Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android

Navid Salehnamadi
School of Information and Computer Sciences
University of California, Irvine, USA
nsalehna@uci.edu

Abdulaziz Alshayban
School of Information and Computer Sciences
University of California, Irvine, USA
aalshayb@uci.edu

Jun-Wei Lin
School of Information and Computer Sciences
University of California, Irvine, USA
junwel1@uci.edu

Iftekhar Ahmed
School of Information and Computer Sciences
University of California, Irvine, USA
iftekha@uci.edu

Stacy Branham
School of Information and Computer Sciences
University of California, Irvine, USA
sbranham@uci.edu

Sam Malek
School of Information and Computer Sciences
University of California, Irvine, USA
malek@uci.edu

ABSTRACT

For 15% of the world population with disabilities, accessibility is arguably the most critical software quality attribute. The ever-growing reliance of users with disability on mobile apps further underscores the need for accessible software in this domain. Existing automated accessibility assessment techniques primarily aim to detect violations of predefined guidelines, thereby produce a massive amount of accessibility warnings that often overlook the way software is actually used by users with disability. This paper presents a novel, high-fidelity form of accessibility testing for Android apps, called Latte, that automatically reuses tests written to evaluate an app's functional correctness to assess its accessibility as well. Latte first extracts the use case corresponding to each test, and then executes each use case in the way disabled users would, i.e., using assistive services. Our empirical evaluation on real-world Android apps demonstrates Latte's effectiveness in detecting substantially more useful defects than prior techniques.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Human-centered computing** → **Accessibility design and evaluation methods**.

KEYWORDS

Accessibility, Automated Testing, Mobile Application

ACM Reference Format:
Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. 2021. Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3411764.3445455

CHI '21, May 8–13, 2021, Yokohama, Japan

ACM ISBN 978-1-4503-8096-6/21/05.
https://doi.org/10.1145/3411764.3445455

1 INTRODUCTION

Mobile applications (apps) are permeating every aspect of the daily life of billions of people around the world, from personal banking to communication, transportation, and more. The ability to access and use these apps with ease is vital for everyone, especially for approximately 15% of the world population with some form of disability [27]. However, recent studies have shown accessibility issues are prevalent in mobile apps, hindering their use by users with disability [1, 15, 23].

To date, various automated accessibility analysis techniques have been proposed to deal with the widespread prevalence of accessibility issues [3, 5, 12, 13]. Common across all these tools is the way they aim to identify accessibility issues in terms of predefined rules derived from accessibility guidelines. For instance, whether a label for an icon is missing, whether there is sufficient contrast between text and background, whether the actionable elements are too close to each other, etc. While it is important for developers to follow these guidelines in the construction of their apps, the rules by themselves are not able to precisely determine the difficulties a user with disability may experience. For example, from a disabled user's standpoint, there is a significant difference between accessibility issues affecting the main functionalities of an app versus those affecting its incidental functionalities (e.g., advertisement banners, copyright disclaimers), yet the existing techniques provide no effective means of distinguishing between the two. Prior studies [1] have shown the developers tend to either not utilize or simply ignore the results of existing accessibility analysis tools, because they produce a massive amount of accessibility warnings, many of which are minor, or simply wrong. A user study on web accessibility [21] indicates that half of the problems that people with disabilities encounter are not covered by Web Content Accessibility Guidelines [26].

Another limitation of the existing automated accessibility analysis techniques is that none consider the assistive services such as *TalkBack* (a screen reader for Android users with blindness or visual impairment) or *SwitchAccess* (an Android service for navigating app for users with motor impairment) in their analysis. Since disabled users are heavily reliant on assistive services in interacting with apps, many important cues are missed when these services are not considered in the evaluation of an app's accessibility. For instance, a screen with a dynamic user interface (UI) may have no apparent accessibility

issue in the implementation of its individual elements, yet be completely unusable by a disabled user due to the assistive technology's inability to detect the changes in UI. As another example, a screen with a revolving list of items in one panel may have no accessibility issue in its implementation according to existing guidelines, yet prevent a disabled user from reaching another panel on that screen using the commonly available assistive technologies.

The key insights that guide our research are that (1) the focus of automated accessibility analysis should be on the main functionalities of an app, and not some incidental features, such as displayed ads, and (2) a high-fidelity form of analysis needs to reflect the way disabled users actually interact with apps, i.e., using the assistive technologies.

Informed by the above-mentioned insight, we have developed a new form of automated accessibility analysis, called Latte, that builds on the way developers already validate their apps for functional correctness. A widely adopted practice in software development is for developers to write system tests, often in the form of Graphical User Interface (GUI) tests, to validate the important use cases (functionalities) of an app for correctness. These use cases are the important functionalities of an app that should also be accessible. Given an app under test and a set of regular GUI tests (written by developers) as input, Latte first extracts a *Use-Case Specification* corresponding to each test. A Use-Case Specification defines the human-perceivable steps a test takes to exercise a particular functionality in an app. Latte then executes the Use-Case Specification using an assistive service, i.e., TalkBack and SwitchAccess. If a use case cannot be completed using an assistive service, it naturally means the corresponding use case has an accessibility problem, which is reported to the developer.

Latte mitigates the limitations of existing automated accessibility analysis techniques by evaluating the accessibility issues in a more realistic setting, i.e., using assistive services. In more than half of the subjects apps in our experiments, Latte detected accessibility issues that were not detected by Google's Accessibility Scanner, the most widely used accessibility analyzer for Android. Moreover, unlike prior solutions that produce a massive number of accessibility warnings by simply scanning an app's screens irrespective of its purpose, our approach produces a small number of actionable accessibility defects that are guaranteed to affect a disabled user's proper usage of the app's main functionalities. Latte produces a detailed report for each failed use case that provides the developer with the exact cause of inaccessibility and steps to replicate it.

Although the most reliable method of validating an app's accessibility is through user evaluation, finding users with different types of disability and conducting such evaluations can be prohibitively difficult, especially for small development teams with limited resources. Using Latte, developers are able to gain useful insights into how their apps behave when engaged through an assistive service, allowing them to fix the issues prior to their release. Our approach can also complement user evaluation by allowing the development teams to hone in on a subset of problematic use cases that are flagged by our tool.

This paper makes the following contributions:

- A novel, high-fidelity form of automated accessibility analysis that evaluates the degree to which important use cases of an app can be accessed by users with disability through assistive services;
- An implementation of the above-mentioned approach for Android, called Latte, that is publicly available [24];
- An extensive empirical evaluation on real-world Android apps, demonstrating effectiveness of Latte in identifying issues that the existing automated techniques cannot detect; and
- A qualitative study of the different types of accessibility failures and warnings that can be detected using Latte.

