Engineering Accessible Software

Thesis proposal

Arun Krishna Vajjala
Department of Computer Science
George Mason University
Fairfax, VA 22030
akrishn@gmu.edu

Committee
Kevin Moran, University of Central Florida (Chair)
Brittany Johnson-Matthews, George Mason University
Andrian Marcus, George Mason University
Thomas LaToza, George Mason University
Vivian Motti, George Mason University

Abstract

Continuous integration (**CI**) is a principle in current software development focusing on enhancing software quality by detecting bugs earlier. A primary task in **CI** is executing the test suite to ensure software correctness, particularly after code modifications. The expected outcome of these tests should be deterministic. However, certain tests may exhibit non-deterministic behavior even when run on an unchanged codebase. Such non-deterministic tests, known as *Flaky Tests*, can pass or fail for the same version of the software. This inconsistent outcome reduces reliability and can complicate the **CI** process.

The traditional way to detect flaky tests is to rerun them multiple times and if a test produces both passing and failing outcomes without any changes in the codebase, it is confirmed flaky [**?**]. Rerunning tests can be costly, leading to unacceptable overhead expenses, and it may not detect flakiness in tests that behave inconsistently in different running environments. Hence, researchers have been exploring alternative methods to detect flaky tests effectively rather than rerunning them.

To address the issue of flaky tests, I began by conducting a rerun experiment, aiming to understand the limitations of this technique beyond its obvious costs. From the data gathered during this experiment, I detect a set of flaky tests. Using this dataset, I developed a machine learning classifier, named FlakeFlagger, designed to predict if a test is flaky or not. The goal was to use FlakeFlagger without the need for reruns by finding the similar symptoms a test shares with other identified flaky tests. I evaluated the performance of FlakeFlagger against state-of-the-art tools to gain a better understanding of its effectiveness.

Flaky tests could exist in their test suites even after being confirmed as flaky for various reasons (e.g. helpful to detect defects). Identifying which failure from these tests are flaky or not is another main challenges. Hence, I proposed machine learning and non machine learning approaches to identify which failure is flaky or not based on the tests failure logs. Using the failure logs for each test, I am currently leveraging them to identify the root causes of test flakiness. This helps understanding the flaky failures and determining how to address them.

Contents

List of Figures

List of Tables

1 Introduction

Software has become an integral part of everyone's lives and its impact continues to grow. Software takes many forms such as web applications, smartphone applications, and desktop or operating system applications, etc [131]. Our everyday lives depend on the use of critical software applications which allow us to bank, invest, get news, and communicate with others. Touch screen devices such as smartphones and tablets provide a quick and easy means of access to important information and functions within our daily lives.

Software accessibility has become more important as more users are dependent on smartphones and computers. People of different abilities have found it difficult to use software the way it is currently developed and designed [110]. According to the world health organization (WHO), 15% of people have some disability [11], making software accessibility more important to ensure all users are able to use applications as intended.Though software engineers and companies are ethically motivated to create more accessible software, the United States Government is also making efforts to require public websites and services to be accessible [13]. The government in conjunction with the American with Disabilities Act (ADA) introduced legislation which "prohibits discrimination on the basis of disability in the activities of public accommodations" [13, 31]. This has lead to a 180% increase in more accessible software as of 2018 [12]. This change in policy increases a need for more accessible software and tools that will help developers make their applications more accessible.

Software engineering research is constantly innovating how software is made and tested. Software testing and developer tools are constantly evolving and are making their way into the accessibility space. Software testing has been around for decades, but has recently grown into a more complex field with the use of computer vision and machine learning techniques to automatically generate and run tests. This exciting new way of testing could provide an ample amount of ways to test for accessibility guideline violations without developers needing to explicitly check for violations on their own. An extremely important part of software engineering research is the constant need for new, robust developer tools to assist developers in the process of designing and creating software. These tools allow developers to have quick access to information and functions that make the development process more efficient. With the evolution of machine learning techniques, research is able to tackle issues that have needed a more complex understanding of the screen and User Interface (UI) in order to detect accessibility issues accurately.

The primary techniques that enable modern approaches for computational understanding of UIs are deep learning techniques for both computer vision and natural language. That is, researchers have begun to adapt models trained to understand open-domain image and text to the task of comprehending user interfaces.

The crux of the above challenges is adapting *generalized* models, which have been shown to effectively learn patterns across a varied range of data domains, to the specific domain of user interfaces. As such, the methods mentioned often *attempt* to capture rich contextual information conveyed through the visual elements of the screen. UI's are designed as graphical interfaces with specific layouts and visual properties that capture important semantic information about the affordances of the UI. That is, the structural, visual, and lexical properties of elements on the screen provide contextual information about each screen's function. Another underlying challenge in learning patterns from user interfaces is the *variability* in designs that convey similar semantic meanings. For example, even a screen as simple as a login screen, which is typically comprised of

two text fields and a button, can be instantiated through a wide variety of different visual designs and lexicon. While current UI embedding techniques, such as VUT and UIBert aim to address these challenges through embedding UI hierarchy information, this information is often flattened into representations that make it difficult to deal with the large variety of semantically similar, yet characteristically different patterns that are found across UI designs.

