouTube	100	93.51	96.75
Twitter	99.8	92.19	96.00
Messenger (Facebook)	98.6	92.86	95.73
Clean Master	100	90.48	95.24
Facebook	99.8	88.30	93.55



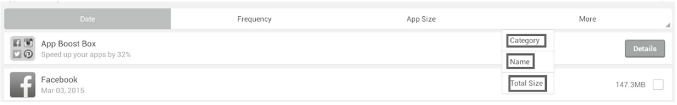
a) Year history button



b) Choose photo menu



c) Information



d) Ordering menu

Fig. 1. Highlighted interactive elements that did not triggered accessibility events: (a) Year history button on Facebook's Activity Log screen; (b) Phone menu items on messenger's new group screen; (c) Pop up information on facebook's write post screen; and (d) Ordering element items on clean master's app manager screen.

6 Discussion

The applications evaluated on this study have a high Event Trigger Ratio and five of them received the 100 % value for this metric, meaning that all interactive elements have triggered events that were captured by the accessibility API. The other

applications lack events for elements such as buttons, menu items, list items, and check boxes. These elements did not raised events because they were created dynamically on execution time or are inside a pop up menu. Additionally, custom controls created by

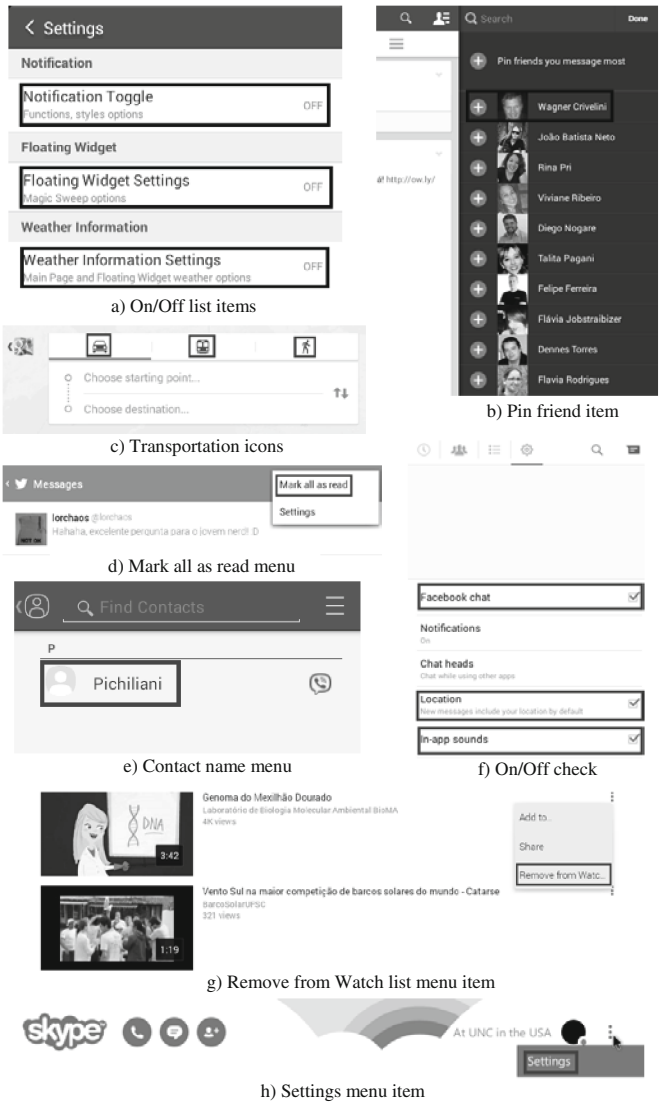


Fig. 2. Highlighted UI elements without textual content: (a) On and Off list items on Clean Master's Settings screen; (b) Pin friend item on Facebook's Pin friends screen; (c) Transportation icons on Map's Route start/destination screen; (d) Mark all as read menu item on Twitter's Messages screen; (e) Contact name item on Viber's Find contacts screen; (f) On and Off check items on Messenger Settings screen; (g) Remove from Watch list menu item on YouTube's History screen; (g) Settings menu item on Skype's home screen.

developers did not raised accessibility API callback events. Figure 1 highlights with rectangles some elements that were unreachable by the accessibility API layer.

The Content Rate metric evaluated the presence of textual information associated with the element that received the interaction and raised an accessibility event. Almost all simple UI elements, such as a button or labels, have the default accessibility description obtained from the element's name, text, caption, or title property.

