Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement

Xiaoyi Zhang, Anne Spencer Ross, James Fogarty
Paul G. Allen School of Computer Science & Engineering
DUB Group, University of Washington
{xiaoyiz, ansross, jfogarty}@cs.washington.edu

ABSTRACT
Accessibility issues in mobile apps make those apps difficult or impossible to access for many people. Examples include elements that fail to provide alternative text for a screen reader, navigation orders that are difficult, or custom widgets that leave key functionality inaccessible. Social annotation techniques have demonstrated compelling approaches to such accessibility concerns in the web, but have been difficult to apply in mobile apps because of the challenges of robustly annotating interfaces. This research develops methods for robust annotation of mobile app interface elements. Designed for use in runtime interface modification, our methods are based in screen identifiers, element identifiers, and screen equivalence heuristics. We implement initial developer tools for annotating mobile app accessibility metadata, evaluate our current screen equivalence heuristics in a dataset of 2038 screens collected from 50 mobile apps, present three case studies implementing runtime repair of common accessibility issues, and examine repair of real-world accessibility issues in 26 apps. These contributions overall demonstrate strong opportunities for social annotation in mobile accessibility.

Author Keywords
Robust annotation; runtime modification; accessibility.

ACM Classification Keywords
Human-centered computing → Accessibility systems and tools.

INTRODUCTION
Mobile apps have become ubiquitous, used in accessing a wide variety of services online and in the physical world (e.g., financial services, transit information). However, many apps remain difficult or impossible to access for people with disabilities, an estimated 15% of the world population [30]. For example, recent research examined the prevalence of accessibility issues in a sample of 100 Android apps, finding that every app included at least one accessibility issue [33]. 95% of the examined apps included touchable elements that were smaller than recommended by Android's accessibility guidelines [17], making them difficult to access for many people (e.g., people with motor impairments). 94% included elements that lacked alternative text, making them difficult to access using a screen reader (e.g., for people with visual impairments). 85% included elements with low text contrast, another barrier for people with vision impairments.

In addition to accessibility issues that can be objectively defined and detected by tools like Google Accessibility Scanner [14], considering the context of an interaction often reveals additional barriers. For example, a person using a screen reader may need to swipe 10 to 20 times before the focus moves to elements that should be readily available (e.g., the "Menu" button in the Dropbox app). As another example, a single inaccessible element can often undermine the overall functionality of an app (e.g., the 5-star rating element of the Yelp app lacks accessibility support, leaving this core functionality inaccessible to many people).

Mobile platforms have begun to support interactive correction of accessibility failures. For example, Android's TalkBack screen reader allows end-users to add custom labels to elements where an app developer has failed to provide a label. However, this functionality is limited to *ImageButton* or *ImageView* elements and requires an app developer has provided a *ViewIDResourceName*, an optional property that is often not specified. In an evaluation reported later in this paper, we examined 50 apps and found that TalkBack can apply custom labels to less than 13.6% of elements that it visits. TalkBack also does not support correcting an element that does have a label, even if that label is misleading.

Prior research in the runtime repair of accessibility failures has often focused on the web, in part because a webpage's underlying representation is available and can be modified prior to rendering by the browser. Social annotation is one powerful approach to accessibility repair [24,40], in which people annotate interface elements with metadata that is then used to repair accessibility failures in future interactions (e.g., annotating images that lack alternative text with text that can then be presented to future people who encounter that image using a screen reader). Such approaches require a robust method for determining when an annotation is applied, typically addressed via the combination of a URL (i.e., indicating the context in which an annotation is applied)

UIST '18, October 14–17, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5948-1/18/10…$15.00
https://doi.org/10.1145/3242587.3242616

and an XPath (i.e., within the context of the URL, indicating which element is annotated). Despite advances in runtime enhancement of mobile apps [31,48], social annotation remains difficult to apply in mobile apps because of a lack of methods for specifying an annotation context (i.e., the lack of a robust notion of a screen identifier, analogous to a URL).

This paper addresses this underlying requirement for robust annotation of mobile app interface elements. We develop a template-based approach to a screen identifier, implemented for the Android platform. We then use this in demonstrating several types of runtime accessibility repair: 1) applying custom labels to interface elements, 2) correcting navigation order, and 3) authoring accessibility context for inaccessible customized elements (e.g., a 5-star rating element).

The specific contributions of our work therefore include:

- Development of methods for robust annotation of mobile app interface elements. Designed for use in runtime interface modification, our methods combine a novel approach to screen identifiers and screen equivalence heuristics with familiar techniques for Android element identification.
- Implementation of initial developer tools for annotating mobile app accessibility metadata, including tools for authoring annotations and applying annotations at runtime.
- Evaluation of our current screen equivalence heuristics in a dataset of 2038 screens collected from 50 mobile apps.
- Three case studies demonstrating the implementation of runtime repair of common accessibility issues, each using the robust annotation methods developed in this research.
- An examination of repairing real-world accessibility issues in 26 apps, including popular Android apps, apps with accessibility issues reported in online forums, and apps identified through an in-person interview with a person who regularly uses the Android TalkBack screen reader.

RELATED WORK

Our current research is informed by prior research in social annotation and web accessibility, in runtime interface modification, and in screen and element identifiers.

Social Annotation and Web Accessibility

The web's representation has long encouraged enhancement of content through annotation, from annotation capabilities in Mosaic [43] to W3C efforts in interoperable annotations [44]. Prior research has developed robust web annotation to enable content customization and adaptation for end-users [4,5,28]. Extensive research in social accessibility has applied social annotation to web accessibility (e.g., [3,22,34,35,40,41]). Systems have explored various techniques, with a core that: 1) a person observes an accessibility failure, 2) that person or another person annotates the interface with metadata used by a tool that can repair the failure, and 3) annotation data is shared so that future users of the interface benefit from the repair. Examples of research in social annotation for web accessibility include providing image alternative text and other metadata [35,40,41], repairing navigation order [34], sharing scripts for site-specific repairs [3], and designing infrastructure for crowdsourcing contributions [22].

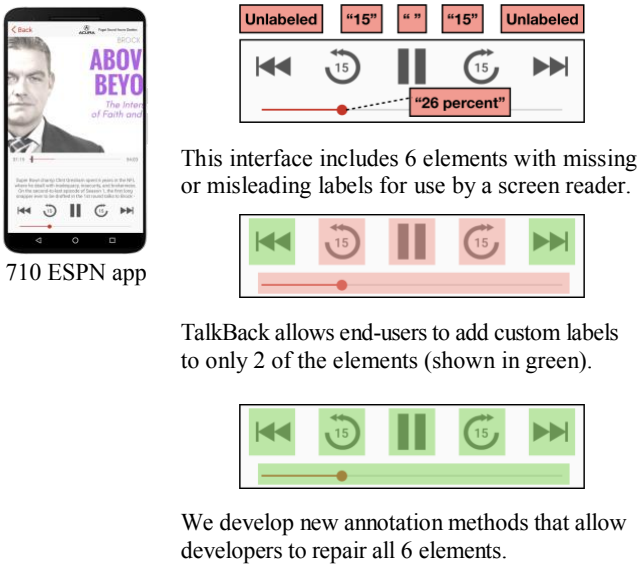

Figure 1: Missing and misleading labels are a common and important accessibility issue that can be addressed by new approaches to robust annotation for accessibility repair.