In this thesis, I aim develop tools for developers to assist them in making their own software more accessible and tackle future problems which require a deeper semantic understanding of screen. This thesis revolves around using newer techniques to tackle previously unexplored problems. I propose a tool that uses computer vision techniques to identify motor-impairment accessibility issues in Android mobile apps and demonstrate its efficacy by outperforming state of the art baselines while maintaining high accuracy. Additionally, I propose a computer vision based screen understanding tool to aid in screen recognition and retrieval tasks in search tasks at a high percentage.

The thesis proposal is organized as follows: Section 2 provides an introduction of works towards the problem of test flakiness. The main contributions of the thesis are discussed in Section 3. Section **??** provides a summary of the findings related to the detection of flaky tests. Section **??** emphasizes current research on how to identify flaky *failures* and explores techniques for their detection. The main key points for my future work, which form the remainder of my PhD, are discussed in Section **??**. Lastly, the current plan for the remaining phase of my PhD is outlined in Section 8.

2 Background

This thesis introduces two ideas within software engineering which intersect at the need for developer focused tools for software accessibility and the comprehension of the UI screen to facilitate advanced accessibility improvement techniques. To gain a deeper understanding of this intersected research area, it is important to understand the motivation behind each research area and how they benefit each other.

2.1 Software Accessibility

Research in software accessibility relies on the knowledge of various demographics of the disabled community. Most research is focused on solving a problem within a single population of disabled people, for example, a study could test to see if Android applications are accessible to Motor-impaired users [121], while another could develop a tool to label UI elements in order to help screen readers read out UI elements to visually impaired people [48]. This section will go through how people with different disabilities use touch screen devices. Table 2 lists a series of accessibility guidelines mentioned in previous work as well as the affected user demographic.

Visually Impaired Users

Visually impaired users use screen readers about 90.5% of the time [48]. Low vision (LV) users, or users who can see, but not to the extent that legally blind people can, do not always depend on screen readers. They depend on being able to see contrast in UI design and fonts, making it so that elements and text on the screen are clearly visible and large enough for them

Table 1: Accessibility guidelines extracted from our systematic literature review of accessibility guidelines – includes recent research and Google's [17] and Apple's [1] accessibility guidelines. (LV = low vision users, DHH = deaf and hard of hearing users)

Accessibility Guideline	Primary Affected User Demographic	Guideline Source	Previous Implementation
Visual Touch Target Size	Motor, LV	[71, 109]	
Touch Target Size	Motor, LV	[1, 17, 20, 22, 107, 45, 31, 27, 66, 71]	X
Persistent Element Location	Motor	[1, 20, 22, 17]	
Clickable Span	Motor, LV	[31]	X
Duplicate Clickable Bounds	Motor	[31]	X
Editable Item Descriptions	LV	[31, 59]	X
Expanding Section Closure	Motor, LV	[1, 17, 20, 22]	
Non-Native Elements	Motor, LV	[17, 45]	X
Motion Activation	Motor, LV	[1, 17, 20]	
Labeled Elements	Visual, DHH	[31, 60, 76, 59]	X
Screen Captioning	LV, DHH	[1, 17, 20, 22, 12, 16, 119, 76, 111, 66]	X
Keyboard Navigation	Motor, LV	[12, 16, 60, 76, 52]	X
Traversal Order	Motor	[16, 31, 60]	X
Adjacent Visual Icon Distance	Motor	[1, 17, 22, 139, 27, 107]	X
Proper Information Organization	Motor, LV	[45]	X
Facial Recognition	Motor	[45, 32]	X
Single Tap Navigation	Motor	[1, 17, 20, 22, 60, 94]	
Poor form design/instructions	Motor, LV	[12, 16]	

to read [14]. LV users do not only depend on being able to see and navigate the screen, various keystrokes and actions are required in order to interact with touch screen devices.

An example of this is shown in Figure 1. LV users have difficulty seeing the screen and resort to guessing where their taps land. The image shows clustered taps in an attempt to type out sentences on a touch device. It is clear that LV users are not properly able to tap where they intend to. Speech to text [19, 18] or braille inputs are a way for LV users to effectively input information into the device [33]. Keystrokes are identified by gesture recognizers built within touch screen devices [112]. These recognizers are built with the able-bodied user in mind, so companies like Apple and Google allow users to modify the sensitivity and duration of the touch inputs [112, 19, 18]. These accessibility measures help LV users use their devices more efficiently.

Deaf and Hard of Hearing Users

Deaf and Hard of Hearing (DHH) users, unlike LV users, are comfortably able to physically use their touchscreen devices. DHH users need live captioning or American-Sign-Language (ASL) interpretations of any audio outputted by the device [40]. There are currently tools that provide live captioning [23] and research that is being done in converting audio to ASL depictions for users [43]. DHH users also rely on text-to-speech or limited verbal communication to provide audio inputs into their devices [61].

