Bility: Automated Accessibility Testing for Mobile Applications

by

Aaron Richard Vontell

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2019

Author .....
Department of Electrical Engineering and Computer Science
February 1, 2018

Certified by .....
Lalana Kagal
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chairman, Masters of Engineering Thesis Committee

Bility: Automated Accessibility Testing for Mobile Applications

by

Aaron Richard Vontell

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented a testing framework, called Bility, to assess the accessibility of mobile applications. Implemented in Java and Kotlin, this framework automatically navigates through an application, finding both dynamic and static accessibility issues. I developed techniques for unique view detection, automatically building a representation of the dynamic behavior of an application, and making navigation decisions based on application state and history in order to find these accessibility issues. I compared Bility's ability to detect issues against existing tools such as the Google Accessibility Scanner. I found that Bility performs significantly better and is able to detect additional static issues, as well as discover dynamic accessibility issues that current tools do not detect.

Thesis Supervisor: Lalana Kagal
Title: Principal Research Scientist

Acknowledgments

I would like to thank my parents, grandparents, and sister for their support and encouragement during my time at MIT. I am also extremely grateful for all of my friends and colleagues who were always open to discussing this work.

I would also like to thank Lalana Kagal for her advisement and support since day one of this project.

Finally, I would like to give a huge thank you to all researchers, developers, and activists who advance the field of accessibility, allowing millions of people across the world to use technology that would otherwise be inaccessible to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile devices have become increasiningly important and even necessary for functioning in today's society. Whether it be communicating with friends or finding information, billions of people use smartphones on a daily basis. For those with disabilities such as difficulty in hearing, vision, motor skills, and cognitive processing, using smartphones can be challenging. By following accessibility guidelines and developing applications with these user groups in mind, applications can be modified to be more accessible. However, while mobile applications are mostly covered by the same accessibility standards as Web applications, the form factor and interaction techniques employed by mobile phones brings new challenges that developers must address during development.

Through research done with my research group [13], I found that while mobile accessibility is extremely important, tools and testing frameworks for developing accessible applications are substantially lacking. Tutorials and guidelines are fragmented, libraries for accessible development are difficult to use, and real-world testing for accessibility is almost nonexistent. As the barrier to entry is too high, developers often avoid the task of making applications accessible. The time and knowledge needed to make an accessible application is a deterrent for most application developers. Furthermore, many accessibility testing frameworks are concerned with static accessibility issues (issues with static snapshots of application screens, such as color contrasts and screen reader labels), while dynamic accessibility issues (issues with the dynamic

behavior of the application, such as navigation and user input response) are addressed less frequently.

In order to deal with these issues, I present **Bility**[1], a framework for testing both dynamic and static accessibility on mobile phones in general. The framework introduces new techniques for analyzing user interfaces, such as determining changes in context solely through user interface information, as well as navigating through an application automatically with little to no developer input. Overall, I have made the following contributions:

1. An algorithm for automatically generating and reducing the state space of a user interface in the context of a specific analysis.

2. A platform-agnostic language used to describe user interfaces, including audio, physical, and motor elements.

3. An automated process to start, navigate, and record an Android application in an attempt to fully explore the application state space.

4. A process for testing dynamic accessibility issues such as keyboard navigation and spontaneous changes in application context through the construction of a state machine representing the application.

An example of this framework, which I have implemented in Java and Kotlin, uses the above mentioned techniques to detect both static and dynamic accessibility issues on Android phones. These issues are based on the Web Content Accessibility Guidelines 2.0 (WCAG) [12], a popular guideline for increasing accessibility on websites. I also compare the capabilities of Bility to existing accessibility tools and techniques such as user testing and the Google Accessibility Scanner. [8]

1.1 Background and Related Work

Evaluating the accessibility of software ranges from manual testing and static testing during development to automated testing post-development. While much accessibility

[1] `https://github.com/vontell/Bility`

research and efforts have focused on the web, the proliferation of mobile devices has made mobile accessibility an important endeavor, and as such a few guidelines and tools have been created to evaluate accessibility on mobile devices.

1.1.1 Accessibility Guidelines and Checklists

The most common form of accessibility testing is manual testing using guidelines and accessibility checklists. For instance, the Web Content Accessibility Guidelines (WCAG 2.0) released by the W3C is often used to assess the accessibility of both websites and mobile applications. [12] The guideline provides principles to follow to make software more accessible, as well as success criteria to determine if a principle has been met. For instance, satisfaction of principle 2.1.1 requires that all functionality of an application can be achieved through the use of a keyboard. [9]

However, WCAG 2.0 is hardly the only guideline that developers choose to follow. For instance, the App Quality Alliance puts forth a large checklist of accessibility criteria that developers can use to assess the accessibility of their applications, geared toward specific platforms such as Android and iOS. [2]

The W3C, App Quality Alliance, and many other groups and initiatives have put forth checklists and guidelines for accessibility, and as such developers have many options for accessibility testing. However, all of these guidelines and checklists rely on manual testing. A developer or user must navigate through the application, checking various accessibility factors. This process can be time consuming and costly. For instance, many usability and consulting companies provide services for assessing the accessibility of applications, but these can cost thousands of dollars. [14]

1.1.2 Accessibility Testing for Mobile Applications

There has not been much research into the development of accessibility testing tools , except for formalizing the various requirements of a mobile application design to be accessible to users with various disabilities [17]. In terms of libraries and frameworks that developers use to test accessible applications, low-level and simple

testing functionality is provided by the Android operating system [7], while some tools such as the Google Accessibility Scanner provide testing of static accessibility issues. However, these tools are difficult to use, frequently non-versatile to handling certain types of applications, and do not detect many of the accessibility issues that exist within applications.

Accessibility Linting Tools

If creating Android applications using Android Studio, the IDE provides linting tools for detecting some accessibility violations during development, giving direct feedback to the user. However, the base linting tools from Android only provide checks for content descriptions on image-based Views. [4] For instance, if an image or icon does not have a `contentDescription` field set, the IDE will provide a warning (see Fig, 1-1). Once a `contentDescription` field on an image is set, screen readers such as TalkBack can then describe the content of that image to a user. These warnings serve as a quick, real-time check for accessibility issues as a developer writes an application.

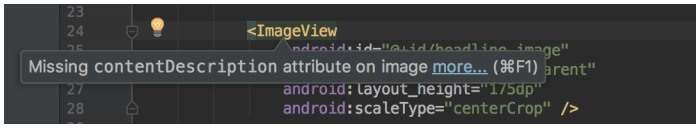

Figure 1-1: A lint warning within Android Studio regarding a missing content description on ImageViews.

Android's AccessibilityNodeInfo Object

Accessibility testing can be automated by writing custom user interface tests for Android applications. Popular libraries such as Espresso, as well as Android's base instrumented testing framework, allows for accessibility testing through the `AccessibilityNodeInfo` object. [5] This class provides functionality for accessing and manipulating user interface information such as text inputs, determining if an element has focus, and checking if an element is clickable.

The information provided by the `AccessibilityNodeInfo` class is great for checking low-level properties of a user interface, but the developer must still provide logic for

navigating through the interface, interacting with components, and using information about these components to determine accessibility. Some work has been done with testing Android accessibility through the Espresso [1] and Robolectric [3] testing frameworks, which utilize this `AccessibilityNodeInfo` class. However, information on how to use these libraries in an effective way is lacking. In fact, the main repository for accessibility testing examples, called eyes-free, has not been updated since 2015. [10] The truth is that these tools are rarely used in accessibility testing, and the traces that they provide to find issues are not as useful as they could be.

Google Accessibility Scanner

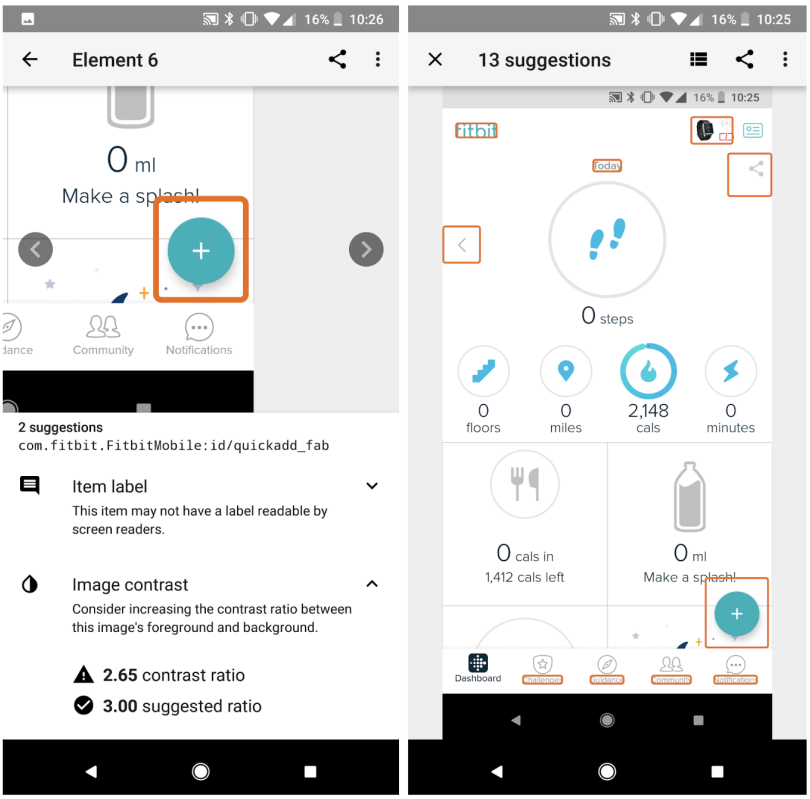

Figure 1-2: The Google Accessibility Scanner finds issues with color and touch area sizes. The left image shows regions where issues have been found, while the right image shows the error and suggestions for a floating action button.

The Google Accessibility Scanner is one of the most feature-rich solutions to accessibility testing on Android. Rather than assessing accessibility through static code checks, or running automated tests during development, the Accessibility Scanner

tests compiled applications, providing an accessibility assessment directly on a device [8].

The Google Accessibility Scanner provides great testing functionality in two ways. First off, the application is extremely easy to use for both developers and non-developers. When the "check" icon is clicked to begin testing an open application, the user is presented with an interface that highlights accessibility issues directly on the user interface. These issues can then be clicked for more detailed information and suggestions, and the application supports sending the test results through email and other sharing services. An example of this usage on the Fitbit application can be seen in Figure 1-2.

Secondly, the application detects more issues than the standard linting and testing tools with no configuration. Discussed further in the evaluation chapter, the Google Accessibility Scanner is capable of finding contrast issues, missing content descriptions, small touch targets, and other static user interface issues. While other tools provide better automation functionality, the Google Accessibility Scanner offers the most complete experience for the least amount of setup.

1.1.3 Gaps with Current Testing Techniques

The problem with existing accessibility testing solutions is two-fold; setting up effective accessibility test is time consuming, and the existing solutions cover only a subset of accessibility issues with minimal configuration. In fact, research shows that even standard automated testing of application functionality and business logic is missing in 86% of developed applications [25], and in turn automated accessibility testing is most often lacking from most applications as well.

If a developer decides to take on accessibility testing, they will find that detecting static issues such as contrasts and touch target sizes is doable with automated testing frameworks and the Google Accessibility Scanner. However, testing dynamic accessibility issues, such as navigation with a keyboard, and detecting actions that are time-based, is much more time consuming. This means that while accessibility testing is possible with little configuration, testing for a whole range of dynamic accessibility

issues is often not completed. Many of these types of principles are detailed and required by guidelines such as WCAG 2.0, and are deemed important for increasing the accessibility of software. Furthermore, the interfaces and reporting techniques used to relay accessibility results to developers can be insufficient or difficult to parse. For instance, some frameworks will report the same accessibility issue multiple times if a user interface element with that issue is repeated from a single definition, although only one real root issue exists in this case.

The goal of Bility is to tackle both of these hurdles; provide an extremely easy-to-use accessibility testing framework that supports a wide range of accessibility tests but that requires minimal configuration. The ease in configuration and use of the Bility framework will encourage developers to evaluate their applications for accessibility and lead to the development of more accessible mobile applications.

Chapter 2

Representation of Mobile User Interfaces

Whether in the context of accessibility testing or other modes of analysis, the representation of a user interface is extremely important in detecting various aspects of a user interface. The representation of an application under study needs to expose important factors that facilitate its analysis.

This section discusses a few current approaches to representing user interfaces, and details why they fall short when testing applications for accessibility. To remedy some of these issues, I will also present new language for describing user interfaces in the context of accessibility testing.

2.1 Current Approaches

Following from the Model-View-Controller paradigm (a commonly used representation of user interfaces), the state of a user interface is defined by its underlying model, which is portrayed through the update of user interface element properties, which is managed by the controller. As a user interacts with a user interface, the underlying model may change, and therefore the information displayed to the user may change as well.

However, this method falls short of accurately representing the state of an applica-

tion from a **user's perspective**. The underlying model or data may surface to the user through changes in user interface properties, but the user has limited knowledge of the entire underlying state. If a piece of data is held within the model for later use, and this data is not portrayed within the user interface, then the state of the user interface is no different than that same user interface without that piece of hidden data (at least from the user's perspective, assuming the user has no knowledge of that hidden data). When it comes to modeling user interface state through accessibility, it is fruitful to track only what we expect the user to know - anything that is actually displayed within the user interface. By only observing this information, we get a more accurate description of what the user can actually respond to.

The "Model" aspect of this paradigm is not the only part that fails when attempting accessibility testing. The "View" component also falls short, especially within mobile applications. When someone discusses a user interface in terms of its Views within Android, they are most often referring to the user interface components that are rendered and displayed on the screen - elements such as buttons, text, and images. However, analyzing only these views leaves out much of the entire user interface's capabilities. For instance, on-device buttons, attachable keyboards, output audio, physical switches, gyroscopes, and vibrations are examples of input and output components of a user interface that are completely ignored by these Views. Given the restrictions imposed by this View representation of user interfaces in Android, it is helpful to define a new way to describe these user interfaces.

2.2 Proposed Approach

2.2.1 User Interface Terminology

The implementation of Bility that I will discuss within this thesis is made for Android (although it can be extended to iOS and other platforms). As seen from above, it will be difficult to discuss the user interfaces of Android using only Views as the base terminology, and therefore I will present new terminology. For the purpose of this

thesis, this language will be referred to as Universal Design Language (UDL).

The current way to describe user interfaces are well known to designers and software developers, most often using the following concepts:

1. Views, components, or elements - the essential objects that a user interface can be composed of. These objects have styles and properties.

2. Hierarchy and structure - the organization of these views, components, and elements.

3. State machines - A way to represent the state of a user interface at any given moment, which also defines how a user may move from one state to another.

In order to capture this idea of user interface elements that are not defined solely as items that are rendered to display, and to also address this idea that a user element is defined solely based on how a user interprets it, I will use the following terms:

- **Perceptifer** - Coming from the Latin roots *percept* (something to be perceived), and *fer* (bearer of), a **perceptifer** is a bearer of things that can be perceived by a user. For example, a button is a perceptifer; it is a visual interface component that can potentially be perceived by a user. Another example is an audio clip, as it conveys sounds that can potentially be perceived by a user. A user can sometimes interact with perceptifers.

- **Percept** - An object of perception, or something that is perceived. A perceptifer will communicate percepts along their associated output channels on a device (i.e. a visual channel such as a screen, an audio channel such as a speaker, or a motor channel such as a vibration motor). Percepts are perceived and processed by a user if that user is capable of receiving that percept along the correct input channel (e.g. a screen if the user is not blind). A percept holds the actual information that allows a user to reason about a perceptifer.

- **Literal Interface** - A set of perceptifers and the percepts they display, as well as the set of output channels that these percepts are communicated upon.

- **User / Persona** - The actual agent which interacts with a user interface by perceiving percepts and making decisions on how to interact with the user interface. In the case of Bility, the persona is virtual and automated, making decisions based on the percepts it consumes.

- **Internal Interface** - An internal understanding and representation of a literal interface as perceived by a specific user or persona. For example, a literal interface will be translated into a different internal interface for a blind user when compared to that of a non-blind user.

The intended purpose of this terminology is to simplify the transition of information from a user interface to a user, while encompassing all aspects of the user interface that would not be accessible solely through a discussion of Views or rendered components, as well as formalizing the difference between a defined interface and what the user can actually understand or perceive from that user interface. This latter aspect will be extremely important when taking disabilities into account which affect the users interpretation and interactions with a user interface.

As an example of this terminology in action, I will walk through two examples. First, imagine a button on a user interface. This button can be described with this terminology in the following way:

- A button is a perceptifer.

- The text it conveys, background color, text color, border color, border radius, font size, font kerning, font family, position on the page, and audio that can be read out by a screen reader are all examples of percepts that this perceptifer bears.

- Its percepts are communicated through a visual channel (the screen) and audio channel (screen reader and speaker).

You will notice that this button perceptifer does not have a percept such as "affords clicking." This is because the fact that a button affords clickability is not inherent to

the perceptifer itself, but is rather an interpreted fact by the user. For instance, if a user has somehow never used a software interface before, they may not understand that this button is something that can be clicked.

However, in the case of a button, it is most likely safe to assume that the described percepts (background color, border, etc...) would permit the user to interpret this object as a button. In order to simplify this notion, it is permitted to define a special type of percept, called a **virtual percept**. A virtual percept conveys information such as clickability, focusability, and the Android-specific type of perceptifer (e.g. a TextView). Without virtual percepts, a persona would need to provide all logic to interpret this set of percepts as a button - this is simply a shortcut which should be used sparingly. An ideally implemented persona would ignore all virtual percepts in favor of interpreting all real percepts.

Another example of this use of terminology is an audio clip played through the speakers of a device, which is described in the following way:

- An audio clip is a perceptifer.
- Its percepts are communicated through an audio channel (the device's speakers).
- The percepts that this perceptifer bears are information such as frequency over time and the volume of the audio signal.
- A virtual percept would be an item such as the language content of the audio clip.

Note once again that information such as spoken word contained within the audio clip are never conveyed through percepts, as the signal may be interpreted in different ways for different users. Rather, the implementation of Bility may provide a virtual perceptifer in these cases for what the human-readable contents of this audio clip may be.

Now that this terminology exists, we can begin to describe user interfaces using a base set of percepts and virtual percepts. The basic list of percepts that can be tracked within Bility on a given perceptifer are:

1. **Text** - Rendered text content
2. **Location** - The pixel coordinates of the top left of this perceptifer
3. **Size** - The total width and height (or bounding box)
4. **Text Color** - A list of colors used within rendered text
5. **Background Color** - A list of colors used as a background
6. **Scroll Progress** - The number of pixels scrolled if this perceptifer is a scrollable element
7. **Alpha** - The opacity level of the entire perceptifer
8. **Font Size** - The size of rendered text
9. **Font Family** - The type of font or font family being displayed
10. **Font Kerning** - The spacing between characters within rendered text
11. **Line Spacing** - The spacing between lines of rendered text
12. **Font Style** - Indicator of bold, italic, underlined, and striked text
13. **Physical Button** - A boolean indicating whether this button is a physical button on the device, such as a volume rocker
14. **Focused** - A virtual percept describing whether this perceptifer currently has focus
15. **Audio Content** - A virtual percept describing the contents of an audio stream, such as the words or specific noises created.
16. **Name** - A virtual percept describing the name of this UI element for the specific platform.
17. **Screen Reader Content** - A virtual percept containing any text that the system would pass to a screen reader for announcement.