Runtime Interface Modification

Techniques for modifying the web benefit from the ability to directly modify a page prior to its rendering by a browser (e.g., modifying the HTML, CSS, or DOM). In contrast, non-web architectures generally lack an ability to access or modify internal representation, requiring different approaches to runtime modification. Prior research in desktop interfaces has replaced of an application's toolkit [12,13,29] or used window manager redirection of input and output [8,39,42]. A meaningful modification requires understanding the content and state of that interface, generally obtained through a combination of accessibility API data (e.g., [6,39]) and pixel-based analysis of interface content (e.g., [8,9,10,20,46]).

Less research has explored runtime interface modification in mobile platforms, in part due to their security architectures and greater concern for performance (e.g., a challenge for pixel-based techniques developed in the desktop context). Notable examples of runtime accessibility enhancements have included macro support [31] and pointing enhancement [48]. The SWAT framework requires rooting a device for an accessibility service to obtain system-level instrumentation of content and events [32], but rooting is a significant security risk and also presents a technical expertise barrier. Recent research in interaction proxies demonstrates a strategy for runtime accessibility repair and enhancement without rooting a phone, without requiring an app's source code, preserving all capabilities of a phone's built-in accessibility infrastructure [47]. These techniques modify interaction using floating windows inserted between an app's original interface and the manifest interface a person perceives and manipulates, deciding how to coordinate and modify interaction with the added floating windows according to information available via standard platform accessibility APIs. The robust annotation techniques we develop in this paper are also based entirely on information available via standard platform accessibility APIs, offering the potential

for broad deployment. Later sections demonstrate several case studies of runtime accessibility repairs implemented using a combination of annotations and interaction proxies.

Screen and Element Identifiers

As noted in the introduction, social annotation on the web has generally been implemented via the combination of a URL (i.e., indicating the context in which an annotation is applied) and an XPath (i.e., within the context of the URL, indicating which element is annotated). This strategy cannot be directly applied in mobile apps because the various screens of an app lack robust identifiers (i.e., the equivalent of a URL). Robust annotation of mobile apps therefore requires both: 1) methods for identifying a screen within an app, and 2) methods for identifying specific elements within that screen.

Prior research exploring *screen identifiers* has not been motivated by runtime interface modification and is generally inappropriate for that purpose. For example, Flow is an Android developer toolkit that allows naming interface states, navigating between them by name, and remembering the history of states [37]. However, such a toolkit must be integrated by an app's developer and cannot be used to reason about screens as part of an external repair.

The Rico project developed a large dataset of mobile app designs, gathered by capturing data during crawls of mobile apps [7]. Rico encounters a screen identification problem in determining whether an interaction during a crawl results in an app entering a new interface state or a state that has previously been captured. They define a context-agnostic similarity heuristic that compares two screens based on: 1) the number of pixels that differ in the two screen images, and 2) the number of differences when comparing the values of *ViewIDResourceName* for elements of the two screens. Two screens were considered the same if both were below manually-tuned thresholds, requiring the same value for 99.8% of pixels and all but 1 *ViewIDResourceName* value. These thresholds resulted in an estimated 9% error rate (6% error incorrectly determining two screens were the same, 3% error incorrectly determining two screens were different). Rico's use of pixel comparison is appropriate for crawling, but problematic for runtime modification (e.g., it requires additional permissions, can present performance challenges).

Other mobile app crawls similarly attempt to minimize revisitation of known screens by testing for similarity. For example, DECAF and PUMA define a generic feature vector encoding the structure of a screen's interface hierarchy, then use a cosine-similarity metric to determine screen equivalence according to a threshold [21,26]. An evaluation in DECAF with a .92 threshold estimated a 20% error rate (including 8% error incorrectly determining two screens were the same and 12% error incorrectly determining two screens were different). The threshold can be varied to obtain different tradeoffs between thoroughness and speed of a crawl, but a developer cannot otherwise correct either class of error.

In contrast to both of the above, the screen identification methods we develop in this paper: 1) use more information in the structure of the interface hierarchy to reduce overall error, and 2) allow the developer of an accessibility repair to explicitly correct any errors in screen identification to ensure robust annotation for runtime interface modification.

Automated testing tools address a need to be in a known state by executing pre-defined interaction sequences that bring an app to known screens (e.g., [1,18,36]). Because the developer of a test knows what screen will be active in each step of that test, they can reference elements of that screen. For example, *UiSelector* is an element identifier used in Android tools [19], specifying elements by properties such as *ContentDescription*, *ClassName*, *State* information, *Text* value, and location in an interface hierarchy. Within the context defined by a screen identifier, we use a similar approach to *element identifiers* so that we leverage developer familiarity with this approach.

ANDROID BACKGROUND

This section reviews several Android capabilities. We first discuss why existing capabilities are inappropriate for robust annotation, then provide background on accessibility services and Android's existing limited repair capabilities.

Android's accessibility services expose a *WindowId* for each *View*. Intended to support input interactions across multiple processes, *WindowId* is not stable (i.e., it will change each time an app is launched). It is therefore inappropriate as an identifier for storing annotations for use in future sessions.

When available, Android's accessibility services also expose a *ViewIdResourceName* for each *View*. *ViewIdResourceName* is Android's primary approach to a robust identifier (e.g., to be used in automated testing). Unfortunately, it is optional and often not specified by an app developer. When specifying an app's layout in an XML layout file, including a *ViewIdResourceName* allows a developer to obtain a reference to that element at runtime (i.e., similar to web programming practices of accessing an element according to an *id* attribute). However, app developers commonly create interface elements directly in code, obtain direct references to those elements, and therefore see no reason to specify a *ViewIDResourceName*. *ViewIDResourceName* is also not required to be unique, and the same value may be used by multiple elements in different contexts (e.g., elements in different screens of an app). *ViewIDResourceName* is therefore also not an adequately available and robust identifier for annotating Android elements.

Android allows an accessibility service to capture an image of the screen if a person grants screenshot permission to the service. A person may refuse this permission. Apps can also specify a *FLAG_SECURE* to disable screen capture, a common practice in apps that contain sensitive information (e.g., in banking apps). Prior research has examined pixel-based analysis and annotation (e.g., [8,9,10,20,46]), but the application of those techniques in mobile apps is limited by potential lack of access to screenshots and concerns for mobile performance challenges in pixel-based analysis.