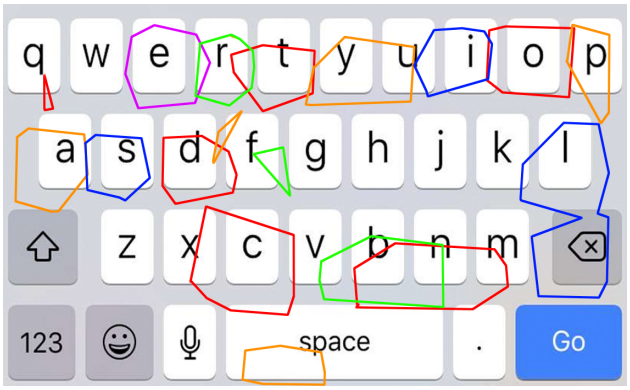

Figure 1: Polygons representing clusters of hit points for LV users [105]

Motor-impaired Users

Figure 2: Switch Interface

Motor-impaired users use devices in two main ways: Switch input and touch input [143]. Switch based input uses an external hardware input device. Switches are common in users with limited to no motor control who still retain their cognitive functions [143]. Current touch devices offer scanning methods that scan a curser across the screen and highlights them as they pass [19, 18]. The users then can click the switch and select the application that is highlighted and continue using the device. An example of this switch interface is shows in Figure 2. The most commonly used switch is a single input switch, which is then connected to an adapter to create signals for the device, and then to the device, in this case, an iPad. The iPad then can scan and highlight rows and columns for the user to tap on the switch and select. This gets very tedious as average words per minute typed using tradition scanning based techniques is 3, which is significantly lower than the average normal users, which is, 39 [87]. The other form of input, touch input, is the same as a normal user, but motor-impaired users generally have a tremor or lack of speed in gestures made on the screen. Slightly motor-impaired users also find themselves using voice commands to navigate applications and send messages since keyboards can be frustrating to use with smaller keys [143]. Work has been done to improve gesture recognition. Google and Apple offer sensitivity and touch settings that allow users to better physically interact with their devices [112, 19, 18].

Neurodivergent and Cognitively Impaired Users

This demographic of users is primarily classified as individuals with cerebral differences such as Attention-deficit/hyperactivity disorder (ADHD), Autism Spectrum Disorder (ASD), dyslexia, and memory loss. It is estimated that 7% of the population identifies as neurodivergent-encompassing

cognitive and learning disabilities [115]. These users, unlike the users presented previously, are severely underrepresented in design of the web. The Web Content Accessibility Guidelines (WCAG) [9] requires accessibility measures for LV, DHH, and motor-impaired users, but does not require the implementation of neurodivergent supporting guidelines. Software and tools that neurodivergent users rely on tend to be low-sensory to avoid sensory overload. Users who are distracted easily or have highly receptive cognitive functionality tend to need software where there is a clear focal point to each functionality in the application without the need of unnecessary animations and sounds [120]. The current set of applications provides many complex avenues of accessing information, but can be challenging to neurodivergent users. An application like Twitter, for example, has an abundance of scrolling animation and task bars that can be used to navigate the app. The animations, videos playing, and options of navigation can overload users since there is no focal point to the screen and what the user should be looking at.

2.2 Developer Tools for Accessibility

The current landscape of developer tools is limited. Studies have shown that current developers do not utilize tools or ignore them because they introduce warnings, some of which are completely wrong [121]. In this survey we found very few developer-specific tools, but found many studies that looked at software guidelines for developers. Guidelines are set for developers on all platforms to make their websites and applications more accessible [19, 18, 36, 132]. GUI guidelines require developers to abide by certain standards and practices in order to help disabled users navigate and use the application as intended. Guidelines specify certain requirements for elements such as size, placement, and color contrast to help users see tap and see them easier [19, 18].

2.3 UI Understanding

Much of the research in accessible GUI analysis has been done in UI enhancement and UI comprehension for use in accessibility studies, hence it is important to understand the current state of UI comprehension techniques. Accessibility based UI augmentation and comprehension is only as good as the current techniques in the field.

Wu et al. [137] worked on a screen parser that is able to predict relationships within elements on the screen. They did this by training a Faster-RCNN using thousands of both IOS and Android screenshots. Then they determined node correspondence and extracted hierarchical correspondence for each of the elements. Then they grouped the elements together into groups that were under a certain category. So interactive elements were one group while background and images would have been another group [137]. This work was done to better understand the relation between elements on the screen, but can also be used to label and identify screen elements and traverse them in a better order for users with disabilities. This approach was expensive and the UIElement detector they used was observed to slow other processes down.

Most accessibility software tools have been related to testing where researchers tend to find accessibility guideline violations. Moran et al. [98] worked to create an automated GUI checker that checks to see if GUIs were made to their intended design [98]. They then used a GUI comprehension technique and a set of design violations to work on a detector that would detect design violations. It gave promising results that correctly detected design violations in Android apps.

This visual GUI testing (VGT) is a great way to start testing for more accessible GUIs. This testing method, however, does not allow GUI boxes to overlap and requires a straightforward GUI. This can hinder GUI mockups and novice developers from being able to test for violations.

Like the project by Moran et al [98], UIBert by Google [34] works to identify and comprehend screen elements. It works by taking in UIs and parsing it to identify all of the elements. It then uses that information to predict which type of app it is. It classifies the icons and then uses those generalized icon predictions to make an app prediction. This is a way for UI comprehension to be useful for screen readers or voice assistants to be able to access a certain type of app, e.g. games or music [34].