Conversely, controls that contain images or combine text and images also had descriptive information. For example, when the profile image of the user received a touch the user's name was sent to the accessibility API. Similarly, image buttons shown on the toolbar to allow the user to navigate back to the application's home screen have "navigate up" or "back" descriptions. The presence of these descriptions demonstrates the developers' effort to implement accessibility features, since the texts are not provided by default for images or controls that don't have text, caption, or title properties.

The controls that did not have descriptions are icons, pop up menu items, custom list items with images, on/off and check controls. The evaluation of the interface shows that most of the controls without accessibility descriptions are inside the settings screen or on dynamic areas. Another common scenario that led to the absence of the element description happened when the interaction generated a simple transitory on screen message produced by the use of the *makeText()* static method of the *Toast* class. Figure 2 shows examples of UI elements that raised interaction events captured by the accessibility API that did not have any description that enables the identification of content.

In general, the accessibility of the evaluated applications is high and the authors did not found impeditive barriers to infer what is on the interaction focus by analyzing only the resources provided by the applications and the accessibility API. Even on the lowest ranked application, namely Facebook, the identification of the interactive UI elements provided enough context and information to guide the actions performed.

7 Conclusions and Future Work

Nowadays the usage of current mobile applications is guided by the visual access to on screen representation of targets in which users interact with. To cope with visually impaired users or eyes-free scenarios, mobile developers rely on platform specific APIs that provides special accessibility features.

This work evaluated the UI recognition accuracy rate on ten popular Android mobile applications by measuring how many UI elements are correctly identified by users when OS accessibility APIs are used. Our findings indicate that overall recognition rate is roughly 97 %, which include the recognition of the UI elements' location, size, type, and content along with the user interactions that triggered accessibility events.

Formal mobile accessibility API evaluations, especially on popular applications, are important to provide evidences that application are accessibly enough to be used by people with visual impairments or at eye-free situations. The results of the evaluation suggested in this work can influence mobile developers to review the accessibility of their applications and also provide basic support for other contexts, including (i) mobile

accessibility services creation; (ii) automatic extraction of a task sequence; (iii) scripting of common actions; (iv) UI automation tests; and (v) support to collaboration frameworks.

Future work includes the evaluation and comparison of other Operating System mobile accessibility APIs and prospective validation of accessibility services (such as screen readers) that are based on the APIs. Another possible direction is the evaluation of the improvement that can be achieved on mobile applications when pixel-based techniques augment accessibility APIs.

References

1. Abou-Zahra, S., Brewer, J., Henry, S.L.: Essential components of mobile web accessibility. In: Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility, article No. 5 (2013)
2. Accessibility | Android Developers (2015). http://developer.android.com/guide/topics/ui/accessibility/index.html
3. Dixon, M., Laput, G., Fogarty J.: Pixel-based methods for widget state and style in a runtime implementation of sliding widgets. In: Proceedings of the CHI 2014 Conference on Human Factors in Computing Systems, pp. 2231–2240 (2014)
4. Findlater, L., McGrenere, J.: A comparison of static, adaptive, and adaptable menus. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM Press, pp. 89–96 (2004)
5. Hurst, A., Hudson, S.E., Mankoff, J.: Automatically identifying targets users interact with during real world tasks. In: Proceedings of the 15th International Conference on Intelligent User Interfaces, pp. 11–20 (2010)
6. Kaufman, D.R., Patel, V.L., Hilliman, C., Morin, P.C., Pevzner, J., Weinstock, R.S., Goland, R., Shea, S., Starren, J.: Usability in the real world: assessing medical information technologies in patient's homes. J. Biomed. Inform. **36**(1/2), 45–60 (2003)
7. Keates, S., Hwang, F., Langdon, P., Clarkson, P.J., Robinson, P.: Cursor measures for motion impaired computer users. In: Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility, pp. 135–142. ACM Press (2002)
8. MacKenzie, I.S., Kauppinen, T., Silfverberg, M.: Accuracy measures for evaluating computer pointing devices. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 9–16. ACM Press (2001)
9. StAmant, R., Lieberman, H., Potter, R., Zettlemoyer, L.: Programming by example: visual generalization in programming by example. Commun. ACM **43**(3), 107–114 (2000)
10. Zhong, Y., Raman, T.V., Burkhardt, C., Biadsy, F., Bigham, J.P.: JustSpeak: enabling universal voice control on Android. In: Proceedings of the 11th Web for All Conference, p. 36 (2014)
11. Windows Automation API (2015). https://msdn.microsoft.com/en-us/library/windows/desktop/ff486375%28v=vs.85%29.aspx