Our approach focuses on using information available via the standard Android accessibility APIs. Each interface element

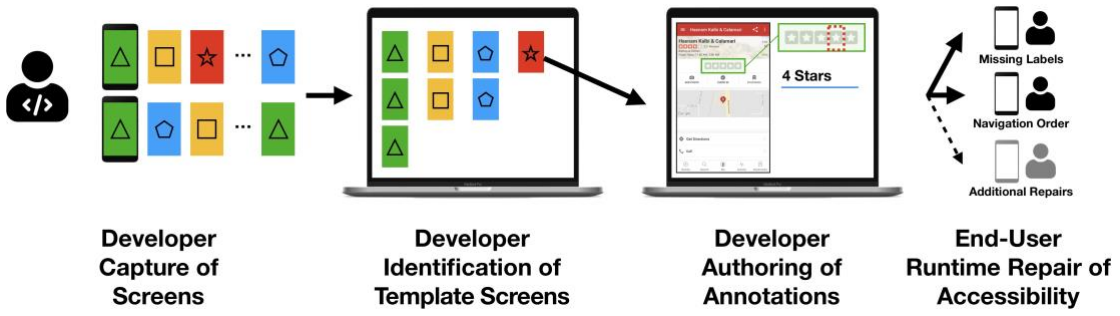

Figure 2: We develop and evaluate new methods for robust annotation of mobile app interface elements appropriate for runtime accessibility repair, together with end-to-end tool support for developers implementing accessibility repairs.

is represented as an *AccessibilityNode* that exposes properties of that element (e.g., *ClassName*, *AvailableActions, Text, ContentDescription*). Each *AccessibilityNode* can also access its parent and any children, allowing us to obtain and consider the tree of all interface elements in a screen.

As noted in our introduction, Android's TalkBack screen reader allows adding custom labels to elements. When a person navigates to an element without alternative text, they may perform a gesture to open the "local context menu", find the "add label" option, and enter a label for that element. Current support for interactive correction has several limitations: 1) It is often difficult for a person with a visual impairment to know the correct label for an unlabeled element, which can require trial and error or seeking assistance from another person. 2) TalkBack support for correction applies only to *ImageButton* or *ImageView* elements, which must have a *ViewIDResourceName* assigned by the app developer. 3) TalkBack does not allow replacing an existing label, even if it is incorrect or misleading. Improved support could enable significant advances relative to such limitations (e.g., greater ability to annotate elements, new ability to share robust annotations among the many people using an app).

APPROACH AND SYSTEM OVERVIEW

Our focus is on developing an approach to robust annotation of mobile app interface elements. Figure 2 overviews our approach, currently implemented in a set of related tools. Later sections discuss details of our approach and our current tools in more detail, while also emphasizing that different tools can be composed to implement the overall approach.

Beginning from the left of Figure 2, a developer implementing an accessibility repair first captures *screens* they would like to annotate within an app. We have developed a *capture tool* that can be used for this, implemented as an Android accessibility service. With the tool running in the background, the developer visits each screen that will be annotated. A software button added by the tool then allows capturing a current screen (i.e., a screen image and a snapshot of the current *AccessibilityNode* hierarchy).

Collection will often include multiple captures of the same screen, as illustrated by color and shape in Figure 2. This can occur when a screen is visited multiple times during collection, or when a screen is captured with different content (e.g., the same Yelp rating screen captured for different restaurants).

The developer of a repair identifies *template screens* that correspond to unique screens in the app. A *template tool* displays unique template screens in a row, with captured variations displayed in the column underneath each template. The tool applies our current *screen equivalence heuristics*, as discussed and evaluated in later sections, so that templates are automatically identified. A developer therefore only needs to inspect and potentially correct identified templates.

A developer then authors annotations using a combination of a *screen identifier* for the template screen in which an annotation applies, an *element identifier* for the annotated element within that screen, and the *metadata* to be associated with that element. We have developed an *annotation tool* to support authoring annotations. It displays the screen image, uses *AccessibilityNode* data to generate an element identifier when a developer selects an element in the image, provides highlighted feedback on elements that will be selected as a developer edits an element identifier, and allows inputting JSON-formatted metadata to be included in an annotation.

A developer can then create an accessibility service that uses annotations for the runtime repair of accessibility issues on end-user devices. We have developed a *runtime library* that supports comparing the current screen of an app against template screens for that app. If the current screen matches a template, the library further supports testing element identifiers of annotations against the current screen. The accessibility service can then use matching annotations in applying its runtime repairs. Later sections discuss three example services we have implemented using this approach.

Supporting a Range of Accessibility Repair Scenarios

Our approach and tools can support a variety of scenarios for a developer implementing annotation-based accessibility repair. Two example scenarios can include: 1) targeted repair in one or two screens of an app, or 2) more general-purpose repair of a class of errors across many different apps.

In the first scenario, a developer might decide to modify the navigation order within a specific screen of a specific app (e.g., in the "file explorer" screen of the Dropbox app). The developer can open the app, visit the screen to be repaired, and capture its data. The developer can then inspect that data in our annotation tool, obtain a screen identifier for the "file explorer" screen, and obtain element identifiers for elements to be modified. The developer could then implement

a custom accessibility service that: 1) uses our runtime library to detect when the app's context matches the "file explorer" screen identifier, 2) obtains references to interface elements in the screen using their element identifiers, and 3) uses the references to re-order navigation in that screen.

In the second scenario, the developer may find they want to extend their repair to other apps in which people report the same type of accessibility issue. Instead of developing many such specialized repair services, the developer can generalize their repair service. They can remove the use of specific screen identifiers and element identifiers, instead defining an annotation type and modifying their code to repair navigation according to any annotations available for the current screen. This might be sufficient for their needs, they might extend our annotation tool to make it easier to author such annotations, or they might examine new approaches to supporting a community of interested people in annotating many apps.

IMPLEMENTING ROBUST ANNOTATION

Annotation is implemented using a combination of a *screen identifier* and an *element identifier*. A screen identifier corresponds to a *template screen*, and a set of *screen equivalence heuristics* are used in both: 1) defining template screens (i.e., determining whether a screen is a *variation* of an existing template screen or distinct from existing templates), and 2) runtime identification of screens (i.e., determining whether the current screen matches a screen identifier). This section discusses each of these key components.

Screen Identifier

A screen identifier corresponds to a template screen and any variations of that screen, where a variation informally has the same screen structure with minor differences in content (e.g., images, text, number of items in a list). Annotations applied to a template will also apply to any variations, which both: 1) minimizes effort that might otherwise be spent annotating many different versions of a screen, and 2) allows our approach in screens containing dynamic content that could not otherwise be feasibly annotated.

A set of template screens is initialized with the first captured screen (i.e., a single template with no variations). Each screen is then compared against the set of current templates using *screen equivalence heuristics*. If a screen is equivalent to an existing template, it is added as a variation. Otherwise, the screen is used as a new template. Although a capture includes both a screen image and accessibility data, the image is used only for developer inspection and annotation. Screen equivalence heuristics must be based in the accessibility data, because the image will not be available at runtime. A developer may also use any variation as the representative screen for a template, as all of the variations are equivalent.

At runtime, an accessibility service can capture accessibility data for a screen and use our runtime library to compare that screen against the set of template screens for that app. This uses the same *screen equivalence heuristics*. If a match is found, annotations associated with that template screen are considered relevant to the current screen of the app.

Our template tool generates a unique and random screen identifier for each template screen (e.g., "screen_1520907"). A developer may also associate a human readable identifier with the screen identifier (e.g., "file explorer"), while ensuring human readable names are unique within an app. Identifiers can then be used with our runtime library to detect a screen (e.g., for a repair to the "file explorer" screen).