18. **Is Root** - A virtual percept which is simply a boolean, which if true means that this perceptifer is the root element of this entire user interface.

19. **Clickable** - A virtual percept which is a boolean, which if true indicates that this UI component can be clicked.

20. **Child Spatial Relations** - A tree or ordering of percepts representing the hierarchy of this perceptifer's immediate children, if this perceptifer is a container of other perceptifers.

2.2.2 Application State Representation

The components contained within the user interfaces of an application are not enough to describe that user interface. Applications respond and react to input, and therefore change over time. This dynamic behavior is incredibly important and necessary to describe the full functionality of a mobile application. Within the context of accessibility, this aspect of an application is also necessary to analyze the potential usability for users with motor disabilities and cognitive disabilities.

Representations as Automatons

User interfaces and mobile applications are inherently stateful, and as such are often described during prototyping as state machines or automatons. For example, Sketch for Mac allows designers to create screens for an application, and then make them interactable by creating arrows that define movements between screens when clicking elements (see Figure 2-1). In iOS development, a storyboard is often used to create a flow of user interface logic, which is essentially an automaton describing the various states of the application and how a user may transition between them (see Figure 2-2).

Within the Bility framework, applications are also represented as automatons and state machines. However, in order to make Bility extremely easy to use, it is not required that developers define this automaton beforehand. Rather, the Bility framework provides a persona that intelligently navigates the user interface, building

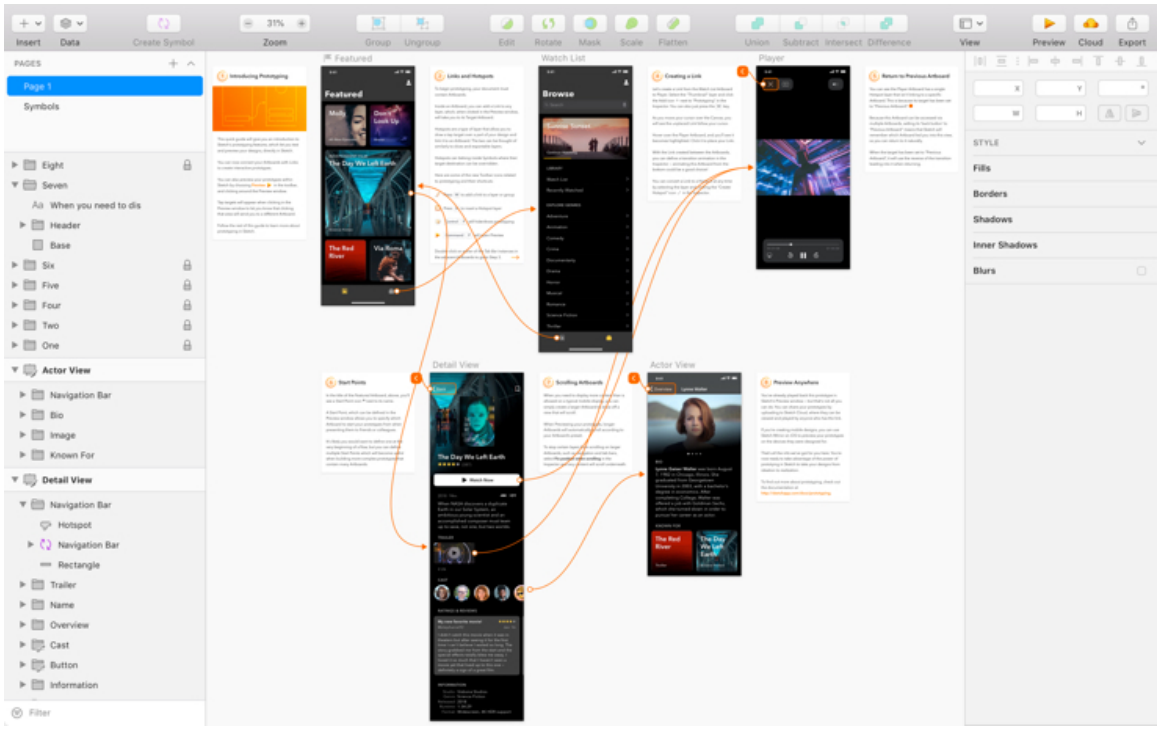

Figure 2-1: A design prototype of an application made within Sketch. The orange arrows define transitions between screens.

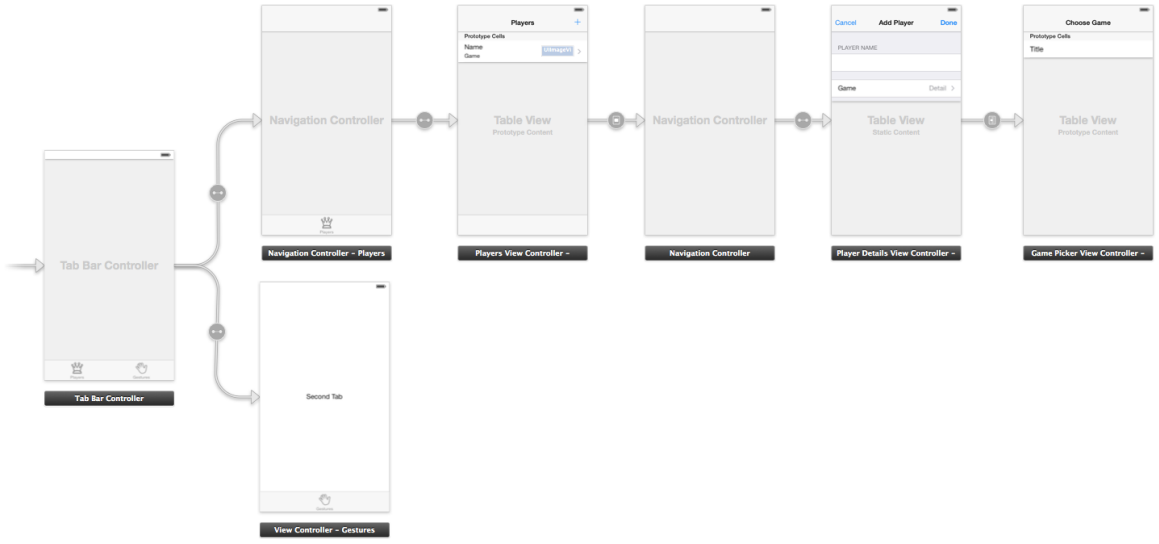

Figure 2-2: A storyboard for an iOS application, where arrows define how a user can move from one screen to another.

up a representation of the application as an automaton made of states which contain a collection of perceptifers.

User Interface Hashing and Simplification

When building a state machine for an application based on simulated user actions and user interface information, deciding what constitutes a change in user interface state becomes difficult. Different types of analyses may call for different properties to track within a user interface. For example, if a particular screen is scrolled down by one pixel, and no viewable elements change in content (just their positions on the page change), this may not constitute a meaningful change in state, and yet a property of the user interface (the scroll progress) has changed.

With this new language of breaking down a user interface into percepts, it becomes easy not only to track important parts of the user interface based on a type of analysis, but it also becomes easy to create a hash representing a user interface. Creating this hash of a user interface allows us to selectively choose which aspects of a user interface are important, as well as allows Bility to quickly compare two user interfaces to see if they are "similar" under these choices.

For example, in the context of accessibility, the Bility framework proposes a base set of percepts to track that are deemed important for determining the state of the application. The percepts are chosen in the following way: **if a percept may affect the static accessibility of the application, it is included as a property when determining state**. The following percepts are chosen as ones which may affect the accessibility of the application: **ALPHA**, **BACKGROUND_COLOR**, **FONT_SIZE**, **FONT_STYLE**, **LINE_SPACING**, **VIRTUAL_NAME**, **VIRTUAL_FOCUSABLE**, and **TEXT_COLOR**. This selection of percepts essentially says that changes in state do not occur when content changes, or even the amount of content changes; rather, a state change occurs when the new state would potentially affect the accessibility of the application. Additionally, virtual percepts such as the order of children within containers is tracked.

I will walk through this algorithm using the example user interface in Figure 2-3. The top row shows the breakdown of user interface elements into containers and their children, all the way to the leaves of the view hierarchy. The bottom row shows the

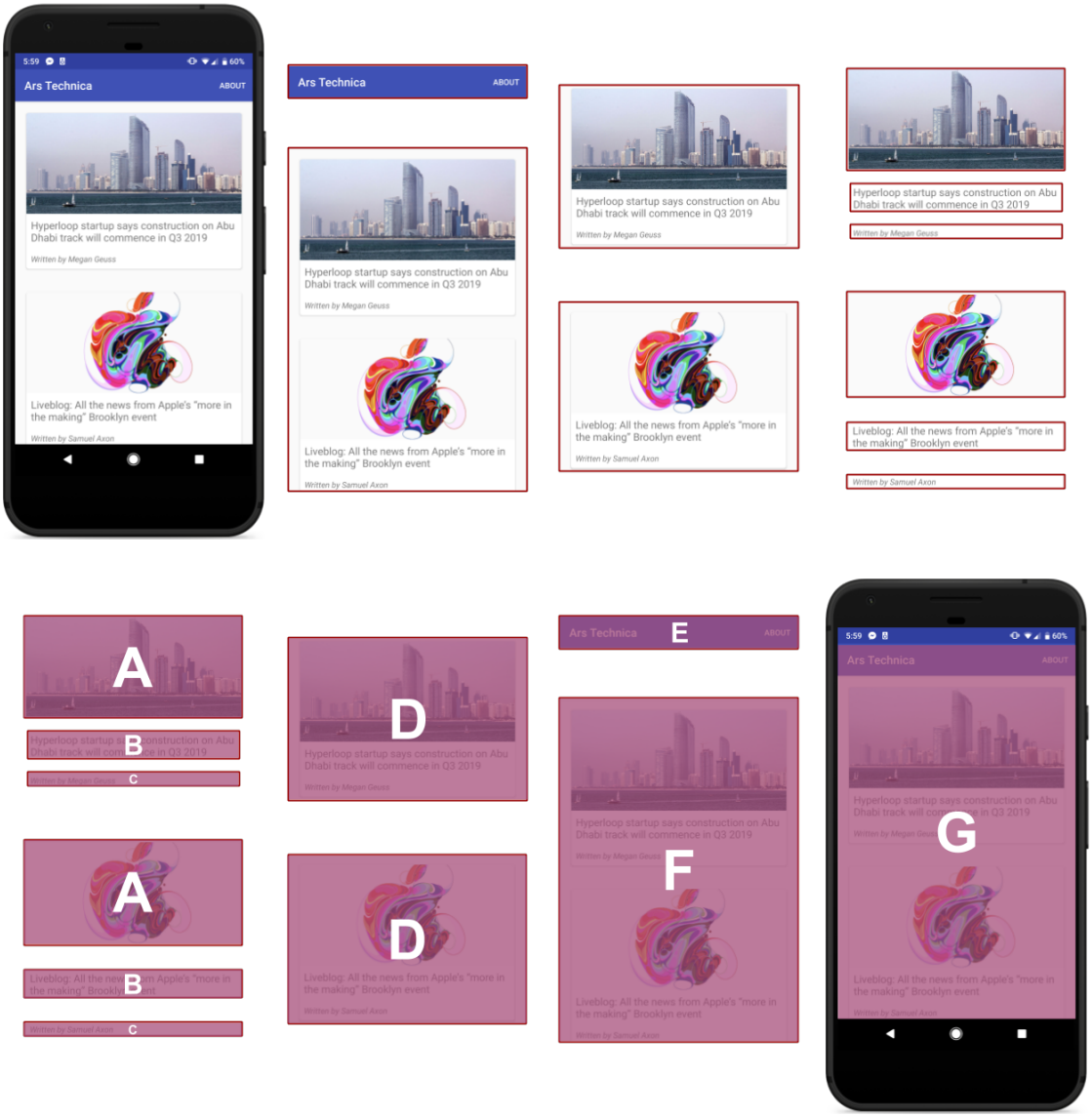

Figure 2-3: The decomposition of a user interface into parts that represent similar components found within the user interface.

"hash" of each element, parameterized by the percepts that are tracked as mentioned above.

For example, within the section marked as "A", the perceptifer being analyzed is an ImageView, which is hashed based on the name of the View (i.e. ImageView) and alpha level (i.e. opacity of 1). Next, the sections marked as "B" are hashed based on

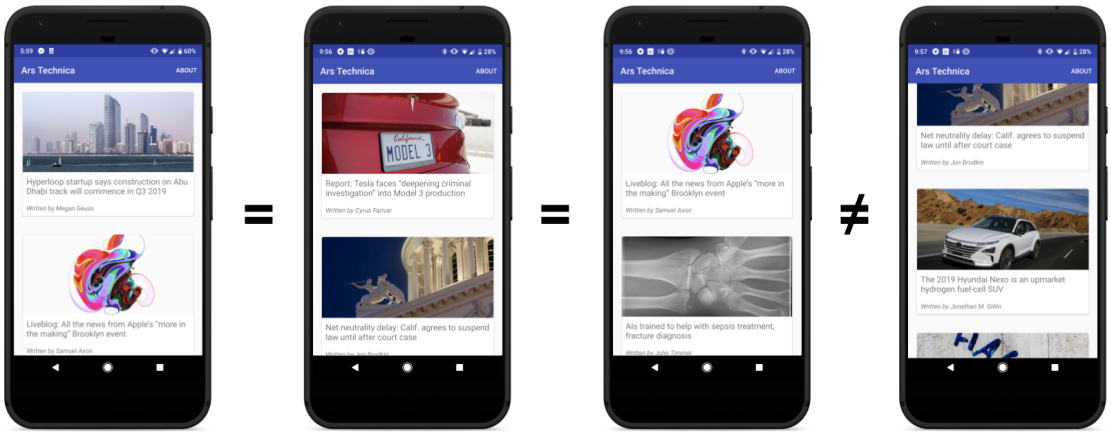

Figure 2-4: The first three UI states are equal according to the base hashing technique, as described in Algorithm 1, since the same layout is present (only image and text content has changed). However, the last state is different, due to the new partial article view at the bottom of the screen.

the tracked percepts of text color, font size, font type, kerning, alpha, and background color. Any text elements within the same properties are hashed to the same value (in this case "B"). Section "C" is the result of hashing the same types of percepts, with this hash considering font styles and sizes that are different from that of hash "B".

Section "D" represents a hash of the container that holds the ImageView and TextViews represented by hashes A, B, and C. The hash of a container is simply an in-order hash of the hashes of its children, where the order is defined by the distance from the origin of the container to the child element. This container hashing happens for each container, with the root container hash representing the hash of the entire user interface (represented as "G" in this example).

The algorithm discussed above to hash a user interface into a single value representing its composition of important aspects is specifically laid out in Algorithm 1. Note that the transformation function `transformFn` is the function that can be used to modify which percepts are considered in hashing. For instance, for the accessibility hash, the accessibility-related percepts mentioned above are returned when given a perceptifer. Also note that the `getIdsOfChildren` function takes in a perceptifer and returns an in-order list of the IDs for it's children, indicated by the CHILDREN_SPATIAL_RELATIONS virtual percept. Using the maps returned by

`getAccessibilityHashes`, one can find the entry for the root container of this user interface, which houses the hash for that user interface.

Algorithm 1: User Interface Hashing

Function `getAccessibilityHashes`(*perceptifers*)**:**
- **Data:** A collection of all perceptifers within a literal interface.
- **Result:** A mapping M of UI hashes to a list of perceptifer ids who have that hash, a map $idsToHash$ of perceptifer IDs to hashes, and a map ps of perceptifer IDs to perceptifers.
- $ps \leftarrow$ hash map of perceptifer ID to perceptifer;
- $root \leftarrow$ `getRoot`(*perceptifers*); // getRoot returns the perceptifer highest in the View hierarchy
- $idsToHash \leftarrow$ initialize hash map of strings to integers;
- `performPerceptiferHash`(*root, ps, idsToHash, true,* $transformFn$);
- $m \leftarrow$ reverse $idsToHash$, chaining conflicts;
- **return** *m, idsToHash, and ps*;

Function `performPerceptiferHash`(*p, pMap, cache, ignoreRepeat,* $transformFn$)**:**
- **Data:** p is a perceptifer, $pMap$ is a map of perceptifer IDs to perceptifer objects, *cache* is a map of perceptifer IDs to perceptifer hashes, *ignoreRepeat* is a boolean, and $transformFn$ is a function that takes in a perceptifer and returns a set of percepts
- **Result:** Returns the hash of the perceptifer p, as well as recursively fills *cache* with the user interface hashing results of all child perceptifers.
- *percepts* $\leftarrow$ `transformFn`(p);
- *children* $\leftarrow$ `getIdsOfChildren`(p);
- **if** *children is not empty* **then**
  - *childHashes* $\leftarrow$ map *children* onto `performPerceptiferHash`($pMap$`[`*child*`]`, $pMap$, *cache*, *ignoreRepeat*, $transformFn$);
  - **if** *ignoreRepeat* **then**
    - *childHashes* $\leftarrow$ *childHashes* with duplicates removed
  - **end**
  - *hash* $\leftarrow$ `Objects.hash`(*childHashes*, *percepts*); // Defined by Java
  - *cache*.`put`(p.id, *hash*);
  - **return** *hash*;
- **end**
- *hash* $\leftarrow$ *percepts*`.hashCode()`; // Defined by Java
- *cache*.`put`(p.id, *hash*);
- **return** *hash*;

2.2.3 Limitations, Configuration, Extensibility

Bility is not limited to solely tracking percepts that are related to the immediate accessibility of the user interface. Tracking components that will affect accessibility is an excellent way to avoid over-reporting static accessibility issues, and while it is a good heuristic for detecting various dynamic states of the application, every application may have specific states that need to be tracked based on different types of percepts.

I have implemented the state logger in a way that makes it extremely easy for a developer to extend and modify this state tracking, which can be done in multiple ways. First, the developer can simply indicate which base percepts should be tracked. For examples, by default, text is not tracked as a differentiator for state, since text is often dynamically generated. If the developer's application is dependent on tracking text (i.e. a label that displays "Yes" or "No"), the user can turn on text tracking.

Second, the developer can create their own percepts specific to their application. For instance, if an underlying data state is extremely important for tracking user state, a developer can add a virtual percept to a perceptifer. This virtual percept can then be used to differentiate between states.

Third, a developer may want to do more complex operations such as "track text if it says 'Yes' or 'No', but otherwise ignore the text." Bility also permits this capability, by allowing the user to override both the hashing function and percept transform function.

The goal of the Bility framework is to detect a wider range of accessibility issues. By tracking dynamic issues within a user interface, we have to balance the trade-off of state coverage and state space explosion. My proposed configurable approach to tracking states in a generalized manner allows developers to adjust this trade-off, with the suggested accessibility percepts acting as a first baseline in accessibility testing.