The remainder of this paper is organized as follows. Section 2 summarizes the related work. Section 3 illustrates an accessibility issue that cannot be detected using existing automated techniques, while Section 4 describes the details of our approach. Section 5 presents our experimental evaluation. Finally, Section 6 concludes and describes our future work. The tool and experimental artifacts can be found on the companion website, https://github.com/seal-hub/Latte.

2 RELATED WORK

Accessibility analysis can be a challenging and time-consuming task, since it requires human expertise and judgment. Researchers have investigated various ways of automating the accessibility analysis process, which can be broadly categorized into two categories: static and dynamic accessibility analysis [25].

Static accessibility analysis tools analyze the screen content and configuration files to look for predefined accessibility violation rules. For example, Lint [9], which is shipped with Android Studio [7], can identify accessibility issues such as missing content descriptions, missing translation, and layout issues. Since it does not consider dynamic properties and views created at runtime, the types of accessibility issues it can detect are limited.

Dynamic analysis based techniques try to mitigate the limitations of static analysis based techniques. Google Accessibility Scanner [3] is one such technique that detects accessibility issues by analyzing the currently displayed user interface components. Though Google Accessibility Scanner can identify a larger number of accessibility issues than static accessibility analysis tools [25], one of its major limitations is that it requires the developers to manually crawl the app under test and activate the tool on each screen. Other testing frameworks like Espresso [8] and Robolectric [22] also require developers to manually specify the test cases which significantly increases developers' workload.

Since there is a large range of disabilities with varying severity, it may not be possible for a development team to manually test an app through user evaluation. Moreover, due to time and budget constraints, such manual approaches often result in insufficient evaluation [25]. Relying on manual evaluation also makes it challenging to re-evaluate new releases of apps, which may frequently occur due to short release cycles, changing requirements, and rapidly evolving technologies [14].

To overcome the mentioned limitations, Alshayban et al. [1] implemented a random crawler to automatically explore the different screens/activities of the app under test while assessing each screen for accessibility violations. However, their technique fails to explore many screens of the app due to the random nature of event generation [1]. Eler et al. developed MATE [16] to mitigate this limitation. MATE improves the exploration process by considering interactable

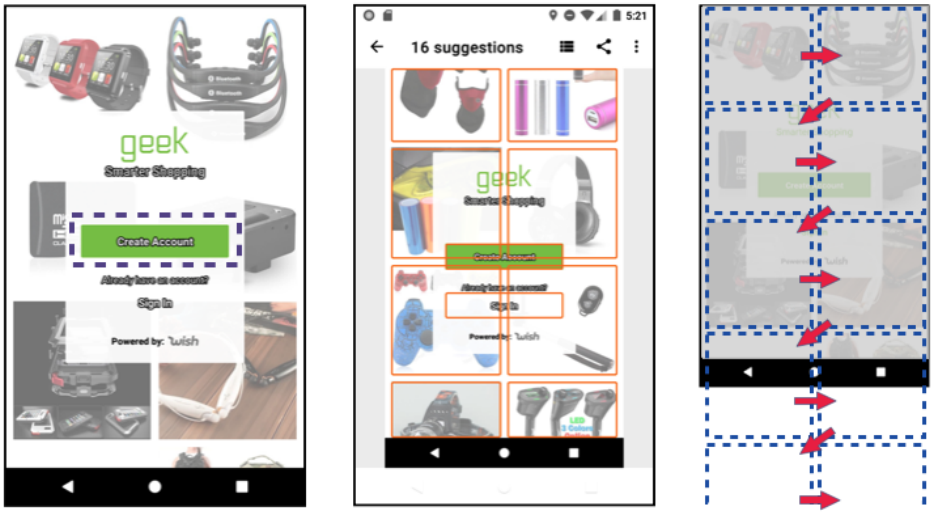


Figure 1: a) The very first step of creating account in "geek" shopping app (the dotted box) b) The accessibility issues reported by Google Accessibility Scanner c) Navigating the app using assistive services (TalkBack and SwitchAccess)

elements, e.g., a clickable button, rather than random events on random coordinates.

However, all existing accessibility evaluation techniques have two common limitations. First, none of the current techniques consider the importance of the functionality affected by accessibility issues. As a result, they fail to report the severity of the identified accessibility issues accurately. Second, none of the existing techniques consider assistive services such as *TalkBack* or *SwitchAccess* in their analysis and miss out on many important cues when evaluating an app's accessibility.

3 ILLUSTRATIVE EXAMPLE

Figure 1(a) shows the launch screen of *Geek - Smarter Shopping* app (version '2.3.7') with more than 10 millions users [18]. The foreground layout contains register and login buttons, while the background is a layout of rolling decorative images. One of the most important use cases in this app is registration, since it is the prerequisite for accessing all other functionalities. This use case starts by clicking on the *Create Account* button (the dashed box in Figure 1(a)) followed by filling a form with user information (not depicted in the figure). A developer can create a GUI test to automatically verify this use case is working. For example, Listing 1 showsww a GUI test in Appium [11] testing framework written in Python. It is basically a sequence of steps performing actions on specific elements on the screen, e.g., clicking on an element with resource-id `com.contextlogic.geek:id/login_fragment_create_account_button`.

While a user without disability interacting with the app can see the full screen and perform all gestures, a user with disability has to rely on assistive services to complete their intended task. For example, a blind user relies on TalkBack [17] to read the textual description of the elements on the screen. TalkBack gives spoken feedback and notifications to users by announcing headers, labels, icons and other

```
1 find_element_by_id("com.contextlogic.geek:id/
      login_fragment_create_account_button").click()
2 find_element_by_xpath("/android.widget.FrameLayout/.../android.
      widget.EditText[1]").send_keys("John Doe")
3 find_element_by_id("fragment_email_text").send_keys("john.
      doe@example.com")
4 find_element_by_id("fragment_password_text").send_keys("StR0nGp@ss"
      )
5 find_element_by_xpath("/android.widget.FrameLayout/.../android.
      widget.TextView[3]").click()
```

Listing 1: The test script corresponding to the registration use case

assistive content defined by developers. The user can explore the app either by reading the elements in order or touching different parts of the screen, asking TalkBack to announce the textual description of the selected element. A user with motor disability, on the other hand, uses SwitchAccess [2] to navigate the app; however, the user can see the whole screen. SwitchAccess is an assistive service that enables users to interact with the device using a special keyboard with a limited set of buttons such as *Next* and *Select*. SwitchAccess highlights the focused element on the screen. The user uses the two buttons to change the focus to next element or select the currently focused element.