Element Identifier

A developer can then reference elements in a template screen using techniques familiar in Android testing frameworks (i.e., *UiSelector* and *XPath* selectors). Our implementation of these selectors also differentiates between *stable* and *dynamic* properties. Stable properties are unlikely to change between screen content updates (i.e., between variations), including *ClassName, Depth, IsLeaf,* and *ViewIdResourceName*. Dynamic properties are more likely to change, including *ContentDescription*, *Location*, *NumberOfChildren, Size,* and *Text*. The current set of properties could be extended if necessary or if future versions of the Android Accessibility APIs expose additional properties of interface elements.

Our annotation tool also automatically generates a unique and random element identifier for each element in a template screen (e.g., "element_59401"). Each default element identifier corresponds to a selector including the element's path in the hierarchy and its stable properties. A developer can verify the default selector by using the annotation tool to inspect how it applies in each variation. If necessary, a developer can edit the default selector, again inspecting how it applies in each variation. They may also associate a human readable name with an element identifier (e.g., "menu button"). At runtime, an accessibility service can use our runtime library to obtain a reference to an interface element using either an element identifier or a supported selector.

Screen Equivalence Heuristics

Annotation requires screen equivalence heuristics for determining a set of template screens for annotation and determining whether the active screen of an app matches one of those templates. As previously noted, we rely only on information available via standard Android accessibility APIs, so that: 1) our runtime library does not require rooting end-user devices, and 2) our runtime library does not require pixel-based analysis of screen images, which may be unavailable and may present performance challenges in a mobile app. Our heuristics are instead based in two key insights regarding identification of template screens.

First, contexts where Android identifiers fail to correspond to a meaningful notion of a screen are not random (i.e., are not well described by ignoring any one *ViewIdResourceName* nor by treating them as noise in a similarity metric). Instead, they are often systematic, resulting from developer behaviors (e.g., failing to provide an identifier, copy-pasting code resulting in non-unique identifiers) or standard toolkit behaviors (e.g., widgets that dramatically change what is presented in a screen with only subtle indications of that change in the accessibility API information for the screen). We develop a set of heuristics based in such systematic

behaviors, and we evaluate our heuristics in a later section. We note these heuristics can also be updated and extended as we gather additional data or as toolkit behaviors evolve (e.g., introducing new widgets that require adjustments).

Second, the two types of error in screen equivalence have different implications. We define a *FalseSame* error as incorrectly determining two screens are the same. This can result in what should be a distinct template screen instead being considered a variation of an existing template (i.e., requiring developer correction), or it can also result in a runtime screen matching an incorrect template and retrieving incorrect annotations. We define a *FalseDifferent* error as incorrectly determining two screens are different. This can result in additional annotation overhead through the creation of spurious templates that could be combined, or it can result in a screen not being annotated at runtime. Our techniques allow the developer of an accessibility repair to correct either form of error, but we design our default screen equivalence heuristics to minimize *FalseSame* errors. This corresponds to preferring a need for greater annotation effort over the possibility of annotations being incorrectly applied at runtime.

Our current screen equivalence is implemented using eight heuristics, each based on a specific app developer practice or toolkit behavior. Heuristic 1 makes an early determination based on explicit app developer indication that screens differ. Heuristics 2 to 5 account for common interface structures that require special consideration, *transforming* the accessibility API representation to better support comparison. Heuristic 6 then *filters* items that should not be considered in comparison. Given these special case checks and adjustments, Heuristic 7 then makes the primary comparison based on values of *ViewIdResourceName* in the two screens. Heuristic 8 then further reduces *FalseSame* errors by comparing values of *ClassName* in the two screens. After discussing each heuristic, we discuss how a repair developer can correct any errors.

1. Compare *ActivityName*: If two screens both have an *ActivityName* value that was specified by the developer, but not the same value, the screens are considered different. This heuristic is intended to reduce *FalseSame* errors.
2. Check for Navigation Drawer: This common Android element presents a menu above an interface by dimming and preventing interaction with elements under the menu. When this heuristic detects an open navigation drawer, it transforms the representation of the interface so remaining heuristics apply only to contents of the menu (i.e., ignoring the occluded background elements). If one screen contains an open navigation drawer, but the other does not, the screens are considered different. This heuristic is intended to reduce *FalseDifferent* errors.
3. Check for a Floating Dialog: This common Android element also occludes elements underneath it. This heuristic similarly detects a floating dialog, transforms the representation so remaining heuristics apply only to contents of the floating dialog, and considers two screens different if only one contains a floating dialog. This heuristic is intended to reduce *FalseDifferent* errors.
4. Check for Tab Layout: Android's tab layout preloads the content of each tab, presenting the same tree to the Android accessibility APIs regardless of which tab is selected. When this heuristic detects a tab layout, it uses a binary *Selected* property of the active tab to transform the representation so remaining heuristics apply according to the content of that active tab. It also considers two screens different if only one contains a tab layout. This heuristic is intended to reduce *FalseSame* errors.
5. Check for Radio Button Group with a Multi-Page View: This alternative approach to tab-like functionality similarly results in an Android accessibility API tree structure that does not adequately correspond to the selected radio button. This heuristic uses a binary *Checked* property of the active radio button to transform the representation so remaining heuristics apply according to content of the active view. This heuristic is intended to reduce *FalseSame* errors.
6. Visibility Filter: Common Android container elements expose elements in their accessibility API structure that are outside the bounds of the screen (e.g., *WebView*), so we transform the representation by filtering to include only visible elements (i.e., elements with *boundsInScreen* values that correspond to non-zero area within the screen). This heuristic is intended to reduce *FalseSame* errors.
7. Compare *ViewIdResourceName*: This stable property of each element will not change when an element's content is modified. If the set of *ViewIdResourceName* values are not the same, the screens are considered different. This heuristic is the primary comparison based on any transformations applied in the previous heuristics.
8. Compare *ClassName*: As with *ViewIdResourceName*, this stable property will not change when an element's content is modified. We consider this additional stable property to helps address situations where *ViewIdResourceName* is not informative. If the set of *ClassName* values are not the same, the screens are considered different. This heuristic is intended to reduce *FalseSame* errors.

Our evaluation shows these heuristics are highly effective, and they can be extended as additional data suggests new heuristics. However, any approach will sometimes require correction by the developer of a repair. For a *FalseSame* error, a developer can write an element selector that differentiates the two screens. Any future screens that match the original template will then be separated into two templates based on whether they match the selector. For a *FalseDifferent* error, a developer combines the two template screens and their variations. Any future screens will be considered equivalent if they match either of the original templates. Although we have not found it necessary, we note that multiple such corrections could be composed as needed.

Annotation Storage

The tasks of inspecting, editing, and using annotations require: 1) collections of template screens, each including a screen image, associated accessibility data, and a screen identifier used for referencing that template screen, 2) variations associated with each template screen, 3) element

identifiers for each element in each template screen, and 4) annotations defined as a combination of a screen identifier, an element identifier, and the annotation metadata to be associated with that element of that screen. Our current implementation stores this data in Google's Firebase.