Chapter 3

System Design

I have taken two core principles into consideration for the implementation of the Bility framework namely ensuring ease of use for developers, and ensuring that the system can be extended and adapted to new interfaces and platforms. This section discusses the implementation of the automated tester using the techniques described in the previous chapter.

3.1 Architecture

Existing accessibility analysis tools (primarily Google Accessibility Scanner and Android Studio linting tools) rely on an Android device or integrated development environment. As the Bility testing framework focuses on identifying dynamic and real-time accessibility issues, it uses both an Android device (emulated or physical) and a host computer (or server) to run the accessibility tests.

An external server is used to offload computation that would otherwise need to be completed on the mobile device during testing. The Android device handles navigation requests from the serve, and sends user interface information (as JSON) and screenshots (as PNGs) back to the server for analyzing. Figure 3-1 shows this architecture in more detail.

For ease of deployment, the database server, website server, and maven server are created and maintained as Docker containers. The Testing server, however, is

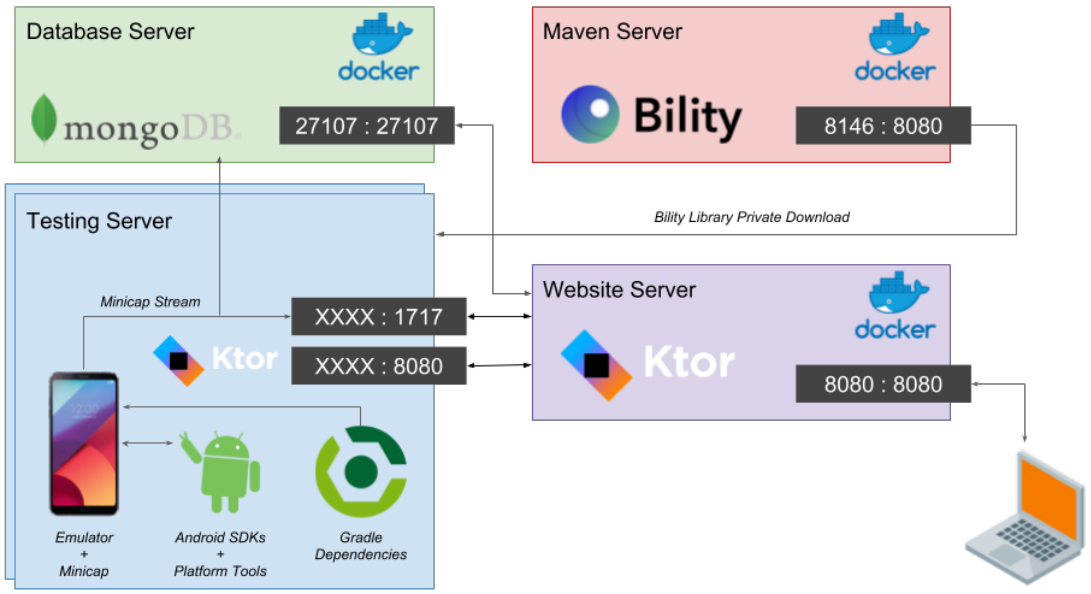

Figure 3-1: An overview of the architecture for running the test server and test devices.

managed and deployed using a script that can be executed on any Linux or Mac-based machine. The testing server does not run on Docker due to limitations of running emulated Android devices on Docker instances caused by hardware mapping. [24]

3.1.1 Testing server

The testing server is a machine that handles the control of an Android device. The current implementation is written in Kotlin and Java, using Ktor[1] as it is a backend framework for HTTP connections. This machine also contains the Android SDK and Platform Tools needed to interact with an Android device.

The Android device can be either an emulator or real device. It needs to be connected directly to the testing server as the communication happens over HTTP rather than the Android Device Bridge.

The Ktor HTTP server provides endpoints for receiving user interface information from the device, and also endpoints for sending data to the device regarding commands and actions, if new actions are available. This looping process can be found in Figure

[1] `https://ktor.io/`

4-1.

The testing server also informs the developer and user of the test results through multiple outputs. For one, it writes all test results to a database server, which is a MongoDB server running within a Docker container. The testing server offers endpoints for the website server (the main portal for the developer) to retrieve test information. Finally, the Minicap library [29] is used to stream video from the Android device to the Website Server, so that the user can view testing in real-time within the browser.

Note that the testing server is not restricted to being a singleton instance. The testing framework supports the ability to duplicate the testing server to test on multiple devices and machines at one time. The website server in turn manages the availability of these machines. This allows for accessibility testing with multiple applications and multiple configurations in parallel.

3.1.2 Website Server

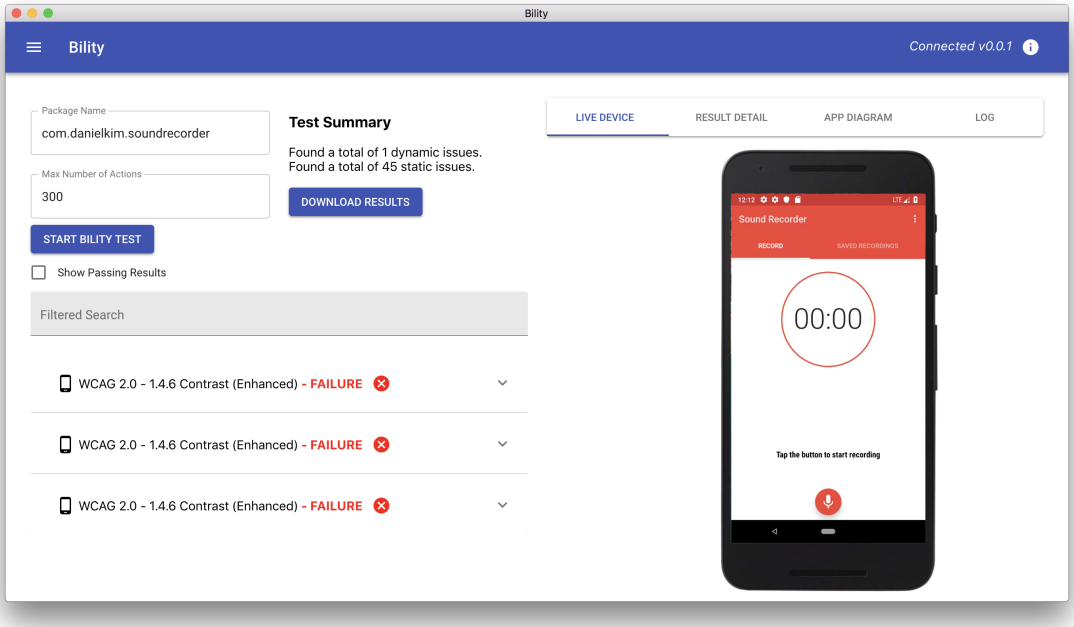

Figure 3-2: The Bility application showing a test in progress with live issue reporting and application viewing.

The website server acts as the main interface that a developer interacts with during the testing process. One principle of the Bility testing framework to to make accessibility testing as easy and informative as possible for the developer. As seen in Figure 3-2, the interface is intuitive and straightforward. The user is able to upload an Android project and configure the test easily. The configuration options include specifications such as the maximum number of actions that the persona can take. The persona also supports configuration of the actions types it can take, the types of issues it can detect, and the wait time between actions (used for slow devices and network connections).

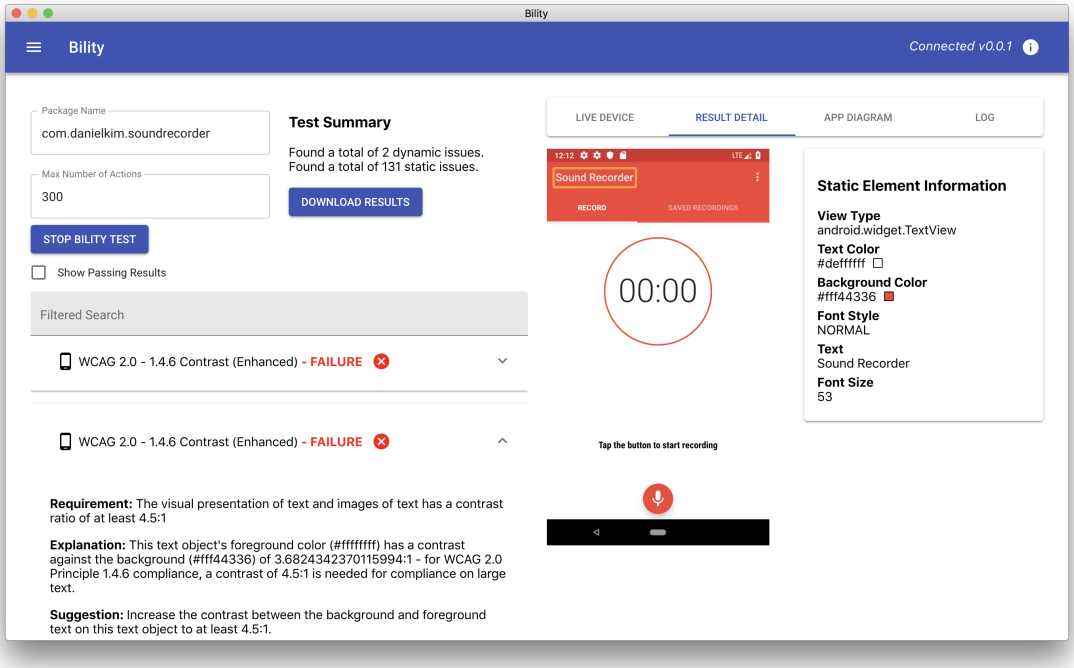

Figure 3-3: An in-depth view of a contrast issue. The left side of the application provides a list of issues and detailed information for each issue, while the right side provides context through screenshots and View information.

Once the user starts the accessibility test, they are presented with a user interface that allows them to both watch the test and view accessibility testing results. As seen in Figure 3-3, these results are rich with information - any issue found includes references to the core accessibility issue as detailed in WCAG 2.0, an explanation for why that issue was found, a count of how many instances of that issue were found,

and even suggestions on how to solve that issue. A screenshot of the user interface component is also shown for reference, if available.

The interface also provides a way for user to view dynamic issues and information about their application. For instance, the left side of Figure 3-4 shows the App Diagram tab, which features a state diagram that shows the states and actions taken to navigate between states. The graph can be interacted with, zoomed, and includes information in the form of action types, action subjects, and screenshots.

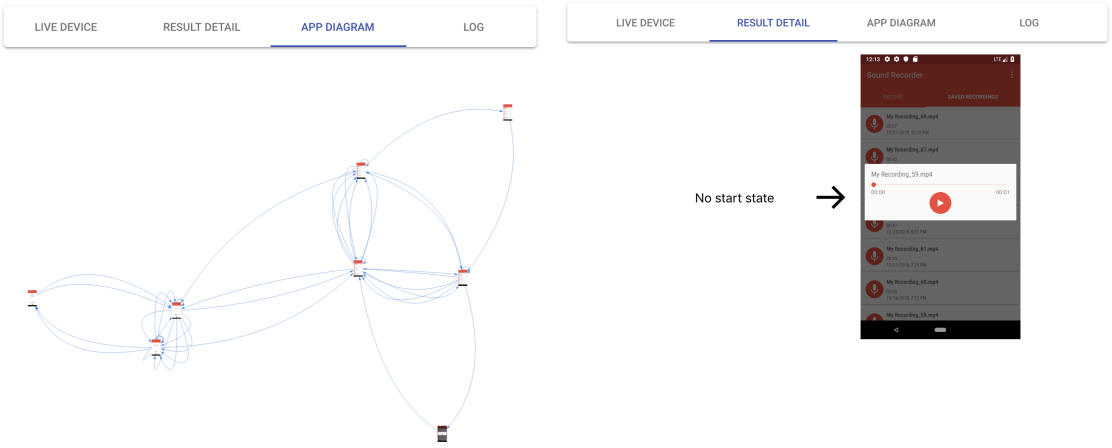

Figure 3-4: The left image displays the App Diagram, a state machine representing that screens and actions supported by this application. The right image shows a detailed view of a keyboard accessibility issue.

Additionally, a detailed view for dynamic accessibility issues shows the relation between two states. For instance, the right side of Figure 3-4 shows the detailed view for a keyboard navigation accessibility issue, showing that there is no state that can reach the destination state using only a keyboard.

The results presented to the user in real time are also saved on the database server allowing for post-test retrieval of this information. Additionally, the user may download the test results using the "Download Results" button, which returns a JSON object with all issues found and all parameters used to run the accessibility test.

3.1.3 Maven Server

Rather than making sure each testing server has the latest code available for the Bility library, all Java dependencies required for running tests are located on a separate Maven server, hosted using Artifactory. [23] Whenever a Testing Server begins to compile an application for testing, it first checks for new versions of the Bility library on this remote Maven server, allowing existing and running testing servers to stay up-to-date with any new issue tracking techniques and result types.

Chapter 4

Accessibility Testing

Using the architecture and concepts described in earlier chapters, my core implementation of Bility includes logic and processes for analyzing and assessing the accessibility of mobile applications. In this section I discuss the specific algorithms and rules for detecting static and dynamic accessibility issues, as well as the algorithm for analyzing and taking user actions based on the current and past user interface information.

4.1 Observe-Process-React Loop

4.1.1 Collecting User Interface Information

The Android device is responsible for scraping user interface information and sending it to the server for processing. There are many techniques that can be used to scrape the user interface - for instance, the Android Device Bridge provides commands for dumping the View hierarchy and basic properties as XML. [15] Another technique is to implement custom Android Views that can log View information. [21]

The Bility library is run using Android's Instrumented Testing tool, which provides a way to gain access to any currently-running Activities, windows, and Views. Using Java reflection to access all available windows (full code for this function can be found in the appendix as Listing A.1), we are granted access to the following information which will prove useful during accessibility analysis:

Figure 4-1: The Observe-Process-React loop involves translating the user interface into its base percepts, sending this information over HTTP, using this information to update the state tracker, and sending back an action based on the overall state of the current interaction.

1. The root ViewGroup(s) of this user interface, providing all View information. This includes both Activity roots, context menus, and dialogs.

2. Which device buttons are available.

3. The existence and functioning of various accessibility tools, such as TalkBack.

With this information, at any moment in time the Android user interface can be analyzed and converted into its percept and perceptifer breakdown.

4.1.2 State Simplification

Once the literal interface has been sent from the Android device to the testing server through a POST request, the techniques mentioned in section 2.3.2 are used to hash the literal interface into a single value. This single value, as well as a screenshot of

the user interface and a reference to the literal interface, are stored within a state machine. A `CondensedState` object holds all of these references and properties, and implements an equality function based on the calculated hash.

When the `CondensedState` is added to the state machine, the previous action taken is made to point to this state as well. If this state is already found within the state machine (i.e. the hash of this state is equal to an existing state), the previous action taken is made to point to that original condensed state.

Before making a decision on the next action to take, the state machine is also updated to include new "empty actions." By detecting the interactable components of the current user interface (i.e. determining which elements are clickable, swipe-able, and support key presses), edges are placed into the state machine that represent these actions, each of which point to a null state. When determining which states are unexplored, it is simply a check to determine which states still have edges with null destinations.

4.1.3 Action Decision-Making

Once the current state of the user interface has been processed, the state machine is updated to be in the correct state. At this time, the persona may take four different approaches to decide what the next action should be. Bility, by default, uses a persona that attempts to take all possible actions to fully explore the state space. First, it may decide to take the QUIT action, which terminates the test. This action is taken when either there are no more states with possible actions to try, when the maximum number of actions has been taken, or when a goal state has been achieved.

If not using a QUIT action, the persona may decide to take a user defined custom action, if one is defined. This option is useful during the initial exploration of the user interface, allowing the persona to reach most states before deciding to stress test with accessibility-related actions. For example, some applications may require the persona to get past a login screen, which requires specific inputs. The execution of a custom action is defined by the presence and absence of certain percepts within the current user interface (e.g. execute a login action if two text boxes with contents "Username"

and "Password" are present).

If a custom action is not given, the persona will automatically try one of two actions. If a transition is available from the current state that has not yet been explored (i.e. an empty edge is available at the current state), then the automaton will attempt to take that action. If an unexplored transition is not available from the current state, then the persona will search for the closest path to a state with an unexplored transition, plan a path of actions to reach that state, and then take that first action to go towards that state. This search is done using Dijkstra's algorithm [18], with edge weights corresponding to the time it took for the actions to be completed previously.

Due to the nature of the state simplification process, it is possible that a state is never reachable again. In these cases, the developer should define a custom action to detect this, and take an action accordingly. Otherwise, the persona will detect failures to reach a destination state (by means of unexpected traversals and repeated actions), and will mark that state as unreachable. These marked states can be ignored or included in final testing through the configuration of the test.

Once an action type has been decided upon, parameters for that action are generated if needed. For instance, for a SWIPE action, it is decided how much distance the persona should swipe, and in what direction this swipe should occur. Functions are defined which generate random parameters for each action (i.e. how much to swipe, where inside a clickable item the persona should simulate a click, which keypress to take, etc).

If the persona happens to fail on executing an action (for instance, a UI element moves or disappears before the action can take place, as is the case for some animations), the persona falls back to taking a random action. No matter what action is taken, the action is saved as a reference when adding the next state into the state machine, in the form of a `UserAction` object (which holds the type of action and parameters of that action).

4.1.4 Responding to Action Requests

One iteration of the Observe-Process-React loop concludes with responding to the decision made by the Bility library, detailed above. An endpoint `/api/getNextAction` is exposed by the Bility server, and acts as a queue that the Android device can query for an action. Once the action is requested and received by the Android device, the queued action variable is cleared. If the Android device reads nothing or null from this endpoint, it simply waits and queries at a regular interview until an action is provided.

Once an action is provided, the GSON library is used to cast the JSON response into a `UserAction` object. As explained previously, this object contains the type of action to complete and parameters for completing that action, such as swipe time or screen coordinates. Bility uses the `UiDevice` object to control the execution of an action. [16] For instance, this object provides the following useful methods:

1. `swipe` - A method that takes in a start (x,y) coordinate, end (x,y) coordinate, and number of steps. A swipe is made that starts at the start coordinate and ends at the end coordinate, with the number of steps defining how long the swipe should take (each step takes 5 milliseconds). Bility uses a default of 100 steps, which indicates a 1/2 second swipe time.

2. `click` - A method that takes in a (x,y) coordinate, and simulates a click on that position on the screen.

3. `pressKeyCode` - A method that takes in a key code and simulate that key code on the device.

Once the action is handled, Bility waits one second until reading the next action. From my testing, this is often enough time to allow the device to respond to whatever action was executed. This loop repeats until a QUIT action is received by the device, which terminates the Bility test.

4.1.5 Example: Text Style Switcher

As an example usage of this framework, I developed a simple application with a few accessibility issues. The application is called "Text Style Switcher", and is a simple one-screen interface with three buttons and a text box. Clicking each button changes the background and typeface of the text box. In particular, the "Make Regular" gives the text box a green background and normal typeface, the "Make Italic" button gives the text box a blue background with italic text, and the "Make Bold" button gives the text box a red background with bold text. Figure 4-2 shows the three main states of this application. Note that the application starts in the state with the green and normal font text box.