While the developer of an app like this is likely to write a test to evaluate the functional correctness of registration, given its important to the overall functionality of the app, the conventional execution of such test does not reveal anything about the app's accessibility issues. To test the accessibility of this app, a conscientious developer would also run the Google Accessibility Scanner [3]—a de facto standard tool for analysis of accessibility in Android—on the launch screen and review the identified issues, as shown in Figure 1(b). In total, 16 accessibility issues are detected by the Scanner, denoted by orange borders placed around the elements with a problem. Out of these, there are 8 *"missing speakable text"* and 6 *"low image contrast"* issues for the decorative images in the background, and 2

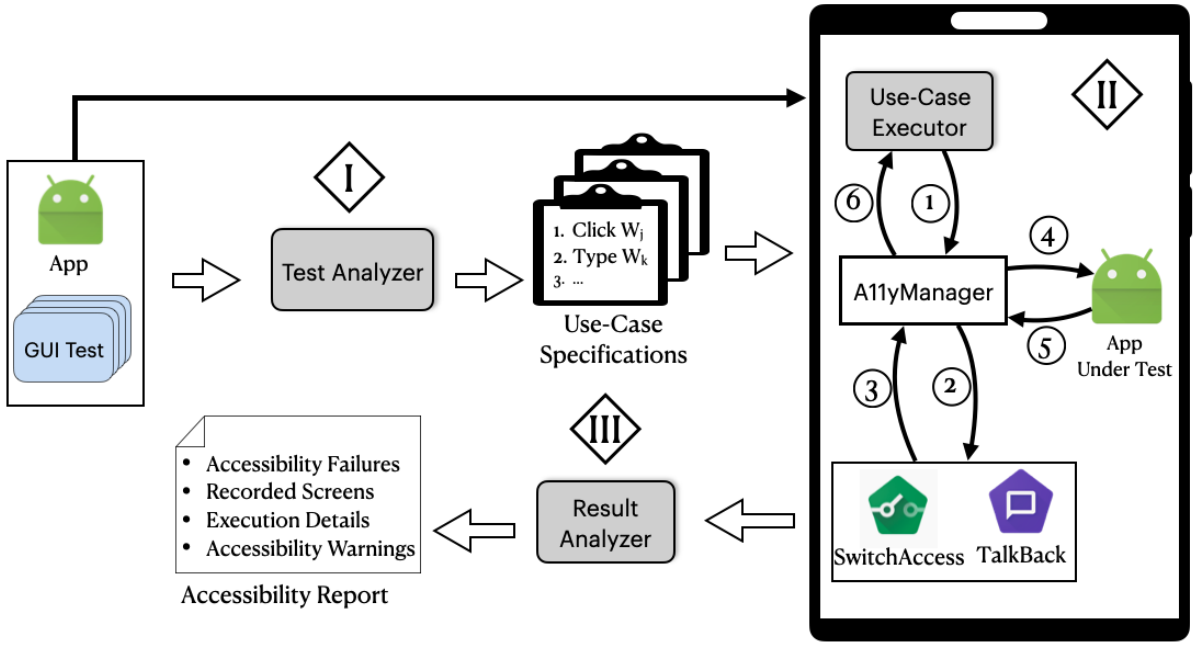


Figure 2: Overview of Latte

"small touch target size" issues for the buttons in the foreground. As can be seen, there are many issues with the very first screen, and no particular hint as to the severity of these issues is provided to help the developer prioritize the effort involved in fixing the reported issues. The only accessibility issue reported for *Create Account* button is the *"small touch target size"*, which in fact does not affect users who rely on assistive tools for their interactions. Once the reported issues are fixed, this screen becomes supposedly accessibility-issue free, according to the automated accessibility scanner.

In practice, however, when TalkBack and SwitchAccess are used to operate this app, the first decorative image in the background receives the focus (top left dotted box in Figure 1(c)). To reach the *Create Account* button, users have to navigate through the elements. But here the decorative background layout refills dynamically, i.e., it is a revolving list. As a result, the focus never reaches to the foreground layout. The navigation path taken through the use of assistive tools is depicted in Figure 1(c) as arrows. This makes it difficult, if not impossible, for both TalkBack and SwitchAccess users to reach the *Create Account* button. In some cases, it may be possible for the user to touch random spots on the screen and find the button by chance; nevertheless, it would be far from perfect and frustrating at the very least.

4 APPROACH

Our objective is to develop an automated accessibility analyzer that is use-case and assistive-service driven. Figure 2 shows an overview of our approach, Latte, consisting of three phases: (I) analysis of the provided GUI test suite of an Android app to determine the corresponding use cases, (II) execution of each use case on the app using an assistive service to evaluate the accessibility of the use case, (III) collection and analysis of the results to produce an accessibility report. In this section, we describe these phases in detail.

4.1 Test Analyzer

A use case is a sequence of interactions between a user and a software system for achieving an objective. In the case of a shopping app, for instance, creating an account, searching for a product, and purchasing a product, are examples of use case. As a common development practice, developers write GUI tests to automatically evaluate the correctness of a software system's use cases. A GUI test is a sequence of steps, where in each step, the test (1) locates a GUI element, and (2) performs an action on that element. For example, the first step (line 1) in Listing 1 locates an element with *resource-id* equal to `com.contextlogic.geek:id/login_fragment_create_account_button` and then clicks on it. GUI tests need to uniquely identify elements on the screen. They leverage the implementation details of an app, such as *resource-id*, to interact with the GUI elements of the app. A GUI test thus follows a *white-box approach*, i.e., uses the implementation details of an app to drive the execution. Although this format is quite effective for machine execution, it differs vastly from how users interact with an app. A user may exercise the same use case as a test, but follows a *black-box approach*, i.e., interacts directly with the UI elements of an app to drive the execution.

Since our objective is to evaluate the accessibility of use cases exercised by tests, we first have to extract a description of the use case in terms of constructs available to a user. For instance, while the test script is able to access a button through its programmatic identifier (i.e, *resource-id* attribute), a blind user would access it through its textual content description. The Test Analyzer component takes a GUI test as input and transforms it into a *Use-Case Specification*,

```
// Resource Id
viewIdResName: com.contextlogic.geek:id/
      login_fragment_create_account_button;
// Textual information
className: android.widget.TextView;
text: Create Account;
contentDescription: null;
// Other fields
boundsInParent: Rect(0, 0 - 498, 110);
boundsInScreen: Rect(291, 856 - 789, 966);
clickable: true;
focusable: true;
focused: false;
selected: false;
longClickable: false;
enabled: true;
importantForAccessibility: true;
...
```

Listing 2: The `AccessibilityNodeInfo` object corresponding to "Create Account" button

```
1 Click on element with Text: "Create Account", ContentDescription:
      null, Class: TextView
2 Type "John Doe" on element with Text: "Name", ContentDescription:
      null, Class: EditText
3 Type "john.doe@example.com" on element with Text: "Email",
      ContentDescription: null, Class: EditText
4 Type "StR0nGp@ss" on element with Text: "Password",
      ContentDescription: null, Class: EditText
5 Click on element with Text: null, ContentDescription: "Submit,
      Class: ImageButton
```

Listing 3: The use case corresponding to the registration test case in the illustrative example

consisting of a set of equivalent steps as those performed by the test at the level of abstraction understood by users. In other words, Use-Case Specification of a test represents the steps a user would need to perform to exercise the same functionality as that of the test.