DATA COLLECTION AND ANNOTATION TOOLS

Our core methods for screen identifiers, element identifiers, and screen equivalence can be applied in a variety of tools. We have created an initial set of tools to support development of repairs based on these methods. This section introduces our current tools and briefly discusses potential alternatives.

Capture Tool

Implemented as an Android accessibility service, this tool runs in the background to allow a developer to capture screens. A developer browses to a screen they want to capture, then presses a software button on the navigation bar. The tool plays a confirmation sound, captures a screen image with associated accessibility data, and uploads them to the database. The capture tool therefore requires screenshot permission, but our runtime tools do not (i.e., captured images are used only used to support annotation and our runtime tools do not use pixel-level data). If a developer wants to capture an app that has disabled screenshot permission, they can use a rooted device or emulator [45]. Although a requirement to root a device is inappropriate for end-user accessibility tools, it is more appropriate for a developer and is the only method to circumvent FLAG_SECURE. Typical capture will include a developer navigating through an app, using the tool to capture different screens, interacting with the app, and capturing variations of screens.

Template Screen Tool

This web application supports a developer inspecting and potentially correcting identified template screens in each app. Images of template screens are shown in the top row, with any variations shown in a column underneath each template. Template screens and their variations are automatically and reliably identified using screen equivalence heuristics, so the tool is primarily used to inspect the results, obtain screen identifiers, make occasional corrections, and access the annotation tool by clicking into a screen. If a correction is needed, the tool supports authoring a selector or combining templates (i.e., as discussed in Screen Equivalence Heuristics).

Annotation Tool

This web application supports a developer in authoring annotations on a template screen. It is currently accessed by clicking a screen image in the template screen tool. The tool shows the screen image with its screen identifier and uses captured accessibility API data to highlight elements when a developer clicks on them. Developers can also author a custom selector and receive feedback through highlighting one or more elements. For each highlighted element, its identifier and properties are shown in a list. An annotation can be authored as JSON-formatted metadata, or a developer can extend the annotation tool with custom functionality for a particular class of annotation (e.g., as with customized annotation interfaces developed in our later case studies).

Runtime Library

Our runtime library supports annotation-based accessibility services by providing key functions for obtaining accessibility data, identifying a screen by comparing it to a library of templates, identifying elements in a screen, and retrieving annotations. The library also supports listening for *ViewClicked* and *WindowStateChanged* events, which can lead to a change of screen structure requiring identification of the new screen. Our library therefore supports overall management of relevant annotations, allowing a developer to focus on the functionality of their accessibility repair service.

Alternative Collection and Annotation Tools

Our current tools support an end-to-end annotation process for developers, chosen as a first primary audience as we develop tools based on this approach to annotation. We envision future research exploring complementary approaches.

For example, an extension of our tools might support end-user capture and annotation directly on their phone (e.g., requiring screenshot permission during capture, but allowing end-users to directly collect and annotate data for a repair). Future research might also examine how to scale annotation, perhaps drawing upon crowdsourcing and friendsourcing techniques developed in other contexts (e.g., [35,40,41]). Our approach to screen equivalence could be included in tools for automated exploration of mobile apps (e.g., [2,21,27]), and such tools could benefit the capture of data for accessibility repair.

EVALUATION OF SCREEN EQUIVALENCE HEURISTICS

To evaluate the effectiveness of our current screen equivalence heuristics, we recruited 5 developer participants to capture screens and identify templates in a dataset of real-world mobile apps. Our sample of mobile apps was 5 top free apps in each of 10 categories. 5 participants were recruited from our department, as our primary criterion was to recruit experienced developers familiar with mobile apps.

Each session began with simple training, showing participants how to capture a screen and how to use the template screen tool to examine identification of template screens in an app. We then asked each participant to capture screens for all of the major features in 10 apps, and if possible to capture one or more variations for each screen. After completing capture for each app, the participant was asked to use the template screen tool to examine the identification of template screens in their capture of that app and to correct any errors. Because our focus was on data collection, participants used a simplified version of the tool that allowed dragging screens to re-arrange them, without a need to identify a selector that could allow the templates to be used with our runtime tools. When a participant completed capture and identification of template screens for the 10 assigned apps, we asked them to examine template screens in another 10 apps captured by other participants. We therefore obtained 2 developer judgments regarding the template screens and variations within each app, and the lead researcher resolved the limited number of disagreements (a total of 12 disagreements in 9 apps). Participants were compensated with a $20 gift card. Data collection took about 5 to 10 minutes for each app.

Participants collected a total of 2,038 screens from 50 apps. Following the same procedure used in [7], we examine equivalence in the 42,504 pairs of screens that result from considering all pairs within each app. Table 1 summarizes the improvement associated with each heuristic. Because our primary heuristic compares values of *ViewIdResourceName*, we report the effectiveness of other heuristics in terms of improvement relative to this. Considering only *ViewIdResourceName* in our dataset results in a *FalseSame* error rate of 3.10% and a *FalseDifferent* error rate of 2.28%. Adding each heuristic reduces these, and the comparison of *ViewIdResourceName* following all previous heuristics results in a *FalseSame* error rate of 0.44% and a *FalseDifferent* error rate of 0.83%. Comparison of *ClassName* then further reduces the *FalseSame* error rate to 0.09% while slightly increasing the *FalseDifferent* error rate to 0.92%.

Overall this is a 97% reduction in *FalseSame* error with a 60% reduction in *FalseDifferent* error, consistent with our goal of prioritizing the minimization of *FalseSame* errors (i.e., as discussed when introducing our screen equivalence heuristics). Remaining errors can also generally be addressed by the developer of a repair (i.e., specifying a selector or merging templates), a capability lacking from prior approaches to screen equivalence (e.g., [7,21,26]). Error rates are well below the 6% *FalseSame* and 3% *FalseDifferent* error rates in [7], though care must be taken in comparing these rates because those numbers are based on a different and much smaller dataset (i.e., 1044 pairs of screens from 12 apps used to tune the equivalence thresholds used in that work). Robust screen identifiers should also allow a developer to author an element identifier for any element in a screen. In contrast, we find the TalkBack screen reader's reliance on *ViewIdResourceName* will allow it to apply a custom label to only 13.6% of TalkBack-visited elements in this data.

Examining this data, we observe a practice of obfuscating *ViewIdResourceName*. For example, the Facebook Messenger app sets *ViewIdResourceName* to "name_removed" for all of its elements. Considering only *ViewIdResourceName* results in 84 *FalseSame* errors in this app, while our heuristics use *ActivityName*, interface structure, and *ClassName* to reduce this to only 2 *FalseSame* errors (which could then be corrected by developer specification of an appropriate selector). We also note approaches based entirely on *ViewIdResourceName*, including the TalkBack screen reader's support for adding custom labels, will be completely ineffective in such an app (i.e., because all elements have the same *ViewIdResourceName).*

Heuristic 8 reduces *FalseSame* error by checking *ClassName*, but slightly increases *FalseDifferent* error. Examining this, we find that advertising banners are a common cause of increased *FalseDifferent* error. For example, the Abs Workout app includes advertising elements that have different *ClassNames* before and after an advertisement is loaded. This suggests a future heuristic might filter advertising elements, perhaps by blacklisting their *ClassNames*.