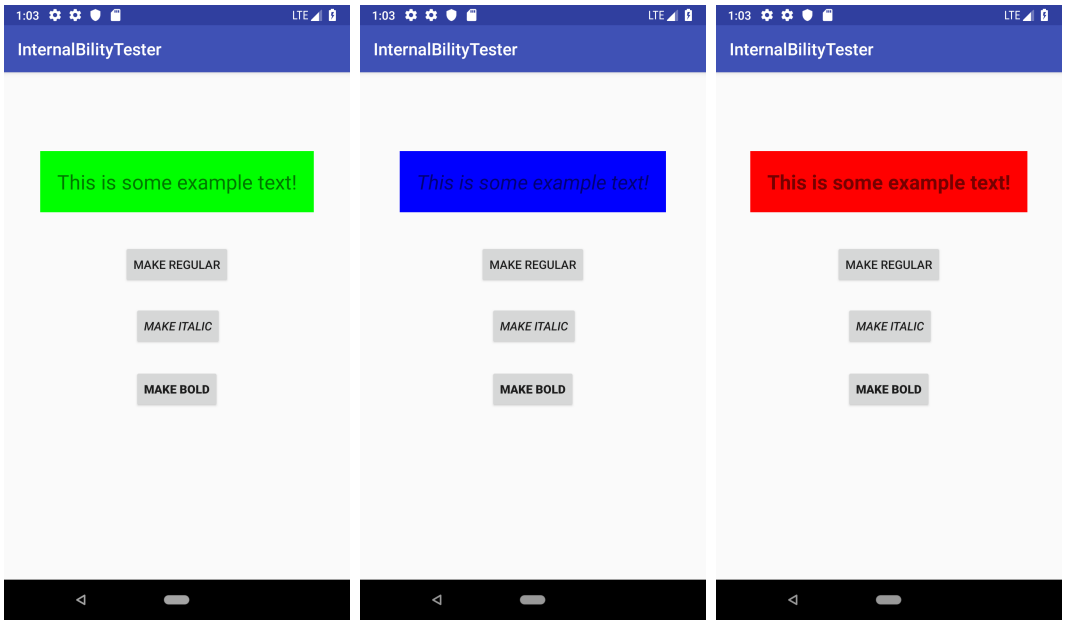

Figure 4-2: The three main states of the text switcher application, each navigated to using the buttons.

This application has the following intentional accessibility issues:

1. Every color (red, green blue) does not satisfy the minimum text contrast as required by WCAG 2.0 Principle 1.4.3. [11]

2. When focus is received on the first button (i.e. MAKE REGULAR), the intended action is done automatically. In other words, the text will change to normal

font and a green background when this button receives focus. This is a violation of WCAG 2.0 Principle 3.2.1, which requires that no change in context occurs when a component receives focus. [11]

Following the Observe-Process-React Loop, let's observe the iteration of one step for this user interface. First, the user interface is translated into UDL, meaning that the currently visible screen is broken down into its perceptifers. When the green box with normal text is shown, an excerpt[1] of the decomposition can be seen below:

```
LiteralInterfaceMetadata(id=1d11527a-db62-4747-b2ad-f1d1fd3dd83d)
    Perceptifer w/ ID e50d32b4-3d2d-4edf-888f-b36dae3d8986
        (R) Percept(type=FONT_SIZE, info=49.0)
        (R) Percept(type=TEXT, info=Make Bold)
        (R) Percept(type=LINE_SPACING, info=57.0)
        (R) Percept(type=LOCATION, info={left=542.0, top=1675.0})
        (R) Percept(type=ALPHA, info=1.0)
        (R) Percept(type=FONT_STYLE, info=BOLD)
        (R) Percept(type=SIZE, info={height=168.0, width=355.0})
        (R) Percept(type=TEXT_COLOR, info={colorHex=de000000})
        (R) Percept(type=BACKGROUND_COLOR, info=Color(color=FFFFFFFF))
        (V) Percept(type=VIRTUAL_NAME, info=android.widget.Button)
        (V) Percept(type=VIRTUALLY_CLICKABLE, info=true)
        (V) Percept(type=VIRTUAL_FOCUSABLE, info=false)
        (V) Percept(type=VIRTUAL_IDENTIFIER, info=2.131165217E9)
```

As you might be able to interpret from this description, the perceptifer shown above is the perceptifer representing the "Make Bold" button. Its percepts consists of observable properties (R, meaning Real) such as the font size, colors, and location on the screen, as well as virtual percepts (V), such as the name of the object and whether or not it has an attached click action.

[1]The entire decomposition is quite large to reproduce here, but can be viewed at `https://github.com/vontell/Bility/blob/master/results/green_literal_interface.txt`

Now that this interface has been converted into a literal interface within UDL, it can be converted into a condensed state. Using the hashing technique described in section 2.3.2, each container and each object is hashed, resulting in an overall hash for the entire user interface. In this case, the font size, line spacing, alpha, font style, text color, background color, and name are used within the hash. This hash is completed using standard hashing techniques in Java, by collecting these percepts into a Set, and then hashing that Set.

Once the condensed state is created using these hashes, the persona analyzes the condensed state to first determine if this application has been reached before. Since the interface is represented as a hash, this is a simple check. In this example, this is the first state that has been encountered, and therefore this state is simply added to the automaton as the start state. However, if an action was taken to reach this state, then this state is added (or found) within the automaton, and a transition is made from the previous state to this current state using the action that was last taken.

Finally, the persona decides which actions are available, and which action it would like to take. Using the persona, each possible key press, clickable perceptifer, and swipeable perceptifer is added as a potential action from the current state, if the transition has not been taken before. These actions are easy to find, as the persona simply filters perceptifers for the correct `VIRTUAL_FOCUSABLE` and `SCROLL_PROGRESS` percepts. Using the algorithm described in the previous section, the persona then chooses an action to take from this state, queuing it on the `/api/getNextAction` endpoint.

Once the Bility Android library requests this action, the action is taken, parsing the action type and action parameters from the request (e.g. if click, here are the coordinates to click). The loop then continues, parsing the new user interface into UDL.

Once a `QUIT` action is reached on this application, a graph representation of this user interface is saved. Figure 4-3 shows the output for the text switcher application (without key-presses included, as that diagram is quite large for printing). This state machine is automatically generated with no input from the developer; the developer

simply points Bility to their application, starts the test, and waits for the test to complete.

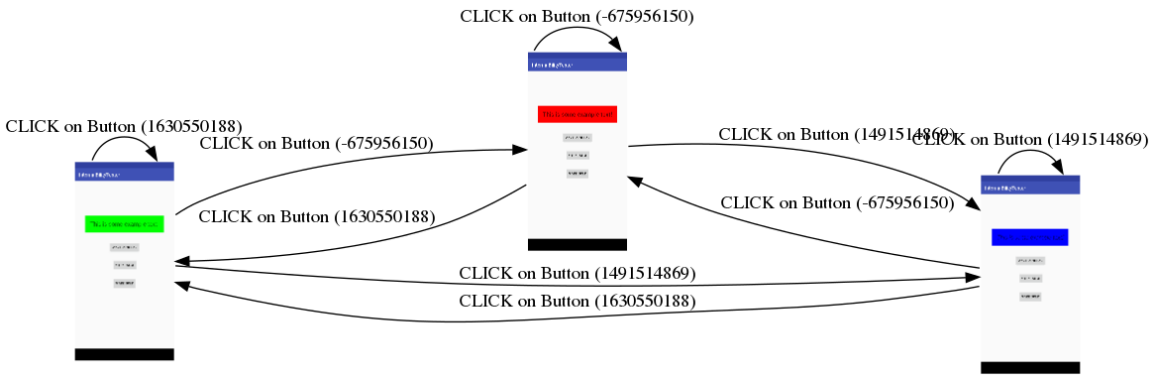

Figure 4-3: A simple state machine representing the text switcher application (using only click actions). Integer values represent the IDs of each button as assigned by the Android Platform at runtime.

4.2 Detection of Accessibility Issues

The Observe-Process-React loop does a best-effort navigation of the user interface that is being tested. During this process, a state machine is constructed, containing both static and dynamic information about the functioning of the user interface. Using this information, accessibility issues can now be detected.

The most popular standard today for testing accessibility is WCAG 2.0 [12], which outlines rules for accessibility satisfaction on websites. Although made for the web, many of the standards outlined in WCAG 2.0 can be analyzed on mobile devices.

Being that Bility is a framework upon which accessibility testing can occur, I have implemented a few of the WCAG principles on top of Bility as an example of its versatility and usefulness. In particular, Bility includes implementations to detect the following static and dynamic issues as outlined by WCAG principles [11]:

Static Issues:

- **1.1.1 Non-text Content**: All non-text content that is presented to the user has a text alternative that serves the equivalent purpose, except for a few exceptions

such as decorations. (Level A) Additionally, although not required by WCAG, an issue is also reported if two screen readers descriptions have the same content, which can cause ambiguity for blind users.

- **1.4.3 Contrast (Minimum)**: The visual presentation of text and images of text has a contrast ratio of at least 4.5:1, except for large text which requires a contrast of at least 3:1. (Level AA)

- **1.4.6 Contrast (Enhanced)**: The visual presentation of text and images of text has a contrast ratio of at least 7:1, except for large text which requires a contrast of at least 4.5:1. (Level AAA)

- **2.5.5 Target Size**: The size of the target for pointer inputs is at least 44 by 44 CSS pixels. (Level AAA, note that this is part of WCAG 2.1)

Dynamic Issues:

- **2.1.1 Keyboard**: All functionality of the content is operable through a keyboard interface without requiring specific timings for individual keystrokes, except where the underlying function requires input that depends on the path of the user's movement and not just the endpoints. (Level A)

- **3.2.1 On Focus**: When any component receives focus, it does not initiate a change of context. (Level A)

- **3.2.5 Change on Request**: Changes of context are initiated only by user request or a mechanism is available to turn off such changes. (Level AAA)

In the following section I detail the methods used to detect these issues using the Bility framework, as well as mention a few other techniques that can be used to find other violations of WCAG 2.0 principles.

4.2.1 Static Accessibility Issues

Missing text-alternatives

A common technique to increase the accessibility of an application requires setting text-alternatives for non-text elements. For instance, every image within a user interface, whether it be an icon or photograph, should have some text alternative that explains either the meaning of that icon or the contents of that image.

Within Android, this is done by setting a content description on images and other View objects. As this content is not usually seen through direct interaction with the user interface, there are two different options for detecting these content descriptions - turning on TalkBack and recording audio percepts, or reading off content descriptions and recording virtual percepts representing content that would be spoken by TalkBack. The latter option is faster and simpler to detect in practice, and as such Bility uses a `VIRTUAL_SCREEN_READER_CONTENT` virtual percept to detect such issues.

Given a perceptifer, the process is as follows:

1. If this perceptifer holds `TEXT` percepts, and that text is not an empty string, report no issue. If the string is empty, report an issue if no non-empty `VIRTUAL_SCREEN_READER_CONTENT` percept is found.

2. If this perceptifer holds a `MEDIA_TYPE` percept of type `IMAGE`, and no non-empty `VIRTUAL_SCREEN_READER_CONTENT` percept is found, report an issue.

3. If this perceptifer holds an `INVISIBLE` percept, do not report an issue. `INVISIBLE` indicates that this element is either a container or decorative.

With this logic, it can be verified whether or not an application is accessible with respect to WCAG 1.1.1 Non-text Content. In the case when visible text is available to be read off by the screen reader, then no issue is reported. If the text content of anything to be read out by the screen reader is the empty string, then a screen reader may fail to report anything, meaning that this element should be reported as a failure. Finally, if the element is an image or non-decorative element with no text, a non-empty screen reader description must be available.

In addition to this logic, a quick check is also made to determine if all screen reader content percepts within an literal interface instance are different. If multiple screen reader content percepts with the same text are found within the same screen, an issue is reported, as these items cannot be differentiated by a blind user.

Contrast of text and icons

Contrast is calculated by calculating the relative luminance of a foreground color to a background color. The Android UDL implementation provides text color percepts for text content, background color percepts for all views capable of holding a background color, and foreground color percepts for images (such as icons).

Given a perceptifer, a `getPerceptsOfType` helper function, which takes in a percept type and perceptifer, is used to get percepts of the type `TEXT_COLOR`, `FOREGROUND_COLOR`, and `BACKGROUND_COLOR`. Text color is used as the foreground if available - otherwise, the foreground color is used.

Next, the luminance L of each color is calculated [34], as defined by the following formulas, given an RGB value where each color component is in the range of 0 to 255:

$$
\begin{aligned}
R_n &= R \text{ component normalized from 0 to 1} \\
G_n &= G \text{ component normalized from 0 to 1} \\
B_n &= B \text{ component normalized from 0 to 1} \\
R_L &= \frac{R_n}{12.92} \text{ if } R_n <= 0.03928 \text{ else } \Big(\frac{R_n + 0.055}{1.055}\Big)^{2.4} \\
G_L &= \frac{G_n}{12.92} \text{ if } G_n <= 0.03928 \text{ else } \Big(\frac{G_n + 0.055}{1.055}\Big)^{2.4} \\
B_L &= \frac{B_n}{12.92} \text{ if } B_n <= 0.03928 \text{ else } \Big(\frac{B_n + 0.055}{1.055}\Big)^{2.4} \\
L &= 0.2126R_L + 0.7152G_L + 0.0722B_L
\end{aligned}
\tag{4.1}
$$

These formulas are derived from the intensity at which each color is perceived by the human eye.

After the luminance of the background color (L_B) and luminance of the foreground

color (L_F) has been calculated, the contrast C is calculated as follows:

$$C = \frac{L_B + 0.05}{L_F + 0.05} \text{ if } L_B > L_F \text{ else } \frac{L_F + 0.05}{L_B + 0.05}$$

The resulting value can be in the range of 1 to 21, which can now be used to evaluate a set of colors against contrast criteria. For instance, WCAG 2.0 Principle 1.4.3 requires that contrast between foreground and background be at least 4.5:1 (i.e. $C >= 4.5$). While WCAG 2.0 Principle 1.4.6 requires contrast between foreground and background to be at least 7:1 (i.e. $C >= 7$). However, if the foreground color is from text, and the text is deemed as large (i.e. 18 point font or 14 point bold font), a contrast ratio of only 3 and 4.5 are required, respectively. In other words, once the colors of the background and foreground are determined, determining contrast situations which cause inaccessibility is quite simple.

However, calculating the foreground and background colors of elements within a user interface are not always straightforward. Transparency in colors may require knowledge of underlying elements and colors, and colors may not be solid or continuous (in the case of a gradients). WCAG 2.0 calls for explicit declaration of colors, ignoring issues of transparency. [33] However, one may address some these issues by blending foreground and background colors using alpha blending techniques. [6, 31] Assuming a white base background color, the `blendColors` function takes in a stack of colors from background to foreground, returning a color value which equals the real composite color after considering the effects of each alpha channel (full code for this function can be found in the appendix as Listing A.3).

As seen in Figure 4-4, Bility has found a contrast accessibility issue, referring to WCAG 2.0 principle 1.4.3. Hovering over the issue item, Bility presents a screenshot of the screen in question, as well as a highlighted border around the violation. Hovering over that UI component, we see that this is a TextView with a text color of `#8a000000` and background color of `#ff0000ff`. Bility also presents a few pieces of information regarding the font properties of this text.

Using the alpha blending technique mentioned previously, we see that the accessi-

bility issue reports a foreground color of `#ff000075`, which against the background has a calculated contrast of about 1.94. This is less than the required contrast of 3:1 for large text (which this TextView qualifies as), and so an issue is reported, providing information and resources that a developer can use to locate and fix the issue.

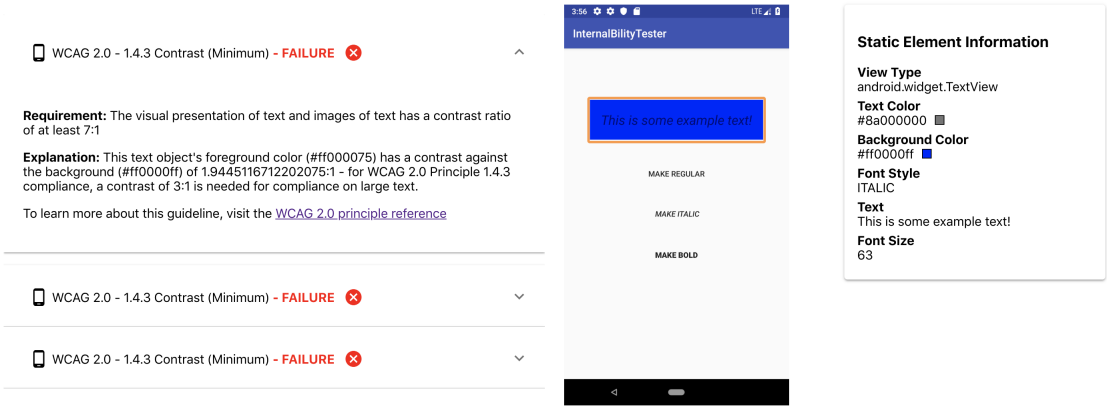

Figure 4-4: An example violation of WCAG 2.0 Principle 1.4.3, which calls for a contrast of 3:1 for large text.

Similar to Principle 1.4.3, Bility will find and report similar violations for Principle 1.4.6, which requires higher contrasts for AA-level satisfaction.

Minimum target size

If the touch target size of a user interface element is very small, then a user may have trouble interacting with it if they have a motor disability, such as Parkinson's. [26] WCAG 2.1 addresses this by requiring a touch target size of at least 44 pixels by 44 pixels for all elements that can be clicked. Within the Bility framework, this is as simple as filtering for all perceptifers that have a `VIRTUALLY_CLICKABLE` percept, and then asserting that each of these elements has a `SIZE` percept with size at least 44 pixels by 44 pixels. If this is not the case, then an issue is reported. For instance, see Figure 4-5 for an example of this detection.

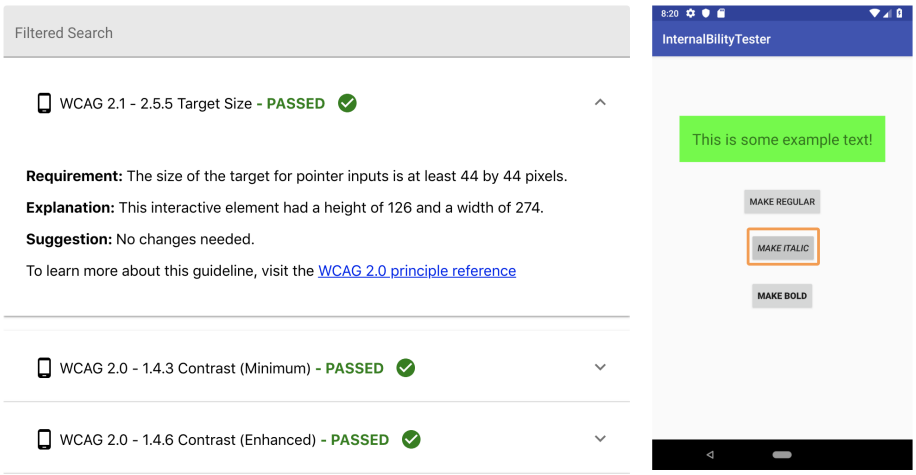

Figure 4-5: An example of a passing instance of Principle 2.5.5. A minimum width and height of 44 pixels is required, and this button is larger in both directions (274 pixels by 126 pixels).