It is really important to understand GUIs when looking to help developers make more accessible software because most techniques from GUI comprehension are used to label and modify GUIs to make applications more accessible, so by looking at the current research in this field, we can see how GUI accessibility is dependent on advancements in GUI comprehension.

3 Thesis

3.1 Problem Statement

Advancements in software accessibility present new and exciting challenges to create tools for developers that transcend past the static usage of the code and metadata in applications. Existing work has made a step in the direction of making software more accessible, but has yet to leverage new, state-of-the art techniques to augment and detect software accessibility issues in applications. Most importantly, previous methods have relied on previously existing accessibility frameworks on the devices. Given the limitations of current research and the critical role that accessibility in mobile devices plays in an ever-changing and growing population of disabled users, it is important to identify comprehensive and effective solutions to address issues in software accessibility.

3.2 Thesis Statement

The thesis statement is that new machine learning and computer vision-based approaches to semantic screen understanding can address the problem of software accessibility by way of detecting, locating, and augmenting UI screens. To investigate this, I perform a search of existing accessibility focused developer tools and tailor my research direction to cater to limitations in the current research. By understanding the needs in the current state of research, i discuss and propose a series of solutions which use computer vision and UI comprehension techniques. Upon evaluation, I discuss each proposed solution alongside their research questions as shown below.

3.2.1 Detecting Motor-Impairment Accessibility Guideline Violations

I started my research in detecting flaky tests by conducting a study of the current state of tools that cater to software accessibility. Following this search, I deduce that the motor-impaired community is not represented as much as low vision or deaf and hard of hearing users. Table 2 shows the list of of motor impairment guidelines and which have been implemented in prior work. I notice that motor-impairment guideline detectors have not been implemented in past works. This leads us to believe that prior work has not attempted this due to the lack of sufficient data and computer-vision based techniques to facilitate the detection of guideline violations. I use this analysis to propose MotorEase. MotorEase is a novel motor-impairment guideline violation detector that uses a computer-vision approach to detect four different accessibility guideline violations within Android applications. The tool uses outputs from existing Android app testing tools to make it seamless for developers to test their applications for motor-impairment accessibility issues. This tool is evaluated on a series of manually annotated screenshots. MotorEase demonstrates a high accuracy in detecting motor-impairment guideline issues in mobile applications while leveraging the visual semantics of the screen. To address the challenge of detecting these guideline violations, we analyze and evaluate the results of MotorEase with these research questions:

RQ$_1$ *How accurate is the Expanding Section detector?*

RQ$_2$ *How accurate is the Visual Touch Target detector?*

RQ$_3$ *How accurate is the Persisting Element detector?*

RQ$_4$ *How accurate is the UI Element Distance detector?*

RQ$_5$ *Does MotorEase identify a limited number of false positive and negative violations?*

The related findings of the detection of motor-impairment guideline violations are detailed in Section 4. The answers to all these questions are primarily summarized from the paper "MotorEase: Automated Detection of Motor Impairment Accessibility Issues in Mobile App UIs" [73]. This wok was submitted ad accepted at the International Conference on Software Engineering (ICSE 2024).

3.2.2 Structurally motivated UI Embedding

With the increased need for screen understanding as shown in Sections **??**, I aim to propose a UI embedding for app screens. Prior screen embeddings treat embeddings as images and not as UIs with rich layout and structural information. Some methods of screen embeddings consider the text on the screen and create text embedding for each screen. However, these methods provide a limited semantic understanding of the screen. Some approaches attempt to combine the text and image features of a UI to create a UI embedding, however, were these techniques fail to leverage the layout of the components of the screen. We take an intuitive approach of identifying structure in a UI and create FRAME, an embedding for UI screens which has a structure bias. This approach creates an embedding that can assist in screen retrieval and similarity tasks. I evaluate the embedding with these four different research questions:

RQ$_1$: *How does FRAME perform against other baselines in screen retrieval tasks?*

RQ$_2$: *How does FRAME perform on various types of screens against the best baseline?*

RQ$_3$: *How important is the structural embedding propagation between graphs in order to enhance the embedding?*

RQ$_4$: *How well does FRAME leverage its ability to abstract the screen and disregard styling?*

The research and its findings are detailed in Section 5. This section also introduces the proposed approaches for failure de-duplication and the methodology for addressing the research questions is discussed. This wok was submitted and is currently under review at the ACM Symposium on User Interface Software and Technology(UIST 2024).

3.2.3 UI Search

We combine the intuition in the previous two solutions to create a comprehensive solution for UI design. Many developers begin the development process with a wireframe design of the UI. This UI design is often just an image and is easy to manipulate if needed. I am currently developing and testing SearchAccess, a UI search engine which allows developers to find similar screens to their mockup that are more accessible. However, SeachAccess not only a search engine, it is a visual environment to view accessibility issues in any given UI. This is a powerful tool that will allow developers to identify accessibility issues in their mobile app mockups and make changes inspired by similar, more accessible, UIs via search. Many times, developers tend to dismiss

accessibility changes because the app is already developed and making changes is time consuming. The existence of a computer vision based approach to identify accessibility issues with only the mockup will encourage developers to make accessible decisions before code gets written for the application. To test SearchAccess, we present some initial evaluation for the tool and intended future work. We present a list of research questions to evaluate the efficacy of SearchAccess:

RQ$_1$: *How does SearchAccess perform in screen retrieval tasks?*

RQ$_2$: *How accurate are the detectors in SeachAccess?*

RQ$_3$: *Are developers able to identify accessibility issues within their UI designs?*

RQ$_4$: *Do developers benefit from UI search when looking to make their apps more accesible?*

The discussion of these research questions, along with detailed insights on how I approach them, can be found in Section 6.