	Heuristic	Error Rate (%) *FalseSame*	*FalseDiff*
-	Only *ViewIdResourceName*	3.10	2.28
1	*ActivityName*	2.51	2.28
2	Navigation Drawer	2.51	1.06
3	Floating Dialog	2.51	0.83
4	Tab Layout	1.55	0.83
5	Radio Button Group	1.48	0.83
6	Visibility Filter	0.44	0.83
7	*ViewIdResourceName*	as above	as above
8	*ClassName*	0.09	0.92

Table 1. Improvements in error rates resulting from the addition of each of our current screen equivalence heuristics.

We also observe a small number of cases that likely cannot be resolved using our current techniques due to an app's complete failure to implement a meaningful representation via the accessibility APIs. For example, the TopBuzz app includes a custom-implemented tab layout that does not expose any indication of what tab is active (e.g., nothing like the *Selected* or *Checked* properties in our current heuristics). It also does not properly expose elements of all tabs to the accessibility APIs, but instead exposes contents of the first tab regardless of which tab is currently active. Resolving such a complete failure may require pixel-based techniques (e.g., as in [8,9,10,11]). Although this will require screenshot permission at runtime, performance implications might be addressed by limiting pixel-based analysis to only such special-case scenarios where accessibility data fails.

CASE STUDIES OF RUNTIME ACCESSIBILITY REPAIR

This section demonstrates repair of three common types of accessibility issues, all implemented using our approach to robust annotation. These case studies are implemented using interaction proxy techniques, and correspond to prior proof-of-concept demonstrations in that research [47]. However, it was previously infeasible to scale demonstrations beyond a handful of elements in a handful of apps, due to a lack of methods for determining where and how to apply a runtime repair. Integrating annotation-based techniques into these demonstrations is an important step toward runtime accessibility repair in mobile apps, which the next section further examines in a set of 26 real-world apps.

Missing or Misleading Labels

As illustrated in Figure 1, many apps contain both unlabeled elements (e.g., elements lacking a *ContentDescription* that will therefore be read as "unlabeled") and elements with misleading labels (e.g., Figure 1's two buttons labeled "15").

We implemented an accessibility service that uses annotations to repair elements with missing or misleading values of *ContentDescription*. A developer uses the annotation tool to identify an element in need of label repair (e.g., by clicking it in the image, by writing a custom selector), then uses a text field to enter an appropriate *ContentDescription*, which the tool stores as an annotation. At runtime, the accessibility

services detect whether the current screen includes any annotations, then uses interaction proxy techniques to repair how annotated elements are read by the screen reader.

Navigation Order Issues

The navigation order of interface elements is important to many people (e.g., a person using swipe gestures to navigate interface elements with a screen reader, a person using a switch interface), but many apps have navigation orders that can make them difficult to use. For example, the navigation order for the Dropbox app begins with the "add" button and then requires navigating through all files in the current folder (i.e., a list of arbitrary length) before accessing the "menu", "select", or "more" buttons.

We implemented an accessibility service that uses annotations to repair navigation order within a screen. A customized annotation interface shows the current navigation order and allows developers to modify the order by moving elements in a list. The resulting navigation order is stored as an annotation associated with the screen, which our accessibility service detects at runtime and uses to correct the navigation order.

Inaccessible Customized Widgets

Whenever a developer creates a custom interface element, they also need to write additional code to expose appropriate accessibility hierarchy and context [15]. Unfortunately, many developers fail to do this, so these custom elements can be difficult or impossible to use with an accessibility service. For example, custom rating widgets found in Yelp and many other apps are often inaccessible (e.g., the Yelp rating widget is exposed as a *TextView* and does not allow a person using a screen reader or switch interface to enter a rating).

We implemented an accessibility service that uses annotations to repair some forms of inaccessible customized widgets. Figure 2 illustrates our enhancement of the annotation tool that supports rubberband selection to define clickable areas within an element, storing a list of these areas with a *ContentDescription* for each as an annotation on the element. At runtime, the accessibility service uses these annotations to create the missing accessibility API representations. This approach can only repair relatively simple custom elements, but also suggests an approach to more sophisticated repairs.

EVALUATION OF RUNTIME REPAIR

Our case studies build upon prior demonstrations of accessibility repairs that have received positive feedback, including feedback in two rounds of studies with 14 people with visual impairments who use screen readers [47]. The end-user experience with repair is the same as in this prior research, but prior demonstrations were limited to a handful of apps chosen by the research team and custom code for each repair. Our current evaluation therefore focused on examining the application of our selected categories of repair to accessibility issues in real-world apps. We first worked with a participant who uses a screen reader, repairing accessibility issues they identified as problematic. We then collected and repaired issues in a larger set of apps.

Participant Feedback on Accessibility Repairs

To gather initial feedback on accessibility repairs implemented in our case studies, we interviewed a person who is blind and has used an Android screen reader for 6 years. Via email prior to the interview, we described the three types of accessibility issues addressed in our case studies and asked if he found these issues in apps he frequently used. He replied to report issues in 6 apps. We then spent an hour capturing screens and authoring annotations to repair the accessibility issues he reported, followed by an additional hour examining the apps to find and repair issues he had not mentioned. We note the runtime repair of accessibility issues in 6 different apps would be infeasible in prior approaches requiring custom code to repair to specific elements in specific apps (e.g., prior repair demonstration in interaction proxies [47]).

During the interview, we first asked the participant to show how he normally used each app and how it was inaccessible. We then enabled our accessibility repair service and asked him to revisit the interactions he had showed us. After he experienced all repairs to the accessibility issues he reported, we disabled our repair service and guided him to screens with additional accessibility issues he had not mentioned. We then re-enabled the repair service, so he could experience the difference. After each app, we asked him: 1) to what extent the accessibility issues are a barrier; 2) if a repair service would change how he uses the app; 3) whether the repair service addressed all accessibility issues he mentioned. At the end of the interview, we asked for his overall opinion and thoughts regarding our approach and its potential.

Overall the participant expressed frustration with accessibility issues: "*These (accessibility) barriers make me not want to use them (apps). I'm a customer, just happened to be blind, but I'd like to use these services.*" He confirmed our repairs addressed the issues he reported, as well as additional issues in the same apps. He described how repairs would change how he uses apps, and might help more people: "*Having the annotation available and making the app accessible make me more likely to use the app. I'd like to be able to use more stuff and do more. Enhancing (the apps) to be more usable and accessible…that makes it better for everybody.*"

One app he identified was BECU (i.e., a local credit union). The app is implemented with *cocos2dx*, a game engine that was probably chosen for its ability provide high-quality animations. It unfortunately exposes very limited information to the Android accessibility APIs. On the login screen, TalkBack cannot navigate focus to the input fields for the account name or password. This app did include support for Android's voice assistant, which speaks a list of available options (e.g., "enter the password"), then requires double tapping and speaking an option. The participant objected that this solution did not meet accessibility expectations: "*that's not what I want, and it is not the way it should be working…I should just be able to double tap on the username and type it*". He also noted that speaking introduces privacy concerns: "*I often wear headphones and (keep) the screen off so that nobody could hear what's*

going on". We repaired the inaccessible login screen by defining a clickable area and defining a description for each input field. We did not continue repair beyond the login screen, both because we did not have credentials to use during capture and because we did not want to ask the participant to expose his personal information in testing.