To extract the use cases from GUI tests, we have developed a novel, dynamic program analysis technique that, given a test script and an app, determines (1) the various GUI elements involved in the test and their attributes, and (2) the actions performed on those elements. Dynamic program analysis entails evaluating a program by executing it. In fact, software testing is the most common form of dynamic program analysis. By dynamically analyzing a test script (i.e, running the test on the app), we are able to identify the `AccessibilityNodeInfo` object corresponding to each GUI element. `AccessibilityNodeInfo` class is provided by the Android framework and represents the attributes of a GUI element on the screen. For example, the `AccessibilityNodeInfo` of the element in the first step in Listing 1, "Create Account" button, can be found in Listing 2. The first field is `viewIdResName` (or *resrouce-id*) that is the identifier of the element. The textual attributes are `className`, `text`, and `contentDescription`. There are also other types of attributes such as coordinates and supported behaviors, e.g., this element is clickable, focusable, etc. We extract the textual attributes (*text*, *contentDescription*, and *className*) for each element, since these are the attributes perceived by users in locating GUI elements. Note that the *className* attribute is perceivable by users, since a sighted or blind user can recognize it visually or textually, i.e., `EditText` element has its distinguishable shape, and TalkBack announces it as *Edit Text Box*. We further extract actions (e.g., click, type) performed on the GUI elements from the test script itself. From Listing 1, we are able to determine that the use case consists of five steps, where the first and last steps click on GUI elements and the other steps enter textual information in GUI elements.

We finally combine the information obtained through the above-mentioned analysis of the GUI tests to arrive at the equivalent Use-Case Specifications. For example, Listing 3 is the Use-Case Specification generated from the GUI test shown in Listing 1. The first step shows the user clicking on a `TextView` element with the text "Create Account" and the last step is clicking on an `ImageButton` element with content description equal to "Submit". Intuitively, we have transformed a white-box description of a use case (i.e., GUI test) to a black-box description of that use case (i.e., Use-Case Specification).

The Test Analyzer component is written in Python programming language on top of the Appium testing framework [11].

4.2 Use-Case Executor

The existing testing frameworks can access all GUI elements and perform any actions on them, even if the target element is not visible to the user. For example, the first step of the test shown in Listing 1 is able to locate the "Create Account" button and click on it, no matter where the button is located on the screen. However, users with disability may not be able to perform such actions smoothly. Blind users need to explore the app using a screen reader to locate the element. Although recognizing elements is comparatively easier for users with motor disability, they may have difficulty reaching and initiating action on the element, as we saw in the illustrative example of Section 3.

To improve the fidelity of evaluating accessibility issues for users with disability, Latte is designed to automatically execute a use case using assistive services. To that end, we have provided our own implementation of `AccessibilityService`—an abstract service provided by the Android framework to assist users with disabilities. The official assistive tools in Android, such as TalkBack and SwitchAcces, are also implementations of this abstract service [4]. `AccessibilityService` can be seen as a wrapper around a mobile device that performs actions on and receives feedback from the device. Since these abilities may introduce security and privacy issues, an implementation of this service must specify the types of feedback it can receive and actions it can perform. For example, TalkBack can receive feedback about all elements on the screen, since it has `android:canRetrieveWindowContent` attribute in its specification. Moreover, it can perform actions, such as click, on elements; however, it cannot perform gesture such as swiping on the screen, since the attribute `android:canPerformGestures` does not exist in TalkBack's specification.

The feedback is delivered to accessibility services through `AccessibilityEvent` objects. An implementation of this service must define the method `onAccessibilityEvent` that is called back by `AccessibilityManager`—a system-level service that serves as an event dispatcher for `AccessibilityEvents` [4]. Accessibility events are generated when something notable happens on the user interface, e.g., an Activity starts, the focus of an element changes, etc. When a change occurs, `AccessibilityManager` passes the associated `AccessibilityEvent` object to `onAccessibilityEvent` method to interpret and provide feedback to the user. For example, in TalkBack, when an element is focused, its textual description is announced to the user. Alternatively, in SwitchAccess, the focused

element is highlighted. An AccessibilityEvent object is associated with an AccessibilityNodeInfo object that contains the element's attributes. For instance, when a user clicks on "Create Account" button (highlited in Figure 1(a)), the system creates an AccessibilityEvent of type TYPE_VIEW_CLICKED, which is associated with the AccessibilityNodeInfo object shown in Listing 2.

Although implementations of AccessibilityService, such as TalkBack and SwitchAccess, are typically used for assisting users to interact with the mobile device, these services can also be designed to cooperate with one another, as we have done here. We have developed our own implementation of AccessibilityService, called *Use-Case Executor*, that takes a Use-Case Specification as input, and sequentially executes the steps defined in it using TalkBack and SwitchAccess. Each step in the Use-Case Specification results in the execution of 6 steps in the device, as shown in phase II of Figure 2. We explain how the approach works using TalkBack below (a similar process is followed in the case of SwitchAccess):