4.2.2 Dynamic accessibility issues

Navigation with keyboard only

While most mobile devices are interacted with using a touch screen, some users with motor disabilities interact with their devices using keyboards and switches. Determining whether an interface is navigable and functional using only a keyboard-like device is essential for these users. WCAG 2.0 principle 2.1.1 therefore calls for user interfaces to be completely functional using only the keyboard (unless an input specifically requires non-keyboard input). [11]

Asserting that the user interface is navigable using only a keyboard can be done by removing edges from the automaton that are not keyboard presses, and then making sure that all states are still reachable using only these actions. This essentially means that only keyboard presses can be used to fully navigate the user interface.

If A is the original automaton, and B is the automaton with all edges removed that are not keyboard interactions, then the following must be true: if a state Y in A is reachable from state X, then state Y must be reachable from state X in automaton B through some path. Simply put, this means that if a user can get from one state to another using typical interaction methods, they must also be able to get between the

same states with a keyboard.

There is one exception to this rule due to user interface elements becoming focused during keyboard and switch usage. In the case of keyboard input, a user interface will often have no focused elements before any keyboard input has been sent. Once keyboard input has been used, a focusable percept will be included within the user interface on some perceptifer. Once entering into this state of moving focus between user interface elements, it can be impossible to remove this focus using key presses (i.e. usually the only way to remove this focus is to click or swipe the interface). Since the implemented hashing of user interfaces keeps track of focused items, this means that the state machine which removes all edges except for key presses may have states that seem "unreachable" which in fact are only unreachable due to the fact that they have no focused elements. In the case that state X cannot reach state Y in automaton B, but a path exists in automaton A, we first check that $hash_{nf}(X)! = hash_{nf}(Y)$, where $hash_{nf}$ is a hash function similar to the accessibility hash function presented early, but excludes the tracking of focused items. If X and Y are equal by this measure, then they are simply different through a focused item, and keyboard reachability is therefore ignored (otherwise, this lack of path from X to Y is reported).

We can see an example of this detection in Figure 4-6, which represents the Text Switcher application with a new modification. In this version I have removed the logic where focusing on the "normal" button would change the text type and background, and have instead disabled the ability to focus on the "bold" button. Within the state machine at the top, you will find that every state is reachable from every other state through some series of clicks or key presses. The state machine on the bottom has removed all click actions, and has left only key presses such as LEFT, RIGHT, UP, DOWN, TAB, and ENTER.

This state machine reveals that not every state is reachable from every other state. First, there is a green, red, and blue state that all have no edges going into them. As mentioned previously, these states are the states that represent the "before keyboard focus has been introduced" state. These are detected through the logic mentioned above, and are ignored as issues.

However, the clique of red states in the bottom left have no edges coming in from the group of blue and green states, meaning that the bold button logic can never be applied when starting from a blue or green state. Using an algorithm such as the Floyd-Warshall Algorithm [19], Bility can detect this lack of a path from the blue and green states to the red states, and will report each origin-destination pair as a violation of WCAG principle 2.1.1.

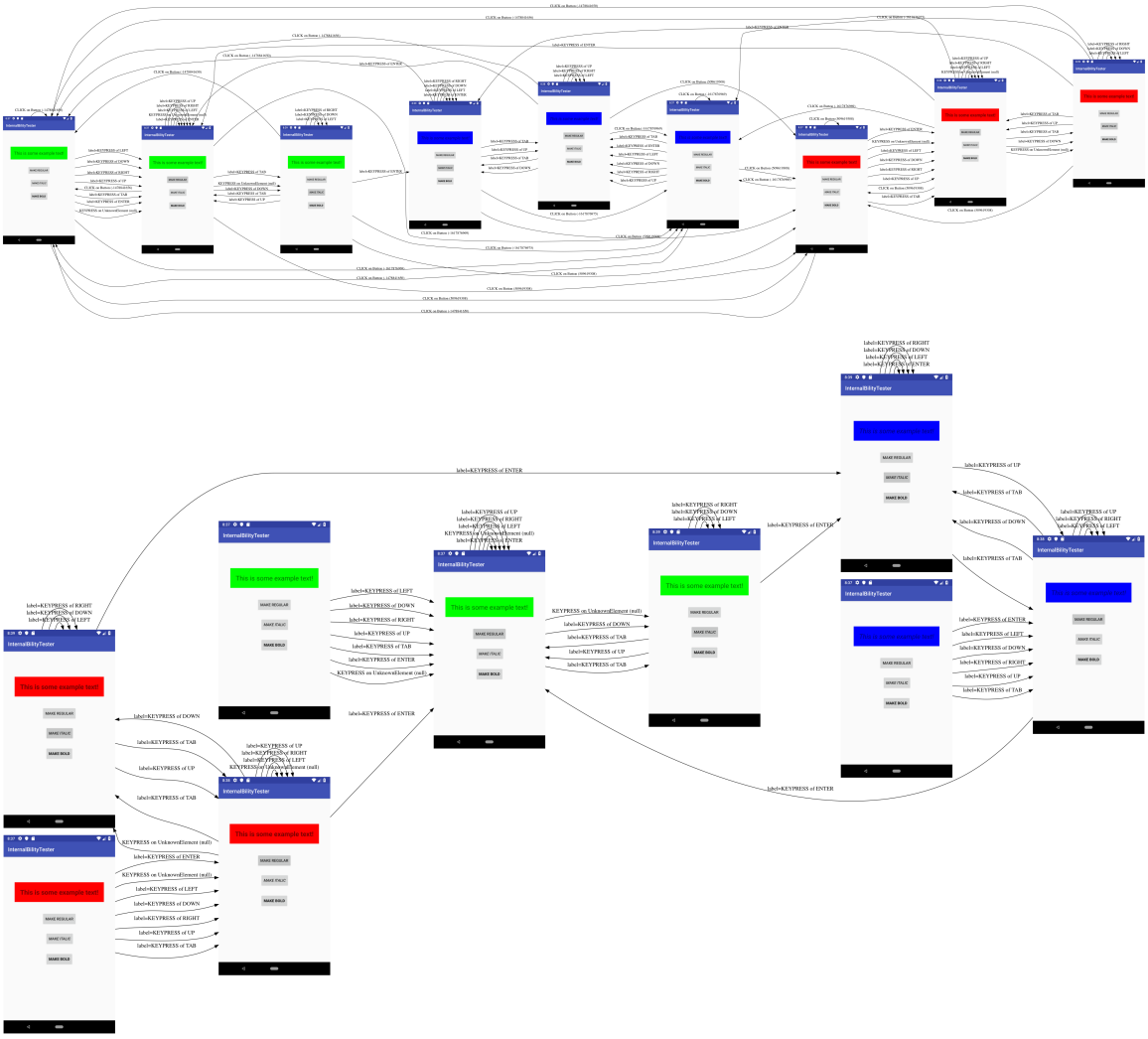

Figure 4-6: The state diagram at the top shows all action types and states reached using those actions. In that diagram, there is at least one edge going into each state. The bottom diagram is the same state machine but with only edges that represent key presses. Notice that three states (one red, one green, and one blue) do not have any edges into them.

Focus does not cause change of context

As mentioned previously, the use of keyboards often cause elements to become focused, or elements such as buttons, text, and images to become "selected", allowing the user to interact with them by simulating actions such as clicks (often by clicking the "enter" button on their keyboard).

Principle 3.2.1 of WCAG 2.0 requires that when an element receives focus, a change in context does not occur. [11] A change of context is defined as any change of the user interface that is more than a change of content, and is a change that may disorient a user. For instance, switching to a new user interface or changing the layout of the current interface are both changes in context.

The state machine built by the Bility persona is well suited for detecting this issue. First, all edges except for focus-inducing edges are removed. In other words, only `KEYPRESS` edges should remain. Next, we need to confirm that the destination state of each edge is not a change of context from the origin state for that edge. With Bility, we simply use the **change of context hashing routine** - if the hashes of the two states are equal, this transition does not constitute a change of context. If these hashes are not equal, Bility reports a violation of WCAG 2.0 Principle 3.2.1.

The change of context hashing routine is similar to the accessibility hashing routine discussed in section 2.3.2, which tracked alpha, background color, font size, font style, line spacing, naming, focus, and text color percepts. The context hashing routine, on the other hand, tracks the same percepts except for the focus percept. In other words, the change of context hashing routine checks if the two states are equal in accordance to the accessibility hash, but ignores any focused items.

No action results in no change

Principle 3.2.5 of WCAG 2.0 requires that changes of context are initiated solely by user requests (or that a mechanism is provided to turn these off). This means that if the user does not take an action, not much should change within the application.

A developer can configure Bility to wait before taking any action (or essentially

create a no-operation action). By default, this action is taken randomly at a certain frequency by the base persona, and also occurs when computation is slow on the testing server.

Given the state machine representing the application, each edge that is not a NONE action is removed. For each NONE edge, it is then checked that it is a self-edge (i.e it is a self loop starting and ending in the same state), or if that edge is not a self-edge, that the two states are the same as defined by changes in context. Similar to checking that change in focus does not change context, the same hashing technique is used here to determine if the two states are of the same context. If they are not, then a violation of WCAG 2.0 Principle 3.2.5 has been found.

Chapter 5

Evaluation

I evaluated Bility in two ways - by comparing its results with that of the Google Accessibility Scanner and a user study to figure out which accessibility issues can be identified by human developers without using any tools.

I chose three open source applications found on GitHub to undergo accessibility testing: a sound-recording app called **SoundRecorder**, a travel companion app called **Travel-Mate**, and a music player app called **Timber**. Each app was pulled from GitHub, compiled by Android Studio, and installed on a Pixel 2 XL device. Each app underwent testing using the Google Accessibility Scanner, a human tester, and the Bility library, with results summarized in each section.

Overall, we found that Bility tracked a wider variety of accessibility issues (three additional issue types) when compared to the Google Accessibility Scanner, avoided over-reporting accessibility issues (sometimes with a factor of eight or nine times fewer issues reported when removing repeated issues), and gave more information regarding both passing and failing accessibility test cases (such as detailed View information). It was also found that humans were better at finding cognitive-related issues when compared to Bility and the Google Accessibility Scanner, such as ensuring that labels and section headers accurately described their contents and function.

5.1 Google Accessibility Scanner Results

The Google Accessibility Scanner was used to test each screen found within an application. The process of using the Google Accessibility to find accessibility issues was defined as follows:

1. Navigate to a state of the user interface that was not yet tested.
2. Click the Google Accessibility Scanner button and share the results to the tester (in this case email).
3. Repeat items 1 and 2 until the human tester is confident that all states have been covered.
4. Once finished, compile the results into a complete useful report.

In the case of step 3, the source code of the application and knowledge of previous interactions with the applications were used to cover as much of the application state space as possible.

SoundRecorder

The SoundRecorder application [22] is a simple Android application that supports recording and playback of audio. It has an about page, a playback dialog, and information about previous and active recordings.

While navigating the SoundRecording application, I found five different app states, related to starting a recording, actively recording, viewing existing recordings, viewing settings, and viewing about information. Overall, the Google Accessibility Scanner found 10 issues. Three of these issues were related to WCAG 2.0 principle 1.1.1, which requires text alternatives for non-text content (in this case on buttons and sliders). The remaining seven issues were related to WCAG 2.0 principle 1.4.3, which requires certain levels of contrast between foreground text and background. However, one of these issues reports contrast levels between an icon and its background. Screenshots of these issues can be found in Figure 5-1.

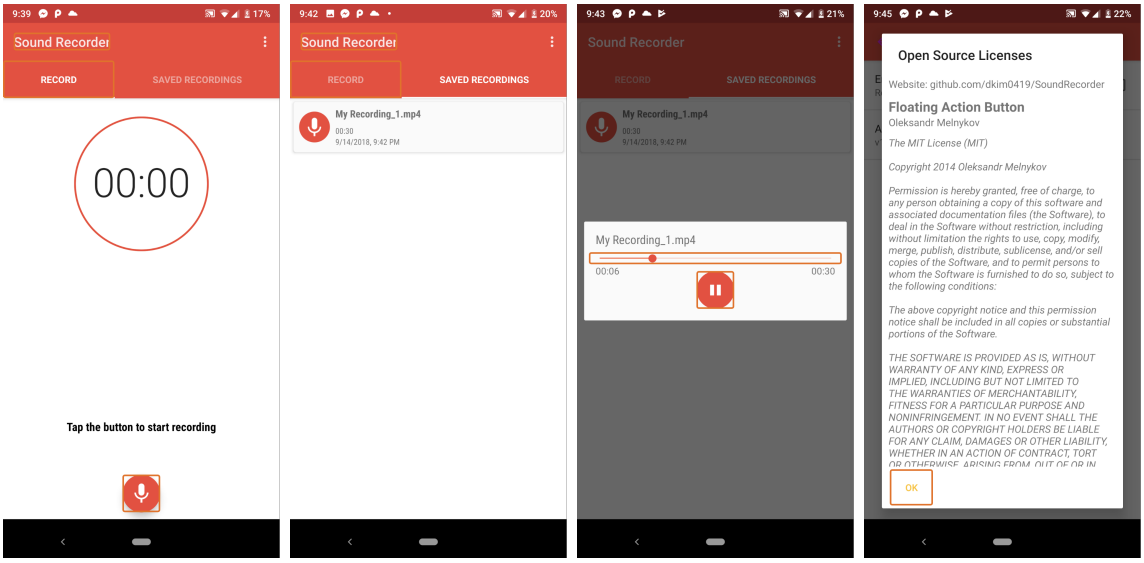

Figure 5-1: Screenshots of some SoundRecorder accessibility issues reported by the Google Accessibility Scanner, such as small touch target size and poor contrast (orange boxes define where the issues are found).

Overall, for the SoundRecording application, the Google Accessibility Scanner was able to find missing screen reader labels, poor text contrast, and poor image contrast.

Travel-Mate

The Travel-Mate application is a feature-rich travel companion application [32], with capabilities for trip planning, viewing weather, find points of interest, and more.

The Travel-Mate application has many more screens than the SoundRecorder app, with a total of 20 screens evaluated on the Google Accessibility Scanner, with each screen being a distinctively different part of the application (e.g. the notifications screen, login screen, weather screen, etc.) A few of these screens with highlighted issues can be seen in Figure 5-2.

Overall, the Travel-Mate application reported a total of:

- 32 issues with low text-to-background contrast.
- 53 issues with touch targets being too small (smaller than 48dp in width or height).
- 18 issues where screen reader content for items was repeated among UI elements.

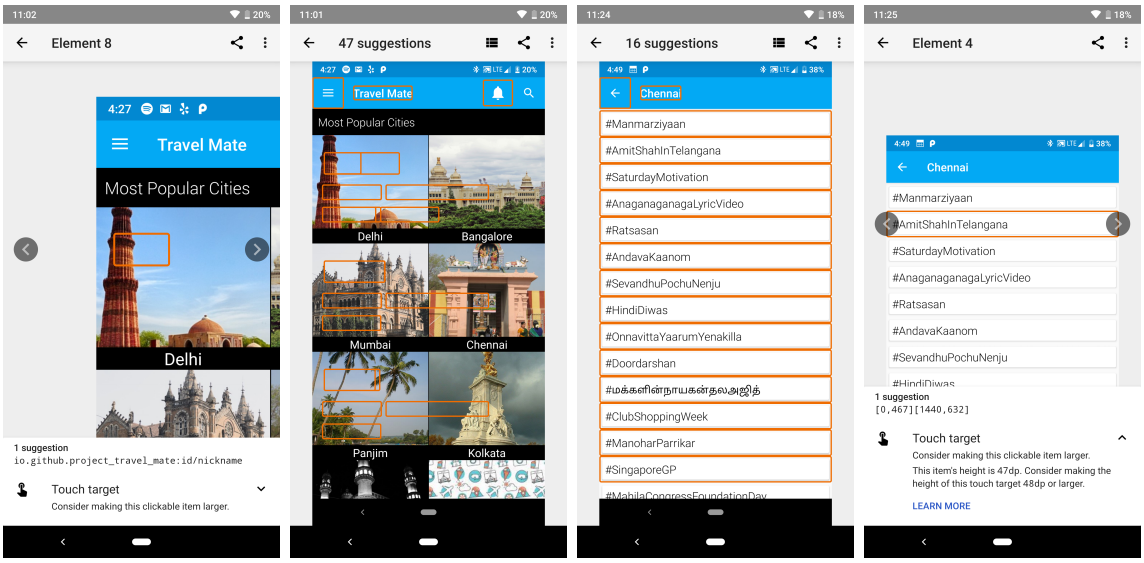

Figure 5-2: Screenshots of the Travel-Mate's detailed issue report within the Google Accessibility Scanner. Issues found include small touch targets and poor contrast.

- 9 issues where a UI element was not compatible with Google Accessibility Scanner's detection techniques.
- 8 issues where clickable items overlapped.
- 13 issues where image or icon contrast with background was low.
- 2 issues where images or icons were missing content descriptions.

A deeper analysis of these results reveal three limitations or errors with the Google Accessibility Scanner. First, the Scanner failed to recognize situations where user interface elements were hidden by other UI elements. From the bordered regions in Figure 5-3, you can see there are a few issues found with elements that appear to be hidden underneath the location images. These elements are not seen through normal usage of the application, and as such their issues would not be relevant to users of the application. These additional reported issues add bloat and time to the developer's efforts in improving the accessibility of their application. Upon further inspection of the code, it seems that these views are part of a custom View used to spotlight various aspects of the home screen on first use.

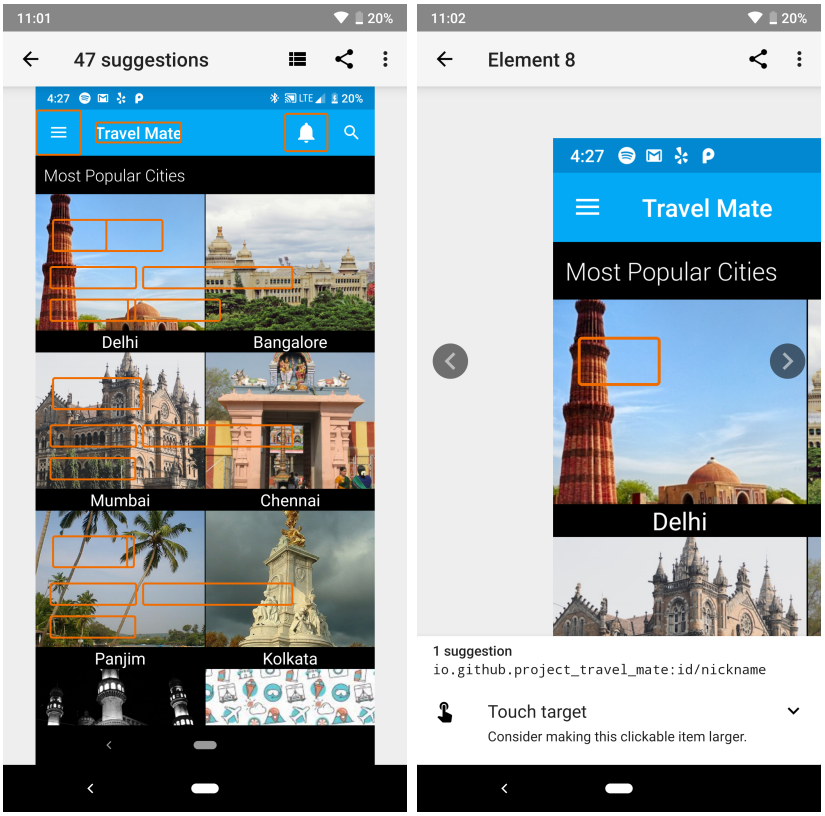

Figure 5-3: Screenshots of the Google Accessibility Scanner reporting overlapping clickable items within the Travel-Mate application

Another apparent limitation of the Google Accessibility Scanner is its inability to realize repeated issues within a dynamically repeated element. For instance, take the list of repeated clickable elements in Figure 5-4. In this example, the clickable TextView is programmatically repeated from one base instance of a TextView declaration, either created from the `TextView` object directly or by inflating a separate XML file. However, although fourteen issues are reported to the developer, there is really only one instance of the issue, which when fixed handles all 14 issues. This over-reporting can mislead developers and give the appearance of overhead, which may lead the developer to abandon their accessibility increase endeavor.