3.2.4 Literature Review on Developer Tools

Given the research efforts above, it is important to understand the impact being made in the field of machine-learning based developer tools to make applications more accessible. I am currently assembling a Systemic Literature Review (SLR) which analyzes the current state of developer tools that facilitate the detection, testing, and augmentation of applications for accessibility issues. This SLR will provide a foundational understanding of the importance this research can provide. It will look at the available tools, the users they target, input/output formats, and experimental methodologies. I aim to present a comprehensive review of the research area at the intersection of machine learning, software engineering, and accessibility to establish the foundation for future work within this research area. This work will attempt to present the answers to these research questions:

RQ$_1$: *Is the research targeted at automating developer activities, enhancing existing software, or creating guidelines for developers?*

RQ$_2$: *What software domains does research on accessibility typically target?*

RQ$_3$: *Which populations of users with accessibility needs has software engineering targeted?*

RQ$_4$: *What type of data do studies use and how can the quality of data and its collection suggest an impact on the research in the field?*

RQ$_5$: *What are the primary means of evaluation for research that targets users with accessibility needs.?*

The discussion of these research questions, along with detailed insights on how I approach them, can be found in Section 7.

4 Detecting Motor-Impairment Accessibility Guideline Violations

The everyday lives of end-users depend on the use of software applications that support critical tasks such as banking, reading news, and communicating with others. Due to the central role of software in modern society, developers have an obligation to ensure that people of *all* abilities and backgrounds are able to use applications to carry out daily tasks. However, this ideal is still very much a goal that engineers must collectively work toward, as past work has illustrated the prevalence of accessibility issues in mobile app software ecosystems [31, 133, 51]. Furthermore, the need for accessible software now transcends a moral pursuit, as government agencies worldwide have begun to advocate for more accessible software by introducing legislation which, "prohibits discrimination on the basis of disability in the activities of public accommodations," [13]. Beyond providing equitable access to software for users with a variety of backgrounds, accessibility features often improve user experience more broadly, as many accessibility guidelines are designed following the general principals of universal design [8], in that the adherence to such guidelines is more likely to lead to an improved user experience for *all* users [125].

Current research at the intersection of developer tools and software accessibility has generally been disproportionally focused on users with certain disabilities, such as visual impairments, i.e., low vision (LV), and hearing impairments, i.e., deaf and hard of hearing (DHH) [110, 145, 62, 133, 51]. The visual nature of software user interfaces (UIs), and large populations of users with visual impairments have made this a natural and important focus area. This focus, however, must expand to study and create tools that aid developers in considering and implementing accessible features that support users with a wider range of disabilities. One understudied demographic, and the focus of this paper, is that of *motor-impaired users*. The current landscape of research on developer tools that aim to support software accessibility for motor-impaired users is somewhat limited, due in part to the difficulty in supporting a wide spectrum of motor-impairment conditions (i.e., ranging from hand tremors, to more limited motor abilities that necessitate the use of assistive devices such as switch controls) and need to consider external hardware [125]. Generally, developers currently lack tools for identifying, understanding, and implementing accessible features for motor-impaired users [31].

The central challenge of building developer tools that support accessibility for motor impaired users is one of *semantic screen understanding*. That is, in order to determine whether a given UI screen follows motor-impairment accessibility guidelines, the *functional* and *visual* properties of UI screens and individual components must be automatically inferred from a given app. For example, motor-impaired users that make use of hardware devices, such as switches, rely on assistive services (e.g., Android `Switch Access` [3]) that iteratively scan through and highlight individual UI components, as illustrated in Figure **??**. This allows a motor-impaired user to easily select the icon or component with which they wish to interact. However, this process can be slower than traditional gesture-based control [88], and switch users often rely on the *consistency* of certain UI element patterns, such as menus, in order to quickly perform actions. As such, a popular motor-impairment accessibility design guideline for mobile apps [1, 17] states that persistent icons that appear across multiple screens, such as tab bars or menus, should retain a consistent ordering. This allows a user to anticipate which UI elements will be highlighted by the assistive service. However, in order to detect inconsistent icon orderings, an automated tool must be able to accurately identify

functional groups of UI components, such as tab bars, and the ordering of icons within them.

To help advance the current state of developer tools to better support motor impaired users, and overcome the challenges related to automated screen understanding, this paper introduces a novel approach, called MOTOREASE, which aims to automate the detection of **Motor** impairm**E**nt **A**cce**S**sibility issu**E**s in mobile apps. MOTOREASE is a novel approach that leverages automated dynamic analysis, computer-vision, and text-processing techniques to detect violations of motor-impairment accessibility guidelines in a given Android application. The approach is comprised of four detectors each targeted toward a popular UI design guideline meant to support motor impaired users. MOTOREASE's novelty lies in both its technical underpinnings and its ability. MOTOREASE combines multiple neural models for screen understanding, allowing it to recognize screen semantics prior techniques cannot. This allows MOTOREASE to identify violations of motor-impairment accessibility guidelines.