The participant also identified the At Bat app, which features listening to live streams of baseball games. However, after paying for a subscription, the participant found he could not access the streams. The "play" button is unlabeled, and a feed source must be selected to enable the unlabeled "play" button. Without instruction, this interaction is extremely difficult for a person using a screen reader. The participant was frustrated by the player: "*it's a big barrier that I am not able to really use that app, it makes me frustrated and I don't understand why they are unlabeled…I don't want to open some random buttons*". We annotated the unlabeled buttons with appropriate labels, repaired the navigation order to more easily move to the audio player, and added an instruction to select a feed. The participant described how these repairs would make the app useful: "*I would actually use it and I paid for it…Right now, I'm not using it at all.*"

Repairs in Additional Mobile Apps

As a complement to our in-depth exploration with the above participant, we made repairs in 20 additional apps. 10 were identified as having accessibility issues by participants in accessibility-related forums [16,23,38], and 10 were selected from the top downloaded apps in the Google Play Store.

Details regarding the accessibility issues we repaired in a total of 26 apps are available at https://github.com/appaccess. Across 24 apps, we found and repaired 115 labels that were missing and 46 labels that were misleading. Across 18 apps, we found and repaired 29 navigation order issues. Across 11 apps, we found and repaired 12 inaccessible custom widgets. We include examples of these repairs in the supplementary video.

Because runtime repair of mobile accessibility issues is a relatively new capability (e.g., [32,47]), and because prior methods have required custom code specific to each repair, we believe this is the largest existing set of runtime repairs of mobile app accessibility issues, thereby providing support for the potential of annotation-based accessibility repair.

CURRENT LIMITATIONS

Our evaluation found that a relatively small number of apps expose an accessibility API representation that fundamentally lacks vital information (e.g., screens in TopBuzz on page 8, the BECU app on page 9). Our current screen equivalence heuristics cannot be effective in such circumstances. Careful authoring of selectors based on the available information might allow a motivated developer to differentiate screens and author repairs, but other approaches may also be beneficial. For example, we have overall avoided pixel-based analysis, but might make limited use of such techniques in situations like these which cannot otherwise be addressed.

Currently, we examine capture and annotation of an app within a single version of that app running on a single phone (i.e., at a single screen resolution and in portrait orientation). We are not aware of any prior research in screen equivalence that has addressed this limitation, but future research toward large-scale deployment of annotation-based repair will need to consider different versions and renderings of the same app. Our approaches should be promising, as they do not rely upon element location or size and large-scale changes can likely be modeled as additional template screens. Scrolling, animation, and dynamic introduction of new elements are also classic difficulties in runtime interpretation and modification. Our runtime tools currently address this by identifying a screen when it first appears, then monitoring events that might indicate a change in the active screen. This has been effective, but additional approaches may be necessary.

Our current implementation is for Android. Although it is not the most popular mobile platform among people who use screen readers, its open platform both enables our annotation techniques and allows advances to be directly deployed in accessibility services. Our overall strategy (i.e., identifying components and patterns that lead to screen equivalence errors) is likely to generalize to additional mobile platforms.

CONCLUSION

This paper has introduced an approach to robust annotation of mobile apps, using techniques appropriate for runtime accessibility repair. We have presented our underlying methods in terms of screen identifiers, element identifiers, and screen equivalence heuristics. We have developed an initial set of tools based on these methods, focused on developer implementation of accessibility repair services. We then evaluated our screen equivalence heuristics, presented our case studies applying annotation in runtime accessibility repair, and examined these case study implementations in repairing real-world accessibility issues. Supporting materials (*e.g.,* code and screen data) are available at: https://github.com/appaccess.

We have demonstrated an initial set of annotation tools, but there are many more possibilities. For example, our approach might be integrated directly into Android's core accessibility services (i.e., the TalkBack screen reader and Switch Access). Annotation could address limitations of these tools in relying upon *ViewIdResourceName*. Future research could also explore tools that do not require developer-level expertise, including crowdsourcing or friendsourcing approaches. Robust approaches to mobile app screen equivalence and annotation can also have applications beyond accessibility, including interface testing, large-scale collection and analysis of mobile apps [7,21,33], and task automation [25]. Overall, we believe many new tools can be developed using the underlying methods developed in this initial research.

ACKNOWLEDGEMENTS

This research was funded in part by the National Science Foundation, under award IIS-1702751 and through a Graduate Research Fellowship, and by a Google Faculty Award.

REFERENCES

1. Appium. Appium. http://appium.io/
2. Tanzirul Azim and Iulian Neamtiu. (2013). Targeted and Depth-first Exploration for Systematic Testing of Android Apps. *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2013)*, 641–660. http://doi.org/10.1145/2544173.2509549
3. Jeffrey P. Bigham and Richard E. Ladner. (2007). AccessMonkey: A Collaborative Scripting Framework for Web Users and Developers. *Proceedings of the Web for All Conference (W4A 2007)*, 25–34. http://doi.org/10.1145/1243441.1243452
4. Nilton Bila, Troy Ronda, Iqbal Mohomed, Khai N Truong, and Eyal De Lara. (2007). PageTailor: Reusable End-User Customization for the Mobile Web. *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys 2007)*. http://doi.org/10.1145/1247660.1247666
5. Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. (2005). Automation and Customization of Rendered Web Pages. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2005)*. http://doi.org/10.1145/1095034.1095062
6. Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. (2011). Associating the Visual Representation of User Interfaces with Their Internal Structures and Metadata. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2011)*, 245–256. http://doi.org/10.1145/2047196.2047228
7. Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. (2017). Rico: A Mobile App Dataset for Building Data-Driven Design Applications. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2017)*, 845–854. http://doi.org/10.1145/3126594.3126651
8. Morgan Dixon and James Fogarty. (2010). Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2010)*, 1525–1534. http://doi.org/10.1145/1753326.1753554
9. Morgan Dixon, Gierad Laput, and James Fogarty. (2014). Pixel-Based Methods for Widget State and Style in a Runtime Implementation of Sliding Widgets. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2014)*, 2231–2240. http://doi.org/10.1145/2556288.2556979
10. Morgan Dixon, Daniel Leventhal, and James Fogarty. (2011). Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2011)*, 969–978. http://doi.org/10.1145/1978942.1979086
11. Morgan Dixon, A. Conrad Nied, and James Fogarty. (2014). Prefab Layers and Prefab Annotations: Extensible Pixel-Based Interpretation of Graphical Interfaces. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2014)*, 221–230. http://doi.org/10.1145/2642918.2647412
12. James R Eagan, Michel Beaudouin-Lafon, and Wendy E Mackay. (2011). Cracking the Cocoa Nut: User Interface Programming at Runtime. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2011)*, 225–234. http://doi.org/10.1145/2047196.2047226
13. W. Keith Edwards, Ian Smith, Scott E. Hudson, Joshua Marinacci, Roy Rodenstein, and Thomas Rodriguez. (1997). Systematic Output Modification in a 2D User Interface Toolkit. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 1997)*, 151–158. http://doi.org/10.1145/263407.263537
14. Google. Accessibility Scanner. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor
15. Google. Building Accessible Custom Views. https://developer.android.com/guide/topics/ui/accessibility/custom-views.html#virtual-hierarchy
16. Google. Eyes-free Forum. https://groups.google.com/forum/#!forum/eyes-free
17. Google. Making Apps More Accessible. https://developer.android.com/guide/topics/ui/accessibility/apps.html#touch-targets
18. Google. Monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html
19. Google. UiSelector. https://developer.android.com/reference/android/support/test/uiautomator/UiSelector.html
20. Tovi Grossman, Tovi Grossman, Ravin Balakrishnan, and Ravin Balakrishnan. (2005). The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2005)*, 281–290. http://doi.org/10.1145/1054972.1055012