(1) Use-Case Executor performs an action using APIs provided by AccessibilityService. For example, Listing 4 shows a code snippet from the implementation of Use-Case Executor that performs a swipe right gesture on the screen. The performed action is received by the AccessibilityManager service, *A11yManager* in short, and generates accessibility events corresponding to the action, e.g., TYPE_GESTURE_DETECTION_START and TYPE_GESTURE_DETECTION_END events for the swipe.
(2) All implementations of AccessibilityService, including TalkBack, receive the generated accessibility events. TalkBack may suppress delivering the incoming events to the app, and possibly translates them to something else. For example, while swiping right on the screen may result in a menu option to be shown, TalkBack may translate that gesture to changing the focus to the next element when TalkBack is enabled on the device.
(3) TalkBack analyzes the incoming event and initiates another action accordingly. For example, in the case of swipe right, TalkBack changes the focus to the next element, and in the case of a double tap, the currently focused element is clicked. Note that TalkBack is not aware of the sender of these events, in this case Use-Case Executor. As a result, TalkBack behaves the same as it would if a human user had performed the action.
(4) A11yManager receives the new action from TalkBack and sends it to the app under test. For example, if the TalkBack's action is clicking on the focused element, A11yManager sends an event to the onClickListener class associated with the focused element in the app. The app receives the action and updates its internal state accordingly, e.g., executing onClick method of the clicked element.
(5) The app informs A11yManager regarding the changes in the GUI elements. For example, when "Create Account" button is clicked, the current screen disappears and another screen with a form appears.
(6) A11yManager receives the changes in the layout and dispatches feedback events accordingly, e.g., a TYPE_VIEW_FOCUSED accessibility event associated with the focused element's AccessibilityNodeInfo object. As a result, Use-Case Executor is informed of the latest changes on the device. For instance, it becomes aware of the element that is currently focused. Note that there is a possibility that because of the changes caused by step 5, i.e., showing a new screen, another interaction is initiated between A11yManager and TalkBack, similar to steps 2 and 3.

```
1 Path swipePath = new Path();
2 swipePath.moveTo(x_left, y_middle);
3 swipePath.lineTo(x_right, y_middle);
4 gestureBuilder.addStroke(new GestureDescription.StrokeDescription(
      swipePath, 0, gestureDuration));
5 GestureDescription gestureDescription = gestureBuilder.build();
6 accessibilityService.dispatchGesture(gestureDescription, callback,
      null);
```

Listing 4: A code snippet from Use-Case Executor that performs a swipe right on the screen using `AccessibilityService` API (`x_left`, `x_right`, and `y_middle` are the left and right horizontal, and middle vertical coordinates on the screen)

Use-Case Executor executes the steps in the use case according to the procedure described above. Each step consists of two parts, locating (focusing) the target element, and performing the target action on it. For the first part, Latte scans the screen by sending swipe right events for TalkBack and Next button events for SwitchAccess, until the element that matches the description in the step is focused. Once the element is located (focused), Latte performs the target action, e.g., if the action is click, Latte sends a double tap event for TalkBack and Select button event for SwitchAccess.

There are two scenarios during the use-case execution where the scanning process may not finish; in other words, none of the focused elements match the description of the target element in the use-case step. First, the textual description of the element is not sufficient to uniquely recognize the element, because either there are multiple elements with the same description (duplicate labels issue) or the target element does not have any textual description (unlabeled element issue). This scenario occurs only in the case of TalkBack. The other scenario occurs when the target element could not be focused (or reached) by TalkBack or SwitchAccess, e.g., illustrative example of Section 3 in which "Create Account" button could not be reached.

Latte defines two termination conditions to prevent getting stuck in such cases: (1) if an element is visited more than a predefined number of times, or (2) if a step takes more than a predefined number of interactions to complete. These thresholds are configurable. Once either one of these conditions is satisfied, Latte marks the step as inaccessible. However, since we would like to identify all accessibility issues in a use case, and not just the first encountered issue, when an inaccessible step is encountered, Latte executes it using the corresponding instruction in the original test script. This allows Latte to continue the analysis and report all accessibility issues within a use case.

The Use-Case Executor component is implemented in Java by extending Android's AccessibilityService. We used the latest versions of TalkBack (8.2) and SwitchAcces (8.2), which were released by Google on Github in July 2020 [10].

4.3 Result Analyzer

To retrieve the information generated during use case execution automatically, we implemented a Command Line Interface (CLI) on top

of the Android Debug Bridge (ADB) [6]. Using the CLI, the Result Analyzer component communicates with the Use-Case Executor to receive and record details of the execution for each step of a use case (recall Figure 2). Moreover, it automatically records the screen during the use-case execution and stores the video clip. Once all use cases are executed, the Result Analyzer aggregates the results and generates an *Accessibility Report*, consisting of the following four components.

Accessibility Failures. For each use case, Latte reports if it encountered an accessibility failure during its execution using assistive services. A use case has an accessibility failure if the GUI element of one of its steps cannot be located (focused).

Recorded Screens. While Latte executes a use case, it records the screens to help developers (1) localize the accessibility issues, and (2) obtain insights into how users with disability may interact with their apps using assistive services.

Execution Details. Latte reports other information extracted from the execution of each use case, including the execution time and the number of interactions to complete the use case. This information can be used as a source of insight for developers.

Accessibility Warnings. If a specific use case takes an exorbitant number of interactions to complete, it indicates a usability concern for disabled users. We report this category of issues as accessibility warnings, since in practice they can adversely affect users with disability. The threshold of what constitutes an exorbitant number of interactions is configurable in Latte. For the purpose of experiments reported in the next section, we empirically observed that on average 1 direct interaction with an app requires approximately 5 times more interactions using TalkBack. We thus set the threshold to 15 times the number of direct interactions, or 3 times the average number of TalkBack interactions.

5 EVALUATION

In this section, we evaluate Latte on real-world apps to investigate the following research questions:

- **RQ1.** How accurately does Latte execute use cases using assistive services?
- **RQ2.** How does Latte compare to Google Accessibility Scanner (the official tool for detecting accessibility issues in Android)?
- **RQ3.** How do the detected accessibility failures and warnings impact the usage of apps?

5.1 Experimental Setup

We evaluated our proposed technique using 20 apps, 5 of which have known accessibility issues, as confirmed through user studies with disabled users in prior work [28]. The rest have been randomly selected from 13 different categories on Google Play (e.g., entertainment, productivity, finance), where 12 of them have more than 1 million installs.

We constructed a set of 2 to 4 test cases per app using Appium [11], which is an open-source testing framework. In total, we ended up with 50 test cases for 20 apps. The test cases reflect a sample of the apps' main use cases, as provided in their descriptions (e.g., register an account, search for products, place products in a shopping cart). For the apps with confirmed issues (first 5 apps highlighted in Table 1), one of the test cases corresponds to the previously reported use case that users with disability could not perform. Our experiments were conducted on a MacBook Pro with 2.8 GHz Core i7 CPU and 16 GB memory (a typical computer setup for development) using an Android emulator (SDK 27).