MOTOREASE is designed to seamlessly integrate into existing software testing workflows, and operates in a fully automatic manner by analyzing common artifacts produced by existing Android testing tools. More specifically, MOTOREASE takes as input UI metadata extracted via the Android `uiautomator` utility, along with corresponding screenshots, from an application being run in conjunction with one of any number of existing automated input generation tools [90, 54, 78, 141, 86, 63, 101, 100, 129]. MOTOREASE then passes this data to a series of four detectors, each of which identifies guideline violations. We aim to detect four main violations for design guidelines related to best practices for *Touch-Target Size, Expanding Section Closure, Persisting UI Elements,* and *Icon Distance*.

4.1 Background and Motivation

In order to understand how the MOTOREASE approach functions, it is important to understand how motor-impaired users use their devices. Motor-impaired users use devices in two main ways: Switch input and touch input [144]. As described earlier, switch based input uses an external hardware input device. Switches are common in users with limited to no motor control but without impaired cognitive functions [144]. Most current smartphone and tablet devices offer accessibility services that support scanning a curser across the screen to highlight UI elements and icons (i.e Fig. **??**) [1, 17]. The users can then click the switch to select the UI element that is highlighted, forming a semi-automated form of user input. This input technique can be far slower than traditional touch-based interactions, as it requires waiting for appropriate UI elements to be highlighted by the system. Thus, switch users rely on consistent and accessible app designs to be able to anticipate the future UI elements that will be focused by the scanning service. One example that highlights how tedious this input can be is related to typing speed, where switch users average 3 words per minute (wpm), and touch-based users average closer to 40 wpm [88].

For users with some, but limited motor-control, touch input is typically used with constraints related to physical limitations such as tremors or gesture speed limitations [96]. Some users with lesser motor-impairments may also turn to using voice commands to navigate applications due to difficulties/slowness related to switch operation [144]. Prior work in the HCI research community has aimed to improve gesture recognition for users with limited motor control [113]. Google and Apple also offer sensitivity and touch settings that aim to allow users with limited motor control to customize touch controls for easier interaction [113, 1, 17]. Google and Apple [1, 17] have also developed UI and human interaction guidelines which aim to support motor-impaired users, but as

Table 2: Accessibility guidelines extracted from our systematic literature review of accessibility guidelines – includes recent research and Google's [17] and Apple's [1] accessibility guidelines. (LV = low vision users, DHH = deaf and hard of hearing users)

Accessibility Guideline	Primary Affected User Demographic	Guideline Source	Previous Implementation	Implemented by MOTOREASE
Visual Touch Target Size	Motor, LV	[71, 109]		X
Touch Target Size	Motor, LV	[1, 17, 20, 22, 107, 45, 31, 27, 66, 71]	X	
Persistent Element Location	Motor	[1, 20, 22, 17]		X
Clickable Span	Motor, LV	[31]	X	
Duplicate Clickable Bounds	Motor	[31]	X	
Editable Item Descriptions	LV	[31, 59]	X	
Expanding Section Closure	Motor, LV	[1, 17, 20, 22]		X
Non-Native Elements	Motor, LV	[17, 45]	X	
Motion Activation	Motor, LV	[1, 17, 20]		
Labeled Elements	Visual, DHH	[31, 60, 76, 59]	X	
Screen Captioning	LV, DHH	[1, 17, 20, 22, 12, 16, 119, 76, 111, 66]	X	
Keyboard Navigation	Motor, LV	[12, 16, 60, 76, 52]	X	
Traversal Order	Motor	[16, 31, 60]	X	
Adjacent Visual Icon Distance	Motor	[1, 17, 22, 139, 27, 107]		X
Proper Information Organization	Motor, LV	[45]	X	
Facial Recognition	Motor	[45, 32]	X	
Single Tap Navigation	Motor	[1, 17, 20, 22, 60, 94]		
Poor form design/instructions	Motor, LV	[12, 16]		

past work has illustrated, developers may often be unaware of such guidelines or ignore them due to other development constraints [122].

In accordance with the work done at an operating system level to create more native accessibility features, industry leaders such as Google and Apple [1, 17], who run the two largest App stores, have developed guidelines for developers to make more accessible apps on their own [15, 10]. These guidelines are UI and metadata guidelines that help developers deign and implement UIs that allow disabled users navigate the UI more efficiently and help native accessibility tools provide more information to the user. Though these guidelines are available to developers, they are not mandatory to follow and many developers are unaware of such guidelines, often ignoring them and creating inaccessible apps [122]. We explore the guidelines we detect through out approach and why we decided on the specific ones we chose.