A few other limitations seen in these results is that the Accessibility Scanner has some trouble with custom View types. For instance, this app uses a `TextInputLayout`, which the Scanner reports back with the following message: *This item's type* ***`TextInputLayout`*** *may not be resolvable by accessibility services. Consider*

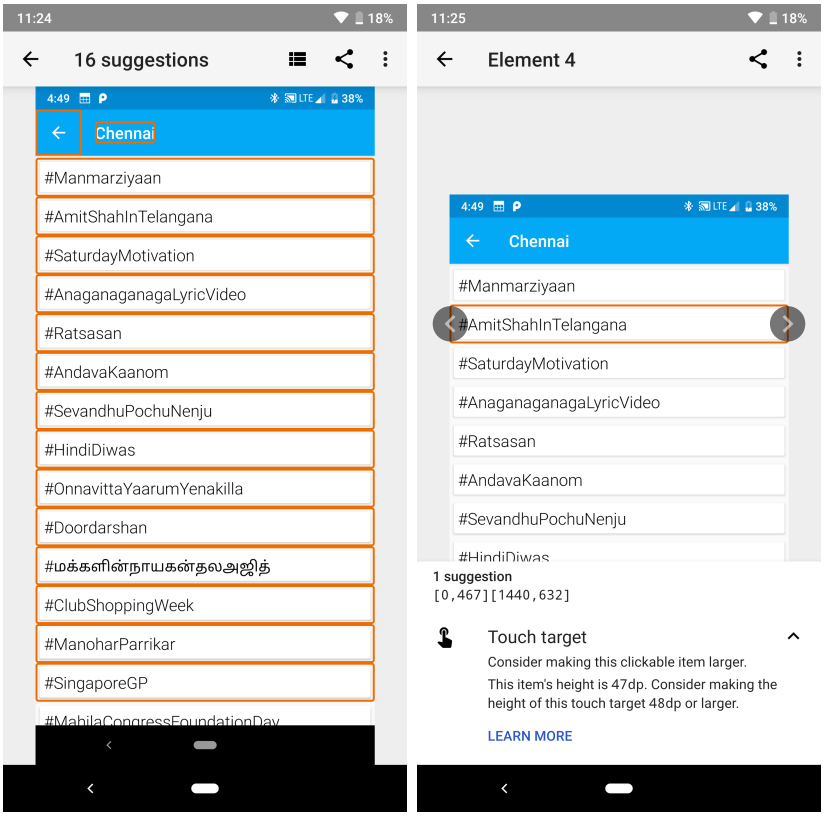

Figure 5-4: Screenshots of the Google Accessibility Scanner reporting touch target size issues for a repeated UI element.

using a type defined by the Android SDK. The Google Accessibility Scanner relies on accessibility services and accessibility nodes, which are not always available for custom views.

Throughout this application, however, I found that the Accessibility Scanner is able to report when screen reader content of various components are repeated on the same screen. A repeated description can cause confusion when navigating the screen with screen readers (e.g. a repeated item with a non-parameterized description will have the same content to read out for every item). This makes it difficult for a user to discern different items from one another.

Timber

Timber is a music and audio feedback application [27], built to display the various components and design patterns found in Google's Material Design. It features a few

audio selection and playback screens.

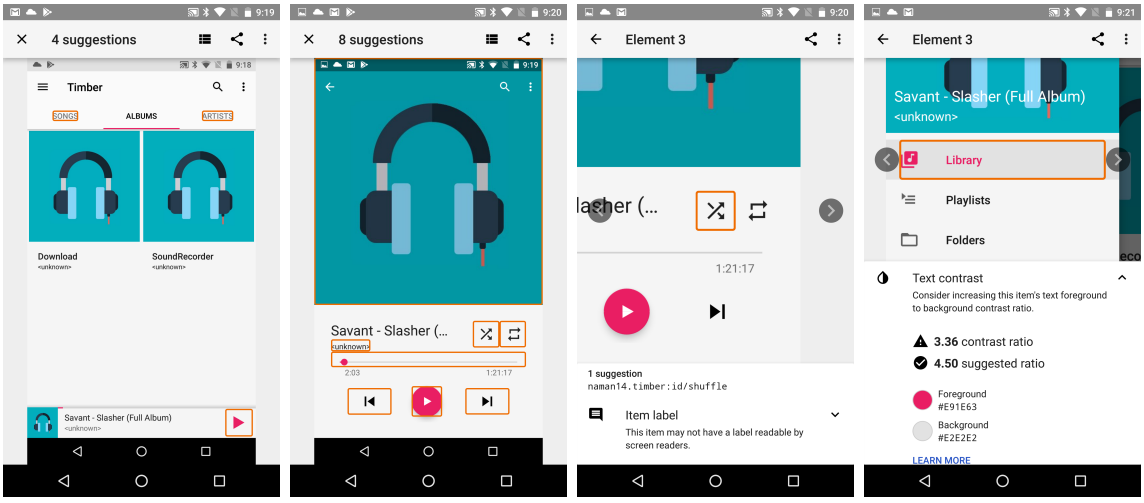

Figure 5-5: Example results from the Google Accessibility Scanner on the Timber application, revealing issues such as poor contrast and missing context descriptions.

The Timber application was evaluated by the Google Accessibility Scanner on 16 screens, varying in both state (e.g. music paused, music resume) and theme (e.g. dark theme, light theme). Overall, the Accessibility Scanner reported a total of:

- 17 issues with low text-to-background contrast.
- 56 issues with touch targets being too small (smaller than 48dp in width or height).
- 2 issues where screen reader content for items was repeated among UI elements.
- 10 issues where image or icon contrast with background was low.
- 73 issues where images or icons were missing content descriptions.

The Timber application is largely inaccessible due to the absence of content descriptions on icons, as well as the small size of these icons. However, many of these issues are simply the same issue repeated on dynamic elements, as seen in Figure 5-6. Within this screen report, 9 issues are reported for the small options icon for each item (the three vertical dots icon). However, this user interface element is truly one

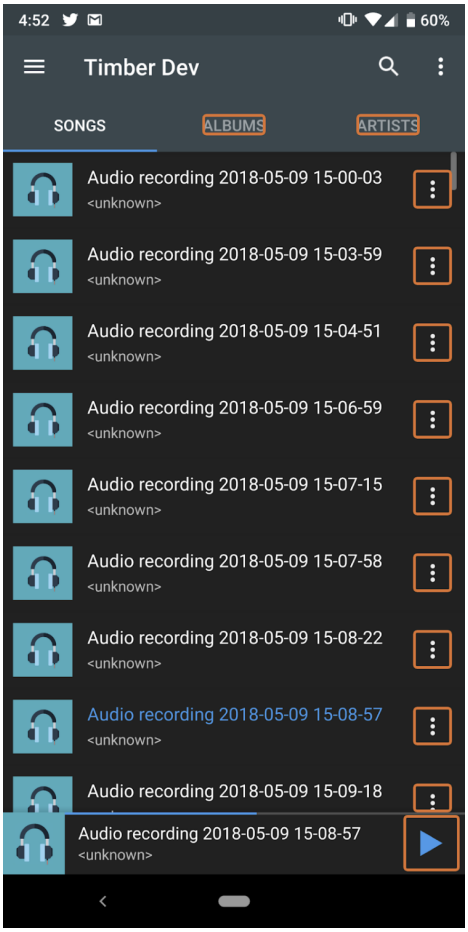

Figure 5-6: A screenshot of the Google Accessibility Scanner results for a page with dynamically repeated elements. Note that each orange highlight is considered to be a single issue, although most of the option menu issues are derived from one View definition.

definition repeated multiple times, and as such the developer only has one issue to address.

Overall, the Google Accessibility Scanner found a wide range of static accessibility issues (such as small touch target, contrast, repeated screen reader content, and missing screen reader content), but the number of issues reported is quite large, due to its inability to handle the dynamic generation of Views.

Overview

The Google Accessibility Scanner was used to evaluate the accessibility of three real applications on the Play Store (and also available as open source projects). Overall,

it was found that the Google Accessibility Scanner was able to achieve the following functionality:

- Accessibility testing on compiled applications without project files.
- Detect missing content descriptions (closely related to WCAG 2.0 Principle 1.1.1)
- Detect poor contrast on text and icons against their backgrounds (closely related to WCAG 2.0 Principles 1.4.3 and 1.4.6)
- Detect small touch target sizes (closely related to WCAG 2.1 Principle 2.5.5)
- Detect repeated screen reader descriptions
- Detect overlapping user interface elements that may interfere with clicking abilities

However there are also a few limitations:

- Issues with elements that are dynamically repeated are recognized as separate issues.
- Only static issues are detected, on a screen by screen basis - the dynamic functioning of the application is not tested for accessibility.
- No automated testing is available for the Accessibility Scanner - a developer must navigate through the application, run a test, and send the results to their computer.

I found that the Google Accessibility Scanner was really good at finding static issues related to colors and text. However, it lacked dynamic accessibility issue detection and was tedious for use (in both the effort needed to run the test and the effort needed to understand and process the results).

5.2 User Study Results

In order to provide a baseline for determining the ability of a developer to find accessibility issues within an application, I ran a user study with fourteen study participants. Each participant was asked to manually inspect and identify accessibility issues they found within each of the five applications. The full questionnaire and instructions given to the participants can be found within a Google Form[1], which also includes all responses.[2] These participants were chosen not for Android or mobile development experience, but rather were chosen because they had taken a user interface course at MIT (6.813 or 6.831), and had some knowledge of design and accessibility issues.

The goal of this user study was to determine a baseline for the types of accessibility issues that a developer may find on easily their own without using external tools. Based on this study, I found that humans are capable of finding issues to cognitive overhead and general usability, but are not as skilled in finding issues related to visual and audio components. In addition, humans had a hard time finding dynamic accessibility issues such as navigation issues and keyboard traps.

SoundRecorder

For this application, the developers were able to find issues related mostly to motor skill disabilities and cognitive disabilities. For instance, comments such as "The button to start/stop recording could maybe be slightly larger to make it easier to press for motor-impaired" and "The buttons are a bit small, which might be difficult for motor-impaired users to click" showed that touch target size was an issue that can be quickly found by a human tester. For issues related to cognitive disabilities, developers had comments such as " The 'recording saved to...' message has a lot of text and only appears for a short time so a cognitively-impaired user might not be able process it in time" and "It's nice and simple for cognitively impaired people".

[1] `https://goo.gl/forms/IEKqveCCEqIBrZpm2`

[2] `https://docs.google.com/spreadsheets/d/1wwKJIBlTWpejORIcfbDzi2CGn_3AD_EmZgcYo5F7RK8/edit?usp=sharing`

When it came to detecting issues for users with vision impairments, the participants also found issues with small text sizes. However, none of the participants mentioned seeing any issues with contrast. Only six participants mentioned the use (or lack) of audio affordances, audio feedback, or screen reader labels for blind users.

Finally, most participants did not have many comments regarding users with hearing impairments. Only four participants mentioned explicit usability issues for these users, explaining that a sound recording app should provide some sort of captioning system, or provide better visual feedback for recording and playback.

Travel-Mate

For this app, the complexity, nesting, and number of possible screens and states led the participants to find issues related to how each disability would be affected by this complexity. For instance, one participant said "There is a lot of navigation for this app, so it might be hard with a screen reader, though it is quite structured which would then make it easier once the menu options had been read to find the correct button," while another participant explained that "Travel-Mate presents a lot of information, but it does not have a clear flow for a user to follow, which could be challenging for a cognitively-impaired user".

In terms of visual and hearing impairments, the participants did not find many issues, explaining that contrasts were good and that no audio features made hearing impairments irrelevant. However, a few participants did mentioned that small font sized could affect usability for some users.

Timber

Most participants felt that the Timber app was quite easy to use and simple, and that its simplicity made use by cognitively impaired users easy. For visually and motor impaired users, one participant suggested "maybe make the font bigger and buttons bigger" while another mentioned that "the buttons are small, which could be a problem for people with motor impairments." Unfortunately, most participants did not mention any accessibility issues for deaf users, as many explained that "I don't

think deaf people typically listen to music."

Overview

The manual human testing of accessibility for each application gave a few insights into the kinds of issues that developers may be able to find on their own, without any extra tools. These issues were most often related to cognitive overhead, where the study participants were able to identify when a part of an application may be confusing, tedious, or not informative enough. For visual impairments, some participants understood the use of screen readers, but only two of the participants attempted to use Android TalkBack feature to determine the accessibility in this manner. Issues related to visual impairments where therefore geared toward an analysis regarding potential users who are not completely blind, focusing mostly on font size and some contrast issues. For hearing disabilities, participants mentioned the importance and lack of visual feedback when audio feedback was used, while motor disabilities were addressed in the form of issues with touch target sizes.

In a loose mapping to WCAG 2.0 principles, I found that the participants were able to identify issues for principles 1.2.1, 1.2.2, 1.2.3, 2.2.1, 2.4.6, and 3.3.2. These principles are generally ones that rely on a human understanding to evaluate (for instance, Principle 2.4.6 requires logic for determining if a label or heading actually describes the content it holds). Issues that require programmatic information or specific values to determine (such as contrasts or the ability to change content based on accessibility settings) were largely unmentioned within the user study.

5.3 Bility Results

I tested each application using the Bility testing framework. I downloaded the source code of each application through GitHub, inserted the Bility Android library into the application, created a test file (see Listing A.2 for an example test file), and started the Bility testing server. For each application, setup took no longer than 5 minutes.

Overall, I found that Bility was not only able to find static accessibility issues

similar to the ones found by the Google Accessibility Scanner, but it also had a more concise report of these issues. Bility also found additional accessibility issues related to the dynamic functioning of the application, such as keyboard navigation and changes in application context.

SoundRecorder

The Bility framework created a succinct report with both static and dynamic issues found within the application. Overall, Bility found the following number of unique issues:

- 4 violations of WCAG 1.1.1 Non-text Content (with a total of 66 instances found)
- 5 violations of WCAG 1.4.6 Contrast (Enhanced) (with a total of 38 instances found)
- 2 violations of WCAG 3.2.1 On Focus

The violations of WCAG 1.1.1 included the recording, play, and pause buttons found throughout the application, as well as the checkbox on the settings screen for enabling high quality recordings. The violations of WCAG 1.4.6 were contained within the action bar or toolbar at the top of the screen, as well as the tabs for navigating between the recording and saved recording screens. These violations also included the various focuses and selections of these tabs (not just the tabs with no interaction or input).

One useful aspect of Bility's representation of these issues is that it will not over-report issues that are dynamically repeated. For instance, see Figure 5-7, which reports that the Play button does not have a content description for the screen reader. Rather than reporting this issue for every single Play button on this screen, it is only reported once, recognizing that other instances of this issue on that page are likely caused by the same dynamically inflated View resource. To get an idea of how many

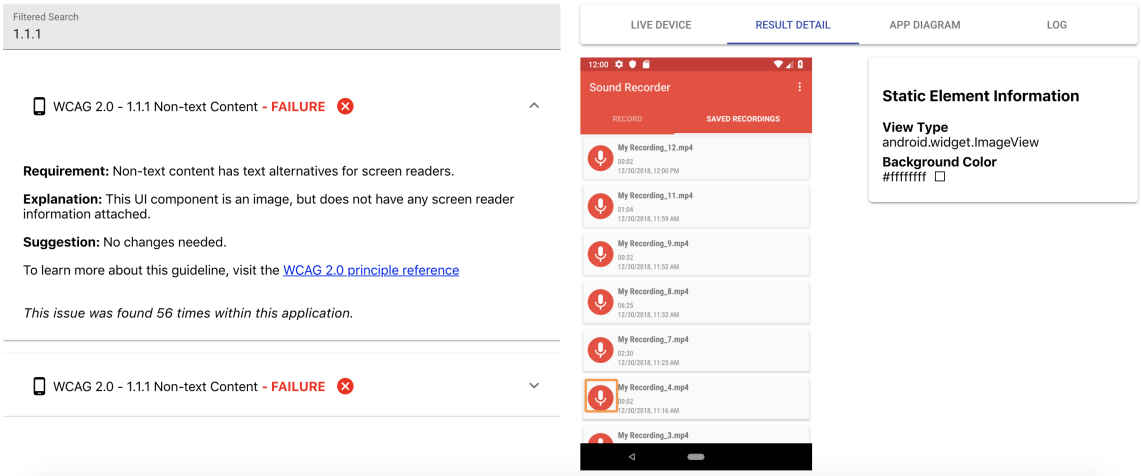

Figure 5-7: An instance of the Non-text Content violation for the Play button within the saved recordings screen.

similar issues exist, Bility also includes a count of similar instances at the bottom of the issues description.

Finally, Bility also found two dynamic accessibility issues, with violations of Principle 3.2.1 On Focus. Seen in Figure 5-8, Bility found that changing focus could cause the ViewPager holding the recording and saved recordings screen to switch between the two states. If a user is using a keyboard, changing focus may therefore cause the user to navigate to an entirely different screen, which may be jarring or confusing for the user. The developer could use this information to disable changes in focus between ViewPager screens, and force the user to use the tabs at the top of the screen to navigate when using a keyboard.