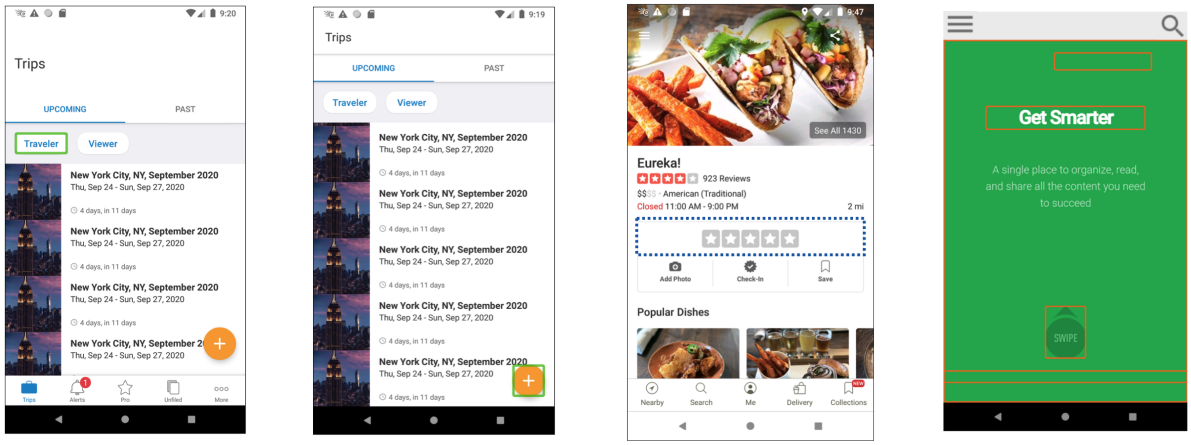
Table 1: The summary of detected accessibility failures. 'x' shows a failure was found in an app (row) while executing under a setting (column). Red bold 'x' is a failure that was detected using Latte but not using Google Accessibility Scanner. '✓' means the test or use case could be executed completely under a setting. The first five highlighted apps have confirmed accessibility issues per prior user study [28] .

	None (Test Cases)	SwitchAccess (Use Cases)	TalkBack (Use Cases)
iPlayRadio	✓✓	**xx**	**xx**
Feedly	✓✓	**xx**	**xx**
Checkout51	✓✓	✓✓	**x**✓
Yelp.	✓✓	✓✓	**xx**
Astro	✓✓	✓✓	x✓
BillManager	✓✓	✓✓	x✓
Budget	✓✓✓	**xx**✓	**xxx**
CalorieCounter	✓✓✓	**x**✓✓	**xx**x
Clock	✓✓	✓✓	xx
Cookpad	✓✓	✓✓	**x**x
Dictionary	✓✓✓	✓✓✓	**xx**x
Fuelio	✓✓✓	✓✓✓	x✓✓
Geek	✓✓	**xx**	**xx**
SchoolPlanner	✓✓✓	✓✓✓	xxx
SoundCloud	✓✓	✓✓	**x**x
TodoList	✓✓✓✓	**xx**✓✓	**xx**x✓
TripIt	✓✓✓	✓✓✓	**x**x✓
Vimeo	✓✓✓	✓✓✓	xx✓
Walmart	✓✓	✓✓	✓✓
ZipRecruiter	✓✓✓	✓✓✓	xx✓

5.2 RQ1. Accuracy of Latte

We first executed the 50 GUI test cases to ensure they are constructed correctly. We then generated the Use-Case Specifications from the tests and executed them using both SwitchAccess with two physical switches (Next and Select) and TalkBack with directional navigation (swiping).

Table 1 summarizes the presence of accessibility failures in different settings. In a cell, 'x' indicates a use case of an app (row header) that could not be executed using an assistive service (column header) due to an accessibility failure, and '✓' indicates a use case that could be fully executed without any failure. As shown under column heading "None", all original test cases passed, since they do not check the accessibility of apps, but rather evaluate the correctness of corresponding use cases. All accessibility results were manually examined and the failures were verified by the authors (the video clips of the failures can be found on the companion website [24]). Latte achieves



(a) TripIt - The initial screen **(b) TripIt - After navigation the bottom menu is disappeared** **(c) Yelp - the dotted box could not be focused by TalkBack** **(d) Reported accessibility issues by Scanner for Feedly (Orange solid boxes)**

Figure 3: The screenshots of some apps with accessibility failures

100% precision (no false positives) in determining accessibility failures in the use cases; in other words, all of the failed use cases in our experiments manifest a real accessibility issue. As can be seen, 11 use cases in 6 apps and 39 use cases in 19 apps have accessibility failures with SwitchAccess and TalkBack, respectively. Additionally, Latte detected 17 and 25 accessibility warnings using SwitchAccess and TalkBack, respectively. The warnings are not reported in Table 1, but discussed in more detail later.

We also analyzed the number of interactions for executing a use case with different assistive services. On average, Latte requires 11, 51, and 43 interactions to finish each use case under None, SwitchAccess, and TalkBack settings, respectively. Additionally, the ratios of the number of interactions required for SwitchAccess and TalkBack over those required for None were 5 and 4, respectively. This means Latte requires more than 4 interactions using assistive services to fulfill a single interaction without such services, giving us a glimpse into the practical challenges disabled users face in their usage of mobile apps.

5.3 RQ2. Latte vs. Google Accessibility Scanner

We ran Google Accessibility Scanner at each step of all use cases. We then compared the failures detected by Latte against the issues reported by Scanner. Red bold '**x**' in Table 1 represents the corresponding use case has an accessibility failure detected by Latte that Scanner could not detect.

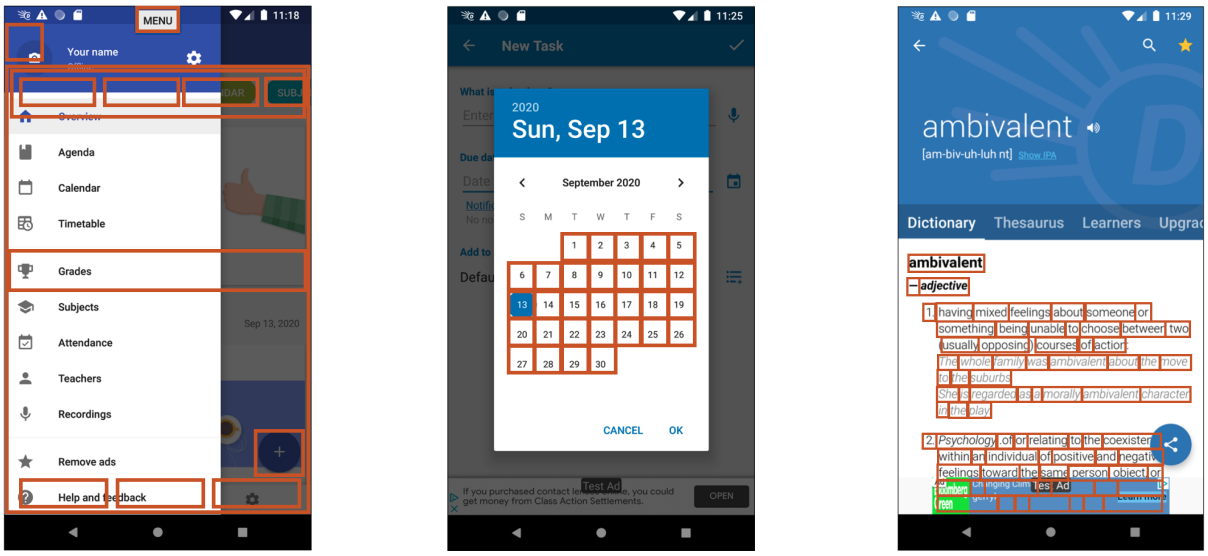
Scanner was able to detect only 18 of the 50 accessibility failures detected by Latte in the evaluated use cases. For each failure detected by LATTE, we examined all of the issues reported by Scanner. If any of those issues were found to be related to the actual fault, we assumed the Scanner can help to find the failure, e.g., Scanner can detect missing labels. Scanner could not detect any of the 11 accessibility failures detected by Latte using SwitchAccess, and 21 of the 39 failures detected by Latte using TalkBack. While Latte was able to detect all of the 5 issues confirmed by actual users with disability in the first 5 apps of Table 1, Scanner was only able to detect 1 of the issues (in Astro app). In addition, Scanner was not able to find the accessibility failures in 8 of our randomly selected subject apps.