21. Shuai Hao, Bin Liu, Suman Nath, William G J Halfond, and Ramesh Govindan. (2014). PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys 2014)*, 204–217. http://doi.org/10.1145/2594368.2594390

22. Yun Huang, Brian Dobreski, Bijay Bhaskar Deo, Jiahang Xin, Natã Miccael Barbosa, Yang Wang, and Jeffrey P. Bigham. (2015). CAN: Composable Accessibility Infrastructure via Data-Driven Crowdsourcing. *Proceedings of the Web for All Conference (W4A 2015)*, 2. http://doi.org/10.1145/2745555.2746651

23. InclusiveAndroid. App and Game Categories. https://www.inclusiveandroid.com/?q=app-and-game-categories

24. Shinya Kawanaka, Yevgen Borodin, Jeffrey P. Bigham, Darren Lunn, Hironobu Takagi, and Chieko Asakawa. (2008). Accessibility Commons: A Metadata Infrastructure for Web Accessibility. *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2008)*, 153–160. http://doi.org/10.1145/1414471.1414500

25. Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. (2017). SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2017)*, 6038–6049. http://doi.org/10.1145/3025453.3025483

26. Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. (2014). DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI 2014)*, 57–70.

27. Ke Mao, Mark Harman, and Yue Jia. (2016). Sapienz: Multi-objective Automated Testing for Android Applications. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2016)*, 94–105. http://doi.org/10.1145/2931037.2931054

28. Michael Nebeling, Maximilian Speicher, and Mc Norrie. (2013). CrowdAdapt: Enabling Crowdsourced Web Page Adaptation for Individual Viewing Conditions and Preferences. *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing System (EICS 2013)*. http://doi.org/10.1145/2480296.2480304

29. Dan R Olsen Jr., Scott E Hudson, Thorn Verratti, Jeremy M Heiner, and Matt Phelps. (1999). Implementing Interface Attachments Representations Based on Surface. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 1999)*, 191–198. http://doi.org/10.1145/302979.303038

30. World Health Organization. (2011). World Report on Disability. http://www.who.int/disabilities/world_report/2011/en/

31. André Rodrigues. (2015). Breaking Barriers with Assistive Macros. *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2015)*, 351–352. http://doi.org/10.1145/2700648.2811322

32. André Rodrigues and Tiago Guerreiro. (2014). SWAT: Mobile System-Wide Assistive Technologies. *Proceedings of the 28th International BCS Human Computer Interaction Conference on HCI 2014-Sand, Sea and Sky-Holiday HCI*, 341–346.

33. Anne S. Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. (2017). Epidemiology as a Framework for Large-Scale Mobile Application Accessibility Assessment. *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2017)*, 2–11. http://doi.org/10.1145/3132525.3132547

34. Daisuke Sato, Masatomo Kobayashi, Hironobu Takagi, and Chieko Asakawa. (2009). What's Next? A Visual Editor for Correcting Reading Order. In *Proceedings of the International Conference on Human-Computer Interaction (INTERACT 2009)*, Tom Gross, Jan Gulliksen, Paula Kotzé, Lars Oestreicher, Philippe Palanque, Raquel Oliveira Prates and Marco Winckler (eds.). Berlin, Heidelberg, 364–377. http://doi.org/10.1007/978-3-642-03655-2_41

35. Daisuke Sato, Hironobu Takagi, Masatomo Kobayashi, Shinya Kawanaka, Chieko Asakawa, and Asakawa Chieko. (2010). Exploratory Analysis of Collaborative Web Accessibility Improvement. *ACM Transactions on Accessible Computing (TACCESS)*, *3*(2), 5. http://doi.org/10.1145/1857920.1857922

36. Selendroid. Selendroid. http://selendroid.io/

37. Square. Flow Github Repository. https://github.com/square/flow

38. StrangelyTyped. My Brief Experiences with Android Talkback/Accessibility. https://www.reddit.com/r/Android/comments/3uqs6z/my_brief_experiences_with_android/

39. Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. (2006). User Interface Facades: Towards Fully Adaptable User Interfaces. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2006)*, 309–318. http://doi.org/10.1145/1166253.1166301

40. Hironobu Takagi, Shinya Kawanaka, Masatomo Kobayashi, Takashi Itoh, and Chieko Asakawa. (2008). Social Accessibility: Achieving Accessibility Through Collaborative Metadata Authoring. *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2008)*, 193–200. http://doi.org/10.1145/1414471.1414507

41. Hironobu Takagi, Shinya Kawanaka, Masatomo Kobayashi, Daisuke Sato, and Chieko Asakawa. (2009). Collaborative Web Accessibility Improvement: Challenges and Possibilities. *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2009)*, 195–202. http://doi.org/10.1145/1639642.163967

42. Desney S Tan, Brian Meyers, and Mary Czerwinski. (2004). WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI'EA 2004)*, 1525–1528. http://doi.org/10.1145/985921.986106

43. Mosaic Design Team. (1993). Group Annotations in NCSA Mosaic. https://www.math.utah.edu/~beebe/support/html/Docs/group-annotations.html

44. W3C. W3C Web Annotation Working Group. https://www.w3.org/annotation/

45. Xposed. DisableFlagSecure. http://repo.xposed.info/module/fi.veetipaananen.android.disableflagsecure

46. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. (2009). Sikuli: Using GUI Screenshots for Search and Automation. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2009)*, 183–192. http://doi.org/10.1145/1622176.1622213

47. Xiaoyi Zhang, Anne Ross, Anat Caspi, James Fogarty, and Jacob O. Wobbrock. (2017). Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2017)*, 6024–6037. http://doi.org/10.1145/3025453.3025846

48. Yu Zhong, Astrid Weber, Casey Burkhardt, Phil Weaver, and Jeffrey P. Bigham. (2015). Enhancing Android Accessibility for Users with Hand Tremor by Reducing Fine Pointing and Steady Tapping. *Proceedings of the Web for All Conference (W4A 2015)*, 29. http://doi.org/10.1145/2745555.2747277