Travel-Mate

The Travel-Mate application was tested on Bility with the following issues reported:

- 11 violations of WCAG 1.1.1 Non-text Content (with a total of 68 instances found)
- 21 violations of WCAG 1.4.3 Contrast (with a total of 143 instances found)
- 33 violations of WCAG 1.4.6 Contrast (Enhanced) (with a total of 221 instances found)

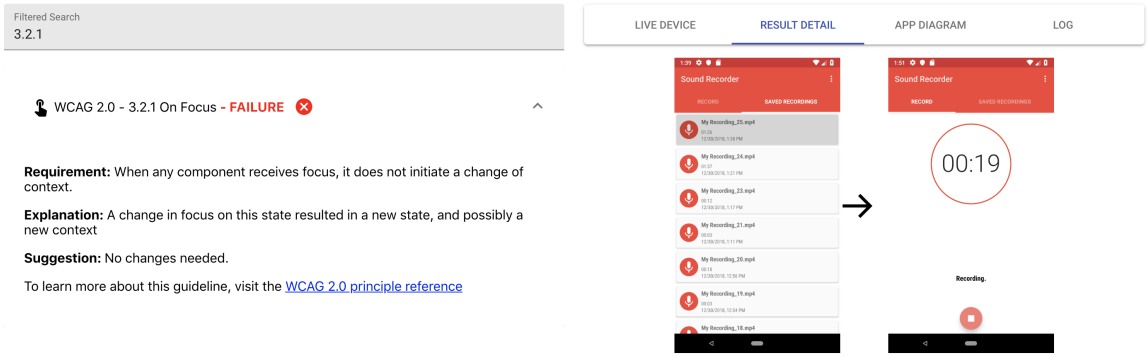

Figure 5-8: Using the keyboard on the recording screen may result in the user moving to the saved recording screens (and vice versa), which is a change in context. This can confuse some users, and Bility therefore marks this as an accessibility issue.

- 3 violations of WCAG 2.1.1 Keyboard

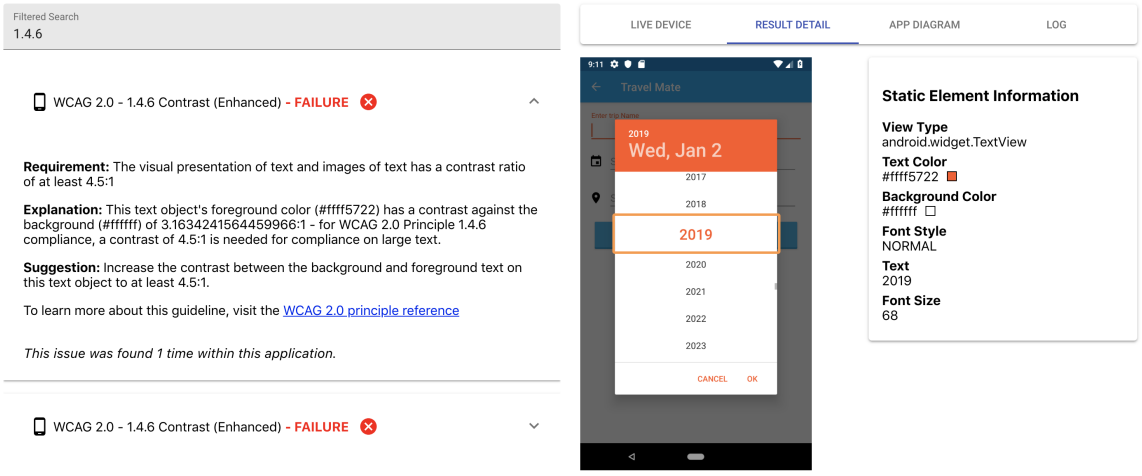

Figure 5-9: Bility proved to be great at manipulating single screens within an application to their full extent. Here we see Bility choosing a date within a date picker for a trip creation screen.

The Travel-Mate application has a very large state space; during manual testing, it was found that this application had at least 20 different screens. When manipulating these user interfaces, each screen can have multiple configurations and internal states, making it extremely difficult to test each state by hand. Bility proved to be useful in this situation, exploring many configurations of a single screen by trying multiples actions on a single screen (see Figure 5-9). However, this exploration behavior led to a trade-off, where not as many screens were explored when compared to manual

exploration. Bility (in its base configuration with no developer input) will attempt any action on any page that has not yet been taken, and as such has no knowledge of how to best explore the entire state space.

In fact, this best-effort attempt to explore the application is seen through the 3 violations of WCAG Principle 2.1.1. Rather than being real keyboard navigation issues, these three states were not reachable by Bility due to the fact that the Travel-Mate application would crash before Bility had a chance to full explore the state space. After multiple trial runs, the results seen here are those for the test run that had the most actions taken during the test (82 actions taken), representing the "most explored" run.

If Travel-Mate were able to catch these exceptions found during runtime, Bility would continue to run until a QUIT action was received. However, the testing server can still report and handle crashes within the application, as all results are processed and stored within real-time.

With regard to static issues, Bility found quite a few issues related to contrast and alternative content labels. However, Bility also misreported some of these contrast issues. In Figure 5-10, an instance was found where the correct background color for a button was not found. This can be improved with better procedures that analyze true pixel color compositions rather than just View properties.

Timber

The Timber application was also tested using Bility. Overall, Bility found the following count of issues:

- 15 violations of WCAG 1.1.1 Non-text Content (with a total of 53 instances found)
- 12 violations of WCAG 1.4.3 Contrast (Minimum) (with a total of 112 instances found)
- 22 violations of WCAG 1.4.6 Contrast (Enhanced) (with a total of 237 instances found)

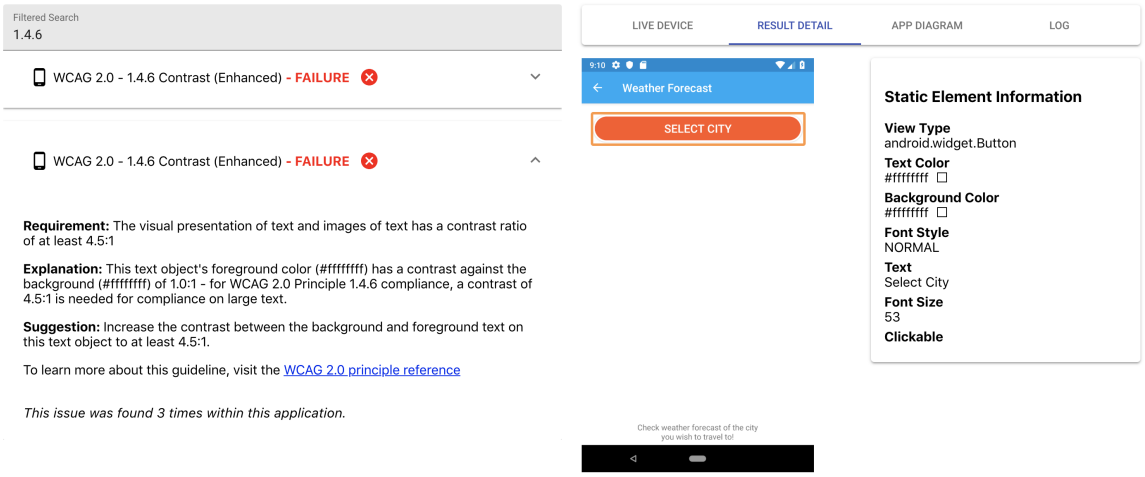

Figure 5-10: Bility would sometimes have trouble finding the correct background color for custom components and components that used background views separate from the foreground views. In this case, a button was given a custom background which did not communicate it's color as a View property.

- 4 violations of WCAG 2.1.1 Keyboard
- 7 violations of WCAG 3.2.1 On Focus

The persona was impressive in its ability to explore the state space, to a much greater extent compared to the manual testing done by human testers and during the Google Accessibility Scanner tests. After allowing the Bility persona to perform 395 actions, it was reported that 176 states with unexplored actions were still present (see Figure 5-12). This huge state space was caused by not only the wide variety of base screens available, but was also due to the fact that the application supports changing accent and background colors. Seen in Figure 5-11, Bility was actually able to find this color changing dialog and test various accents.

Timber also presented a few dynamic accessibility issues. For instance, Figure 5-13 shows the detection of keyboard navigation errors (WCAG 2.0 Principle 2.1.1) where some states are not reachable through keyboard use only. Using this information, I confirmed within the application that many screens do not respond to keyboard input, making it impossible to reach some screens using only the keyboard. A developer can then use this information to make sure that no screen presents a keyboard trap to the user.

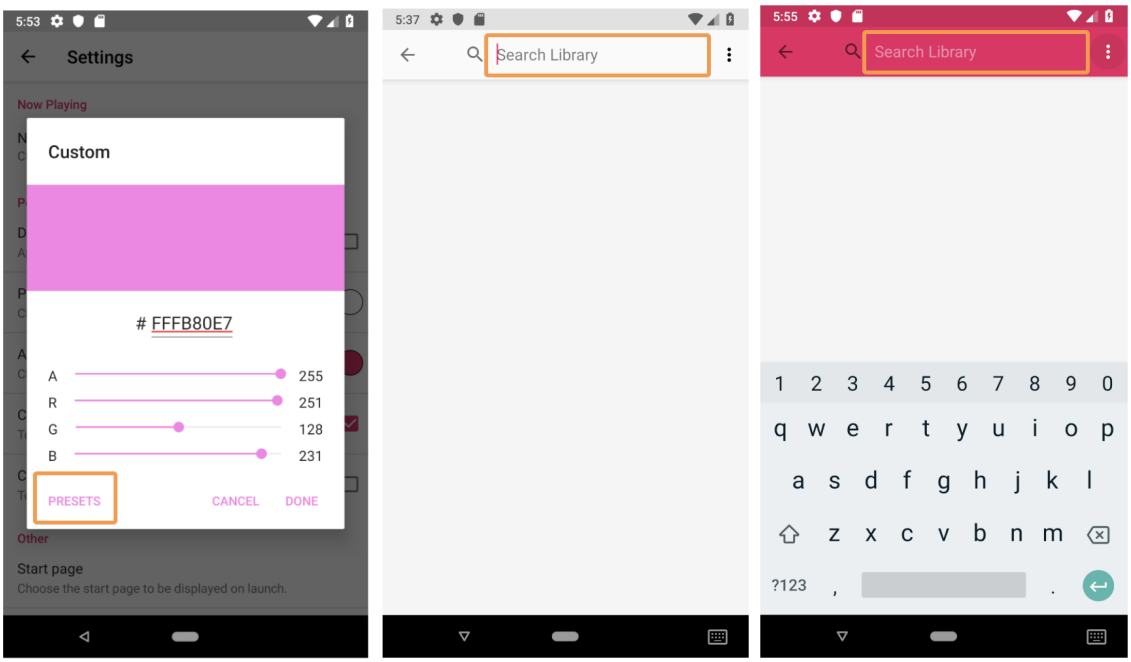

Figure 5-11: The first image shows Bility's encounter with the accent change dialog, while the second and third images display the different accents used throughout the application. In this instance, the search EditText is the same, but due to the change in background, any issue with the EditText is reported multiple times.

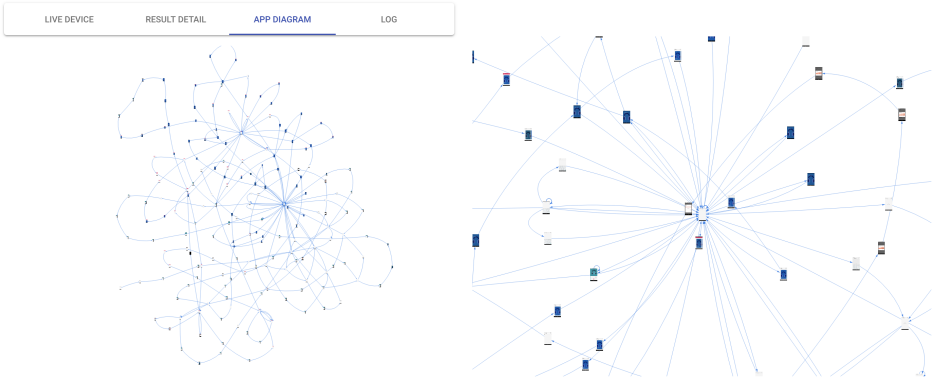

Figure 5-12: A birds-eye view of the state diagram for the Timber application (zoomed in on the left). Some states were explored heavily, while others still have many actions to test.

The static issues found by Bility include missing content descriptions on buttons and images throughout the user interface, as well as poor contrast on some text-based user interface elements. However, testing this application using Bility revealed a few limitations of Bility as well, specifically with respect to off-screen items and image backgrounds. For instance, Figure 5-14 shows the reporting of an issue with

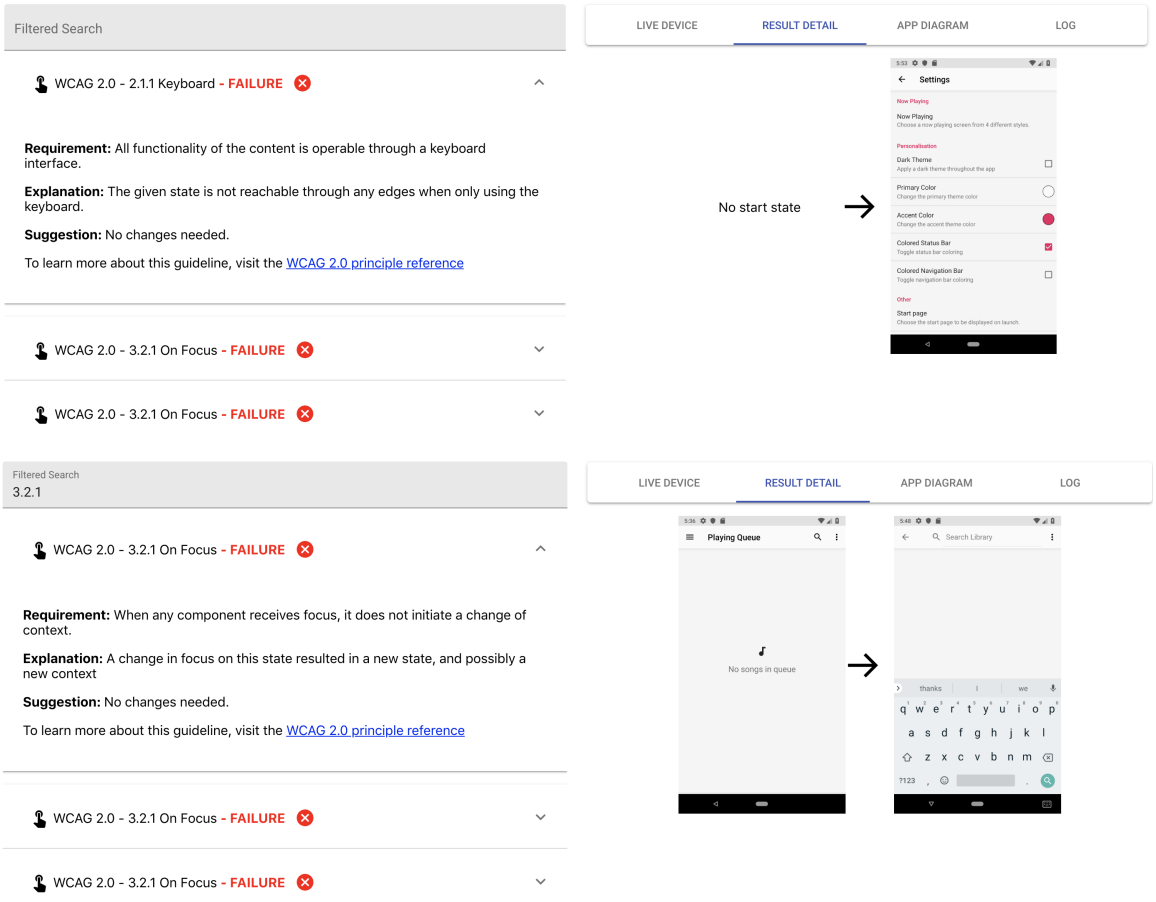

Figure 5-13: The top screenshot shows the reporting of one screen not reachable through keyboard use, while the bottom screenshot shows a change in context caused by a focus change (i.e. focusing on EditText causes a keyboard to appear).

contrast, since the TextView has a color of white, and no background color is found. Rather, the background is an ImageView, which is not scraped for colors and color regions. The default behavior of Bility is to assume a white base background, causing a white-on-white instance of poor contrast. However, an alternative solution to remove these issues is to disable contrast checks when no background color is found.

Overview

Using the Bility testing framework on the three applications proved as a useful case study to assess the capabilities of Bility. I found that Bility was able to achieve the following:

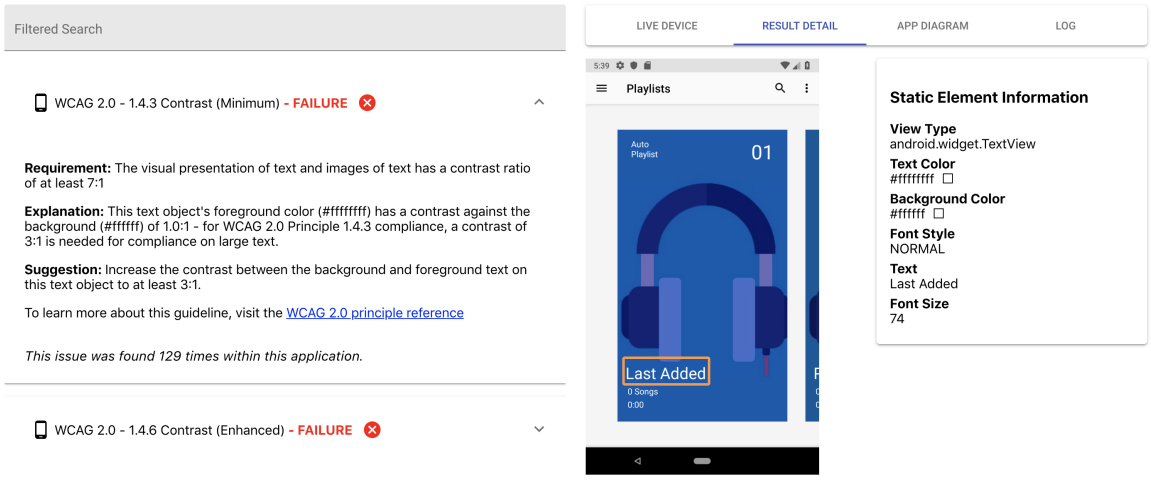

Figure 5-14: The highlighted TextView is over an ImageView, which does not report it's color composition. Therefore, white text is considered to be on transparent background, which gets interpreted to white.

- Detect static accessibility issues such as poor contrast, missing screen reader labels, and small touch target sizes.
- Detect dynamic accessibility issues such as keyboard traps and context changes when focusing.
- Automatically test an application for accessibility with minimal configuration from the developer.
- Report accessibility issues in an informative way without repeating issues within dynamically-generated views and layouts.

Within the design section, it was also noted that Bility is capable of testing repeated screen reader labels and changes in context when no user input is given. While this logic is implemented, the explored states of these applications did not reveal any issues related to these accessibility tests.

However, a few limitations and drawbacks of Bility were also revealed during this evaluation:

- Bility was not always able to automatically explore the application state space to the same extent as a human.

- Some issues related to contrast were incorrectly reported, most often related to the use of custom views and custom background.

A more detailed explanation and perspective on the strengths and weaknesses of Bility can be found within the Result Comparison section.