Scanner reports an exorbitant number of issues that would overwhelm a typical developer. It reports on average 34 issues per use case for a total of 1,716 issues in the 50 use cases in our experiments. Interestingly, out of the 1,716 reported issues by Scanner, only 18 were relevant to the serious accessibility failures reported by Latte. In comparison, Latte produces at most one accessibility failure per use case. For example, in Figure 3(d), Scanner detected a number of issues, e.g., "Get Smarter" has low text contrast. The Scanner did not report any problem regarding the top two buttons (menu and search icons) that cannot be reached using TalkBack and SwitchAccess, making the app totally inaccessible.

5.4 RQ3. Qualitative Study of Detected Accessibility Failures and Warnings

5.4.1 Accessibility Failures. We manually examined all use-case failures and categorized them into the following three groups:

Dynamic Layout. Some apps change the visibility of elements on the screen dynamically. For example, Figure 3(a) shows the initial screen of *TripIt* app. If a user wants to reach the bottom menu, e.g., clicking on the Alert icon, she needs to explore the elements to locate the target widget; however, during the directional navigation with TalkBack, the bottom menu disappears (Figure 3(b)). The reason behind hiding the menu is to improve the user experience by providing more space in the middle list (where a sighted user is looking for an item). However, this change in the layout makes the bottom menu inaccessible for a blind user, since she does not know the menu has disappeared. The accessibility failures in *TripIt* and *Dictionary* apps belong to this category. This observation is consistent with the findings in a prior work [20] that showed usability and accessibility concerns are not a subset of each other. Furthermore, this example suggests improving the usability of a use case for some users may in fact degrade the accessibility of that use case for others.



(a) School Planner **(b) Todo List** **(c) Dictionary**

Figure 4: Screens of few apps with accessibility issues

Navigation Loop. Assistive services may not reach a GUI element in some apps because of a static or dynamic loop in directional navigation. Developers can create a static loop by defining custom traversal order over elements using `accessibilityTraversalAfter` attribute. While Accessibility Scanner can detect static loops, none of the apps in our experiments had this issue. On the other hand, a dynamic loop is caused by inserting elements while the user interacts with an app. For example, as shown in Section 3, the images in the background are inserted as the user navigates through them, making the navigation list virtually infinite. This issue is usually caused by `RecyclerView` widget where its adapter indefinitely inserts items into the container. The accessibility failures of this type could be found in *Yelp*, *CalorieCounter*, *CookPad*, *Geek*, and *SoundCloud* apps.

Non-Standard Implementation. Developers use customized GUI widgets in their apps that may have different behavior when users interact with them using an assistive service. For example, Figure 3(c) is the page of a restaurant in Yelp where users can rate the restaurant (the dotted blue box). However, TalkBack cannot focus on the rating widget since it is a customized `TextView` without any text. Therefore, even a sighted user using TalkBack cannot select this widget to rate the restaurant. Another source of these failures is using `WebView` widgets carelessly. `WebView` allows Android apps to load UI elements using web technologies, e.g., HTML, JavaScript. For example, in *Feedly* app (Figure 3(d)), the search icon at the top right is a `WebView` icon where its `clickable` attribute is false, meaning it cannot be invoked using assistive services. This attribute, however, does not prevent a user without disability from directly tapping the icon, which results in the corresponding JavaScript event handler to be invoked. Latte detected these types of failures in *iPlayRadio*, *Feedly*, *Checkout51*, *Yelp*, *Budget*, and *TodoList* apps.

5.4.2 Accessibility Warnings. We also studied the use cases with *Accessibility Warnings* and categorized them into four categories. Recall that Latte reports an accessibility warning when an step in the use-case execution takes more than a specific number of interactions (15 interactions in our experiments).

Overlapping Layouts. Most of the apps have multiple layouts that overlay on top of each other, i.e., Activity, menu, and dialogue layout. A user who directly interacts with the screen only considers the elements on the top layout. However, TalkBack and SwitchAccess visit all focusable elements regardless of the layout hierarchy. Therefore, users who use assistive services often navigate through elements even if they are not on the top layout. For example, Figure 4(a) shows the main screen of the School Planner app. As can be seen, the side menu is the active window (it is fully visible). However, it takes at least 12 interactions for a user to even reach the first item in the menu. Developers can fix this issue by making the elements in the non-top layouts unfocusable.

Far-Off Widget. All screen elements can be accessed virtually in no time for a user who directly interacts with the device. However, users relying on assistive services access the elements sequentially. Therefore, it takes longer for them to access a frequently used element located at the end of the navigation list. For example, in the TripIt app, Figure 3(a), a user has to navigate all elements from the top to the bottom to access the *fab* icon (the icon with a plus sign, highlighted in Figure 3(b)). To resolve these issues, developers can define a custom navigation to reduce the interactions required to reach the important elements. For example, in the School Planner app (Figure 4(a)), the fab icon is located at the top of the navigation list, although its actual position on the screen is at the bottom right.

Grid Layout. Grids provide an efficient layout for presenting multiple items in a small space, all of which can be accessed in no time

for users without disability. However, since a grid's items are accessed linearly by SwitchAccess, it takes a lot of interactions to reach the last element on the gird. For example, in the TodoList app, the calendar widget has 30 items in the grid that need to be visited before reaching to "CANCEL" or "OK" buttons (Figure 4(b)). To fix this, developers can provide different layouts for different settings, e.g., a text-based date picker when TalkBack or SwitchAccess are enabled.