4.1.1 Accessibility Guideline Literature Review

In order to fully capture the current landscape of accessibility guidelines that may impact various populations of users, and to aid in selecting the most impactful guidelines that aim to assist motor impaired users we conducted a systematic literature reviews on research at the intersection of

software engineering, human-computer interaction, and accessibility. To conduct this review, we followed the methodology set forth by Kitchenham et al. [70]. We defined a single research question that asked *"What accessibility guidelines have been identified and discussed in prior research?"*. We used the relatively simple search string of "accessibility" to search DBLP, the ACM Digital Library, and IEEE Xplore, for work at the intersection of accessibility and software engineering for the date range of January 2010 - December 2022. The purpose of using such a simple search string was to "cast a wide net" and ensure that we did not miss important work. We defined inclusion criteria as follows: (i) must have been published in our studied date range, (ii) must have been published at one of 16 conference venues (ICSE, FSE, ASE, ICSME, MSR, ICPC, ISSTA, ICST, SANER, UIST, CHI, SPLASH, OOPSLA, PLDI, CSCW, ASSETS) or 5 journal venues (TSE, TOSEM, EMSE, JSS, ASE) that cross cut software engineering, HCI, and accessibility, (iii) the paper must describe a study or developer tool directly related to an accessibility issue that impacts end-users. The scope of our search was limited to these venues and digital libraries as they provide the highest quality of research in all matters including accessibility. Our search results returned 2948 papers from our selected conferences within our given date range. Then, two authors manually checked each paper for adherence to the final inclusion criteria, resulting in 20 papers that intersect our desired research areas *and* discuss developer guidelines for addressing accessibility issues. In addition to these 20 identified primary studies, we also examined Apple's and Google's design guidelines related to accessibility [1, 17], as several of our primary studies referenced these sources.

After the search process concluded, one author extracted all accessibility guidelines discussed in the papers and platform documentation, and after the process, three authors met to discuss and verify that all guidelines were properly extracted in a joint meeting. This process resulted in the derivation of 18 *accessibility design guidelines*, which are illustrated in Table 2. In this table, we provide (i) a short description of the guideline (with full definitions and examples available in our online appendix), (ii) the primary affected user groups, (iii) the sources that described the guideline, (iv) whether or not automated support for guideline has been implemented in past commercial or research developer tools, and (v) the guidelines targeted by MOTOREASE. **It should be noted that none of the guidelines that MOTOREASE targets have been explicitly targeted by prior tools.** While touch-target size has been explored in prior work [31, 27, 66, 17], *visual* touch-target size, which is of critical importance for motor-impaired users [71], has not been explored. While there is some recent work on Keyboard Accessibility failures in web applications that could affect Motor-impaired users [53], this work does not explicitly target any of the guidelines targeted by MOTOREASE. The Groundhog [123] tool is also inadvertently able to detect *some* expanding section closures. The lack of exploration of these guidelines is largely due to the fact that implementing tools that detect when such guidelines are *not* followed requires new types of automated UI screen understanding.

Of the many guidelines identified, MOTOREASE supports identifying violations for four of them, *Touch-Target Size, Expanding Section Closure, Persisting Elements, and Icon Distance*. These guidelines were selected for three main reasons: (i) current lack of implementation (ii) the need for complex screen screen understanding and less trivial methods of analysis (iii) integration with the output of automated UI testing tools for Android. Our goal in conceptualizing and implementing MOTOREASE is to create a tool that can effectively identify and address motor accessibility issues that that remain undetectable through conventional static analysis techniques. Our primary objective entailed the formulation of a set of guidelines that require a more sophisti-

Figure 3: Illustration of four studied accessibility guidelines

cated understanding of the screen, enabling a more nuanced search for motor accessibility barriers within mobile applications. We are confident that the combination of these guidelines, previously unexplored, in conjunction with UI comprehension, will result in a highly robust and functional accessibility testing tool, that caters to the unique requirements of individuals with motor impairments.

The four accessibility guidelines targeted by MOTOREASE are (i) ***Visual Touch Target Size***, (ii) ***Persistent Element Location***, (iii) ***Expanding Section Closure***, and (iv) ***Visual Icon Distance***. One of these guidelines (*Visual Touch Target Size*) was identified from a prior accessibility study, whereas the others come directly from Apple and Google's accessibility deign guidelines for mobile apps [1, 17]. These four guidelines were chosen as they had not been implemented by past work and could easily integrate with automated input generation (AIG) tools for Android, which is an important practical component of MOTOREASE as we explain in Section **??**. In the following subsections, we describe each motor-impairment accessibility guideline in detail, and the screen understanding challenges in detecting guideline violations. More detailed descriptions of all guidelines can be found in our online appendicies [5, 6, 7].

4.1.2 Expanding Section Closure

Pop up menus and modal dialogs can provide meaningful information to the user, but closing them can be non-trivial for motor-impaired users who utilize a switch, as they may not contain explicit UI elements for closing the menu or dialog. As such, UI design guideline advocated for by both Apple and Google [1, 17] state that such closure UI elements should be present and easily interactive. Many expanding sections can be closed through a swiping gesture to dismiss the section or an external tap on the screen not within the bounds of the expanding section. Both of these options pose an accessibility issue for motor-impaired users due to the need for gestures and

assumptive tapping on non-intuitive screen locations. A violation and adherence to this guideline is illustrated in Figure 3-1. Detecting UIs that violate this design guideline can be difficult as it requires automatically identifying (i) whether a pop-up menu or modal dialog is present within a given screen, and (ii) whether or not a UI element supports closing the pop-up.