5.4 Analysis of Evaluation Results

Feature	Bility	Google Accessibility Scanner	Espresso	Lint*
Automated	✓		✓	✓
Static Code Checking				✓
Runtime Static Checking	✓	✓	✓	
Low Contrast Detection	✓	✓	✓	
Screen Reader Label Detection	✓	✓	✓	✓
Keyboard Trap Detection	✓			
Multi-platform Extendable	✓			
Testing on Debug APK	✓		✓	
Testing on Production APK		✓		
Optional Test Configuration	✓			
Live Issue Reporting	✓	✓		✓
Elimination of Repeated Issues	✓			
Viewing of Passing Instances	✓			

Table 5.1: Comparison between Bility other tools used for accessibility testing and evaluation.

A comparison of Bility, Google Accessibility Scanner, the Espresso testing library, and Android Studio's linting tools can be found in Table 5.1, which compares each software-based solution with respect to various criteria which are relevant during a developers testing process. Table 5.2 compares the results of each technique assessed within the Evaluation chapter with respect to the success criteria outlined in WCAG 2.0 and 2.1.

Overall, it is clear that Bility surpasses the other frameworks in both its ability to detect dynamic accessibility issues and its ability to detect accessibility issues with minimal configuration and processing efforts by the developer. Using navigation results of the application, Bility is able to analyze a state machine data structure to

Principle	Bility	Google Accessibility Scanner	Human
1.1.1 Non-text Content	✓	✓	
1.2.1 Audio-only and Video-only			✓
1.2.2 Captions (Prerecorded)			✓
1.2.3 Media Alternative (Prerecorded)			✓
1.4.3 Contrast (Minimum)	✓	✓	
1.4.6 Contrast (Enhanced)	✓	✓	
2.1.1 Keyboard	✓		
2.2.1 Timing Adjustable			✓
2.4.6 Headings and Labels			✓
2.5.5 Target Size	✓	✓	
3.2.1 On Focus	✓		
3.2.5 Change on Request	✓		
3.3.2 Labels or Instructions			✓

Table 5.2: Comparison between Bility, Google Accessibility Scanner, and humans with respect to WCAG principles and success criteria.

find accessibility issues related to the use of keyboards and unexpected changes of context. Other libraries are not currently able to do this, as only the current user interface being displayed is analyzed, with no tracking or knowledge of previous states. Furthermore, this dynamic exploration is done with little configuration or effort from the developer. For libraries such as Espresso, the developer must write navigation code to explore the application, while Google Accessibility Scanner requires the developer to manually click through the application. Bility, on the other hand, simply requires the developer to copy and paste one test file into their application, which will handle all automatic navigation.

Using the novel user interface hashing technique, Bility is also able to avoid the reporting of issues that have effectively been reported before. A common technique within Android development is to dynamically create a View from one resource or declaration; Bility is able to identify situations where an accessibility issue is possibly one which is already covered by a previous identification of that same issue within a similar view.

However, the Google Accessibility Scanner does perform with higher accuracy for some accessibility issue types. For instance, Bility will sometimes fail at accurately

detecting background colors when background colors are embedded within custom views and custom background drawables. The Google Accessibility Scanner avoids these issues by either analyzing drawables for the real background color or ignoring any issues if the custom view does not support accessibility information retrieval.

Chapter 6

Conclusion

6.1 Future Work

There are four categories of extensions and capabilities that can be added to Bility namely accessibility violation logic, automatic navigation, platform versatility, and user experience.

With respect to accessibility violations, the default implementation of Bility only detects violations pertaining to a subset of WCAG 2.0 and WCAG 2.1. However, further violations can be detected by adding logic to the accessibility detection routine on the test server. For instance, not only can more WCAG principles be added, but logic related to other guidelines and checklists such as AQuA and the compliance guide for Section 508 and the ADA can be added. [2, 28] Furthermore, recent improvements and availability of machine learning techniques may be used to increase the types of accessibility issues detected. For instance, using natural language processing, Bility may be able to address WCAG 2.0 Principle 3.1.5 Reading Level and 2.4.6 Headings and Labels, which describe rules for ensuring that reading level and the description of sections are appropriate for the application. [11, 30]

In order to navigate and explore the state space of an application, Bility uses a simple navigation procedure that prioritizes taking actions that have not been taken yet. Although this process works well for simple applications, those with many configurations and screens can cause termination of the test to take quite a long time.

With the data mining techniques and processes outlined in Rico and ERICA out of the Data Driven Design Group from the University of Illinois at Urbana-Champaign, Bility may be able to choose a user action given a user interface by running that screen through a trained model. [20, 21] This decision making based on real user interaction data may provide better coverage of the user interface. Additionally, more complex and specific rule-based logic may also be developed to handle interactions such as gesture drawing.

As stated previously, Bility separates accessibility violation detection logic and decision making logic from phone interaction logic. Various drivers for other platforms, such as iOS and the web, may be developed to work with Bility's core testing framework. For instance, websites may be navigated and converted into UDL using the Selenium framework. Supporting more platforms will provide useful accessibility insights to developers of many different types of software.

Finally, improvements can be made to the user experience of the Bility framework. The website used to view live reporting results and application diagrams is sufficient for beginning to tackle accessibility issues. However, developers may benefit from tutorials attached to issues, insights into the location of the issue within the source code, and further configurations for the behavior of the framework, such as indicating which types of issues should be reported.

6.2 Conclusion

Ensuring the accessibility of mobile applications has been difficult, due not only to the lack of breadth in the types of tests provided today, but also to the considerable amount of setup, time, and expertise needed to run the tests. The Bility testing framework addresses both of these issues; it is an accessibility testing framework that tests for both static and dynamic accessibility issues, and that can be easily extended to handle other accessibility requirements that rely on either static or dynamic data about the application. Furthermore, Bility provides a simple but informative interface for testing that requires close to zero effort by the developer. With this increase of

result information and decrease in developer effort, Bility has the potential to improve accessibility of many existing and future mobile applications.

Overall, in this thesis I made the following contributions:

1. An algorithm for automatically generating and reducing the state space of a user interface in the context of a specific analysis.

2. A process for testing dynamic accessibility issues such as keyboard navigation and spontaneous changes in application context through the construction of a state machine representing the application.

3. A platform-agnostic language used to describe user interfaces, including audio, physical, and motor elements.

4. An automated process to start, navigate, and record an Android application in an attempt to fully explore the application state space.

I have presented a system for automatically finding both static and dynamic accessibility issues within mobile applications, specifically on Android. An example implementation was used to test multiple open source Android applications for accessibility issues, with the results compared to similar tests by the Google Accessibility Scanner and human subjects.

Given the results of the evaluation, I have concluded that the Bility testing framework is an extremely powerful and easy-to-use tool. When compared to the Google Accessibility Scanner, Bility is able to detect similar static issues. However, Bility is also able to detect dynamic issues that the Google Accessibility Scanner is not able to.

Through this research, I found that tracking the dynamic state of an application over time in response to user input is incredibly useful for a wide variety of accessibility issues. Existing accessibility testing frameworks, which do not have this information, only assess the accessibility of a single screen at a time and fail to cover many scenarios where accessibility may be effected.

Furthermore, I introduced a new Universal Design Language to describe user interfaces in a more complete and perception-based way made testing for accessibility quite easy. Rather than interpreting a user interface solely through platform-specific code (which is quite limiting), using a universal description for user interfaces opens the possibility of extending Bility to many more platforms. For instance, while Bility is written for Android applications, it's separation of mobile application logic and accessibility testing logic allows this framework to open the doors for implementations in iOS, Chrome, and other platforms, which can connect to the primary Bility backend for accessibility testing.

The base implementation of navigation logic by using a state machine was also incredibly impactful for the Bility platform. A huge barrier to getting developers to test their applications for accessibility is simply the effort and time needed to setup accessibility tests for their applications. Rather than having users provide this logic themselves, Bility provides an automatic navigator, drastically reducing the time needed to test accessibility within applications.

However, the time needed to setup the test is not the only part of the testing process that has efficiency improvements. The frontend website used to display results condenses and succinctly explains accessibility issues, making the review of accessibility issues faster as well (when compared to compiling the results of a human or software test).

Finally, Bility presents many opportunities for improvements (discussed within the Future Work section). Not only can the framework be easily extended, but the modularization allows for performance improvements, such as the ability to distribute components of the test server. By open-sourcing the Bility framework, it is my personal hope that Bility will grow to be more accurate and efficient in its testing, which in turn will drive the development of applications that are more accessible to those who have disabilities.

Appendix A

Code

Listing A.1: Android Window Searching

```
1
2  /**
3   * Returns the root view of a given View
4   * @param view The View to get the root view of
5   * @return The root view of the View (as defined by android.R.id.content)
6   */
7  public static View getRootView(View view) {
8      return view.findViewById(android.R.id.content);
9  }
10
11 /**
12  * A function that uses Java reflection to find all
13  * instances of windows within the current activity,
14  * including the activity content view, dialogs, and
15  * context menus.
16  */
17 public static List<View> getViewRoots() {
18
19     List<ViewParent> viewRoots = new ArrayList<>();
20
21     try {
22         Object windowManager;
23         if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
24             windowManager = Class.forName("android.view.WindowManagerGlobal")
25                     .getMethod("getInstance").invoke(null);
26         } else {
27             Field f = Class.forName("android.view.WindowManagerImpl")
28                     .getDeclaredField("sWindowManager");
```

```
            f.setAccessible(true);
            windowManager = f.get(null);
        }

        Field rootsField = windowManager.getClass().getDeclaredField("mRoots");
        rootsField.setAccessible(true);

        Field stoppedField = Class.forName("android.view.ViewRootImpl")
                .getDeclaredField("mStopped");
        stoppedField.setAccessible(true);

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
            List<ViewParent> viewParents = (List<ViewParent>) rootsField
                .get(windowManager);
            // Filter out inactive view roots
            for (ViewParent viewParent : viewParents) {
                boolean stopped = (boolean) stoppedField.get(viewParent);
                if (!stopped) {
                    viewRoots.add(viewParent);
                }
            }
        } else {
            ViewParent[] viewParents = (ViewParent[]) rootsField.get(windowManager);
            // Filter out inactive view roots
            for (ViewParent viewParent : viewParents) {
                boolean stopped = (boolean) stoppedField.get(viewParent);
                if (!stopped) {
                    viewRoots.add(viewParent);
                }
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }

    List<View> rootViews = new ArrayList<>();
    for (ViewParent vp : viewRoots) {
        if (vp instanceof View) {
            rootViews.add(getRootView((View) vp));
        }
        if (vp.getClass().getCanonicalName().equals("android.view.ViewRootImpl")) {
            try {
                View view = (View) Class.forName("android.view.ViewRootImpl")
                        .getMethod("getView").invoke(vp);
                rootViews.add(getRootView(view));
```

```
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            } catch (NoSuchMethodException e) {
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }
    }

    // Remove any null views
    rootViews.removeAll(Collections.singleton(null));

    return rootViews;
}
```

Listing A.2: Bility Test File

```
/*
 * This file (BilityTest.java inside the androidTest
 * directory) is an example usage of Bility for
 * the SoundRecorder application.
 */

package com.danielkim.soundrecorder;

import android.support.test.runner.AndroidJUnit4;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.vontech.bilitytester.BilityTestConfig;
import org.vontech.bilitytester.BilityTester;

import static android.support.test.InstrumentationRegistry.getInstrumentation;

@RunWith(AndroidJUnit4.class)
public class BilityTest {

    private BilityTestConfig config;

    private final static String url = "http://10.0.2.2:8080";

    @Before
```

```
27      public void configureAppSpec() {
28
29          config = new BilityTestConfig();
30          config.setPackageName("com.danielkim.soundrecorder");
31          config.setMaxActions(400);
32
33      }
34
35      @Test
36      public void beginBilityTest() {
37          new BilityTester(url, getInstrumentation(), config)
38                  .startupApp()
39                  .loop();
40      }
41
42  }
```

Listing A.3: Alpha Color Blending

```
1  /**
2   * Given a list of colors in order of background -> foreground, calculates a
3   * resulting color. Assumes that all colors are stacked on top of a white
4   * background (i.e. 0xFFFFFFFF)
5   * References:
6   *  http://graphics.pixar.com:80/library/Compositing/paper.pdf
7   *  https://en.wikipedia.org/wiki/Alpha_compositing#Alpha_blending
8   */
9  fun blendColors(argbColors: List<Long>): Long {
10
11     var currentBaseColor = 0xFFFFFFFFL
12     argbColors.forEach {
13
14         // base = DST, next = SRC
15
16         val baseA = ((0xFF000000L and currentBaseColor) shr 24) / 255f
17         val baseR = ((0x00FF0000L and currentBaseColor) shr 16) / 255f
18         val baseG = ((0x0000FF00L and currentBaseColor) shr 8) / 255f
19         val baseB = ((0x000000FFL and currentBaseColor)) / 255f
20
21         val nextA = ((0xFF000000L and it) shr 24) / 255f
22         val nextR = ((0x00FF0000L and it) shr 16) / 255f
23         val nextG = ((0x0000FF00L and it) shr 8) / 255f
24         val nextB = ((0x000000FFL and it)) / 255f
25
26         val outA = nextA + baseA * (1 - nextA)
27
```

```
28          currentBaseColor = 0
29          if (outA != 0f) {
30              val outR = round(((nextR * nextA + baseR * baseA * (1 - nextA)) / outA)
31                  * 255).toInt()
32              val outG = round(((nextG * nextA + baseG * baseA * (1 - nextA)) / outA)
33                  * 255).toInt()
34              val outB = round(((nextB * nextA + baseB * baseA * (1 - nextA)) / outA)
35                  * 255).toInt()
36              val outAInt = round(outA * 255).toInt()
37              currentBaseColor = outB.toLong() + (outG shl 8).toLong() +
38                  (outR shl 16).toLong() + (outAInt shl 24).toLong()
39          }
40
41      }
42
43      return currentBaseColor
44
45  }
```

Bibliography

[1] Accessibility checking. https://developer.android.com/training/testing/espresso/accessibility-checking. Accessed Dec 16, 2018. Android Developers (Espresso).

[2] Accessibility Testing Criteria for Android Applications. https://www.appqualityalliance.org/Accessibility_Testing_Criteria. AQuA - App Quality Alliance.

[3] AccessibilityChecks - Robolectric. http://robolectric.org/javadoc/latest/org/robolectric/annotation/AccessibilityChecks.html. Accessed Dec 16, 2018. Robolectric Javadocs.

[4] AccessibilityDetector.java. https://android.googlesource.com/platform/tools/base/+/studio-3.0/lint/libs/lint-checks/src/main/java/com/android/tools/lint/checks/AccessibilityDetector.java. Accessed Dec. 16, 2018. Google Git.

[5] AccessibilityNodeInfo. https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo. Accessed Dec 16, 2018. Android Developers.

[6] Alpha compositing - Alpha blending (Wikipedia). https://en.wikipedia.org/wiki/Alpha_compositing#Alpha_blending. Accessed Dec 27, 2018.

[7] Android Accessibility Help. https://support.google.com/accessibility/android/?hl=en#topic=6007234. Google Support.

[8] Get started with Accessibility Scanner. https://support.google.com/accessibility/android/answer/6376570?hl=en. Android Accessibility Help.

[9] Keyboard: Understanding SC 2.1.1. https://www.w3.org/TR/UNDERSTANDING-WCAG20/keyboard-operation-keyboard-operable.html. WCAG 2.0 - W3C.

[10] Project Eyes-Free. https://github.com/rmtheis/eyes-free. Accessed Dec 16, 2018. Github Repository.

[11] Web Content Accessibility Guidelines 2.0, W3C World Wide Web Consortium Recommendation 11 December 2008, Success Criteria 1.1.1, 1.4.3, 1.4.6, 2.1.1, 2.4.6, 2.5.5, 3.1.5, 3.2.1, 3.2.5. https://www.w3.org/TR/2008/REC-WCAG20-20081211/. Accessed Dec 29, 2018.

[12] Web content accessibility guidelines (wcag) 2.0. W3C Recommendation 11 December 2008.

[13] A. Vontell, W. Caruso, B. Park, L. Kagal. Analysis of the Native Mobile Accessibility Landscape. https://github.com/vontell/Bility/blob/master/papers/AMAWhitePaper.pdf. Unpublished Draft.

[14] Essential Accessibility. Do i need an ada compliance consultant? https://www.essentialaccessibility.com/blog/ada-compliance-consultant/, 2017.

[15] alex-p (StackOverflow). Is there a way to get current activity's layout and views via adb? https://stackoverflow.com/questions/26586685/is-there-a-way-to-get-current-activitys-layout-and-views-via-adb. Accessed Dec 27, 2018.

[16] Android Developers. UiDevice. https://developer.android.com/reference/android/support/test/uiautomator/UiDevice.

[17] Jefte Macedo Walter Correia Marcelo Penha Fabio Silva Andre Santos Marcelo Anjos Clauirton Siebra, Tatiana Gouveia and Fabiana Florentin. Usability requirements for mobile accessibility: a study on the vision impairment. 14th International Conference on Mobile and Ubiquitous Multimedia (MUM '15), pages 384–389. The OX Association for Computing Machinery, ACM, New York, NY, USA, 2015. DOI: http://dx.doi.org/10.1145/2836041.2841213.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*, page 595–601. The MIT Press, 3rd edition, 2009.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*, page 558–565. The MIT Press, 3rd edition, 2009.

[20] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 845–854, New York, NY, USA, 2017. ACM.

[21] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. Erica: Interaction mining mobile apps. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 767–776, New York, NY, USA, 2016. ACM.

[22] Daniel Kim (dkim0419). Soundrecorder. https://github.com/dkim0419/SoundRecorder, 2017.

[23] JFrog. Artifactory - Universal Repository Manager. https://jfrog.com/artifactory/. Accessed Dec 27, 2018.

[24] Jing Li. thyrlian/AndroidSDK. https://github.com/thyrlian/AndroidSDK. Accessed Dec 26, 2018. Github Repository.

[25] M. Linares Vasquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. How do Developers Test Android Applications? *arXiv e-prints*, January 2018.

[26] P. Maziewski, P. Suchomski, B. Kostek, and A. Czyzewski. An intuitive graphical user interface for the parkinson's disease patients. In *2009 4th International IEEE/EMBS Conference on Neural Engineering*, pages 14–17, April 2009.

[27] Naman Dwivedi (naman14). Timber. https://github.com/naman14/Timber, 2018.

[28] Office of the Assistant Attorney General. Software Accessibility Checklist. https://www.justice.gov/crt/software-accessibility-checklist, August 2015.

[29] openstf. openstf/minicap. https://github.com/openstf/minicap. Accessed Dec 26, 2018. Github Repository.

[30] Sarah E. Petersen. *Natural Language Processing Tools for Reading Level Assessment and Text Simplification for Bilingual Education*. PhD thesis, Seattle, WA, USA, 2007. AAI3275902.

[31] Thomas Porter and Tom Duff. Compositing digital images. *SIGGRAPH Comput. Graph.*, 18(3):253–259, January 1984.

[32] project-travel mate. Travel-mate. https://github.com/project-travel-mate/Travel-Mate, 2018.

[33] W3C. WCAG 2.0 - Contrast Ratio (Note). https://www.w3.org/TR/WCAG21/#dfn-contrast-ratio. Accessed Dec 29, 2018.

[34] WCAG Working Group. Relative luminance. https://www.w3.org/WAI/GL/wiki/Relative_luminance. Accessed Dec 27, 2018.