Web View. There is a common practice for mobile developers to reuse web content (implemented in HTML/JavaScript) for some parts of their apps using `WebView` widget [19]. However, assistive services cannot analyze web elements properly, as shown earlier in the case of *Non-Standard Implementation* category of accessibility failures. Even if improper usage of web elements does not make an app inaccessible, it can degrade the user experience. For example, in the case of Dictionary app, shown in Figure 4(c), the definition of a term is shown in terms of a series of web elements, and each word in the passage is a clickable Android GUI element. Consequently, TalkBack and SwitchAccess need to navigate through all of these elements to reach the end of the text.

6 CONCLUDING REMARKS

In this paper, we described a novel, high-fidelity form of automated accessibility analysis for Android apps, called Latte. It reuses tests written by developers to evaluate the correctness of the important use cases in their apps to also validate the accessibility of those use cases. Latte first extracts use cases corresponding to an app's tests, and subsequently executes them with the help of assistive services. We evaluated the effectiveness of Latte by analyzing 20 apps selected from 13 different categories from Google Play and identified 32 accessibility failures that could not be identified using the state-of-the-art technique. A qualitative analysis of the results obtained in our experiments allowed us to identify a number of interesting categories of accessibility failures and warnings, together with ways in which developers can resolve them.

Although Latte finds accessibility failures undetected by Accessibility Scanner, it cannot detect accessibility barriers for users who do not use assistive services, e.g., low text contrast. Moreover, we plan to conduct user studies to gain a deeper understanding of the accessibility issues that Latte cannot detect. Therefore, Accessibility Scanner cannot be replaced completely by Latte, rather they complement each other. Our current work focuses on two assistive services and two gestures, clicking and typing. Our future research entails extending Latte to handle additional assistive services and more complex gestures. The version and settings of assistive services may improve or degrade the accessibility of the apps; consequently, our experiments may be affected by them. In future work, we would like to study the impact of settings of assistive services, which may be used by advanced users, on the accessibility of apps. Our eventual goal is to develop an extensible framework capable of identifying a broad list of accessibility issues that can be incorporated into app developers' workflow seamlessly.

ACKNOWLEDGMENTS

This work was supported in part by award numbers 1629771 and 1823262 from the National Science Foundation and an Exploration award from the School of Information and Computer Sciences at the University of California, Irvine. We would like to thank the anonymous reviewers of this paper for their detailed feedback, which helped us improve the work.

REFERENCES

[1] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility issues in Android apps: state of affairs, sentiments, and ways forward. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*. ICSE, Virtual, 1323–1334.

[2] Android. 2020. About Switch Access for Android. https://support.google.com/accessibility/android/answer/6122836?hl=en.

[3] Android. 2020. Accessibility Scanner - Apps on Google Play. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=en_US.

[4] Android. 2020. *AccessibilityService in Android*. Google. Retrieved September 15, 2020 from https://developer.android.com/guide/topics/ui/accessibility/service

[5] Android. 2020. *Android accessibility overview*. Google. Retrieved August 20, 2020 from https://support.google.com/accessibility/android/answer/6006564

[6] Android. 2020. *Android Debug Bridge*. Google. Retrieved September 15, 2020 from https://developer.android.com/studio/command-line/adb

[7] Android. 2020. *Android Studio*. Google. Retrieved August 27, 2020 from https://developer.android.com/studio

[8] Android. 2020. *Espresso : Android Developers*. Google. Retrieved August 20, 2020 from https://developer.android.com/training/testing/espresso

[9] Android. 2020. *Improve your code with lint checks*. Google. Retrieved August 20, 2020 from https://developer.android.com/studio/write/lint?hl=en

[10] Android. 2020. *TalkBack and SwitchAccess source code by Google*. Google. Retrieved September 15, 2020 from https://github.com/google/talkback

[11] Appium. 2020. Mobile App Automation Made Awesome. http://appium.io/.

[12] Apple. 2020. Apple Accessibility - iPhone. https://www.apple.com/accessibility/iphone/.

[13] Apple. 2020. Apple Accessibility Scanner. https://developer.apple.com/library/archive/documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html.

[14] Giorgio Brajnik. 2004. Comparing accessibility evaluation tools: a method for tool effectiveness. *Universal access in the information society* 3, 3-4 (2004), 252–263.

[15] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*. ICSE, Virtual, 322–334.

[16] Marcelo Medeiros Eler, José Miguel Rojas, Yan Ge, and Gordon Fraser. 2018. Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*. ICST, Västerås, Sweden, 116–126.

[17] Google. 2020. Get started on android with talkback - android accessibility help. https://support.google.com/accessibility/android/answer/6283677?hl=en.

[18] Wish Inc. 2020. Geek - Smarter Shopping. https://play.google.com/store/apps/details?id=com.contextlogic.geek&hl=en_US.

[19] Opinion Matters. 2020. *Mobile App Backlog Is Directly Damaging Revenue in the Enterprise*. BizReport. Retrieved September 15, 2020 from http://www.bizreport.com/whitepapers/mobile_app_backlog_is_directly.html

[20] Helen Petrie and Omar Kheir. 2007. The relationship between accessibility and usability of websites. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI, San Jose, California, USA, 397–406.

[21] Christopher Power, André Freire, Helen Petrie, and David Swallow. 2012. Guidelines are only half of the story: accessibility problems encountered by blind users on the web. In *Proceedings of the SIGCHI conference on human factors in computing systems*. CHI, Texas, USA, 433–442.

[22] Robolectric. 2019. robolectric/robolectric. https://github.com/robolectric/robolectric

[23] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O Wobbrock. 2017. Epidemiology as a framework for large-scale mobile application accessibility assessment. In *Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility*. ASSETS, Baltimore, MD, USA, 2–11.

[24] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. 2020. Latte companion website. https://github.com/seal-hub/Latte.

[25] Camila Silva, Marcelo Medeiros Eler, and Gordon Fraser. 2018. A survey on the tool support for the automatic evaluation of mobile accessibility. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*. DSAI, Thessaloniki, Greece, 286–293.

[26] W3. 2020. *Web Content Accessibility Guidelines (WCAG) Overview*. World Wide Web Consortium. Retrieved August 20, 2020 from https://www.w3.org/WAI/standards-guidelines/wcag/

[27] WHO. 2011. *World report on disability*. World Health Organization. Retrieved August 20, 2020 from https://www.who.int/disabilities/world_report/2011/report/en/

[28] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. 2018. Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. UIST, Berlin, Germany, 609–621.