4.1.3 Visual Touch-Target Size

Motor-impaired users who experience tremors in their hands can experience difficulty tapping precisely on icons. This makes it difficult for them to interact with elements as intended. Apple and Google suggest minimum UI element sizes of 44x44 pixels and 48x48 pixels respectively, such that users with minor motor impairments can more easily tap icons [1, 17]. Typically, the "size" of a UI element is defined by the *touchable area* of that element, and not the *visual area* occupied by the pixels of a given element. However, as stated above, it is important to provide sizable *visual* touch targets to users with more limited motor control, such that they can better hone their more limited movements to tap desired UI elements. Most past work that aims to identify icons or UI elements that do not meet a given minimum threshold read UI metadata to examine the "touchable area" only, even if the visual size of the icon does not fill the entire area. Thus, this can create a disconnect between the *visual* touch target size, and the *touchable area*. An example of this is shown in Figure 3-2. The  icon next to the Instagram logo has a touch target size of larger than 44x44, however, the visual size of this icon is quite small, a making it difficult to tap. Detecting such issues can be difficult as it requires automatically inferring the visual area on a screen occupied by a given UI element. MOTOREASE aims to overcome this issue by leveraging optical character recognition and neural object detection techniques.

of just motor-impaired users.

4.1.4 Persistent Element Location

Applications link various screens together to aid users in completing complex tasks, however, certain UI elements need to exhibit *consistent* placement to assist switch users with anticipating UI element scanning. This guideline specifies that icons that appear across multiple screens should appear in the same general area of the screen [1, 17]. This means the locations of elements such as back buttons or search icons that appear across multiple different screens should remain consistent. An example of a back button with a consistent location is illustrated in Figure 3-3. Violations of this guideline can be difficult to detect as it requires automatic identification of corresponding UI elements across screens which may exhibit visual variability (e.g., displayed on different backgrounds).

4.1.5 Adjacent Visual Icon Distance

The design and placement of interactive icons that signal functional affordances is critical to ensuring a positive user experience, particularly for individuals with motor impairments. For users who may struggle with fine motor movement, it can be difficult to tap a single location on the screen without accidental triggers of other areas [71]. As such, the *Adjacent Visual Icon Distance* guideline states that adjacent "clickable" UI elements should be positioned at least eight pixels apart from one another. This can be challenging as it again requires the automated inference of the visual area occupied by different UI elements. In this context, the issue of miss-clicks may

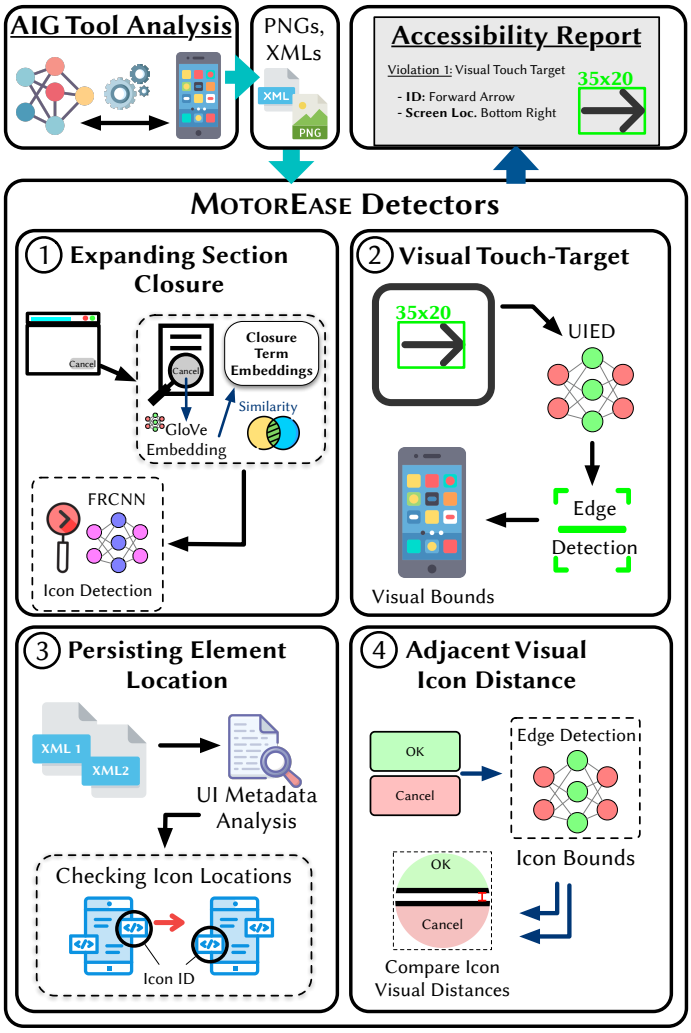

Figure 4: Overview of MotorEase's Workflow

trigger unwanted functions within the app. It is imperative that interactive icons can be accessed individually without any risk of accidental activation of adjacent icons. Google's Android Accessibility Guidelines [17] stipulate that icons must be placed a minimum of 8 pixels apart to prevent miss-clicks by users with tremors or other motor impairments. Ensuring proper icon placement is crucial in enabling users with fine motor difficulties to interact with these icons effectively and without any possibility of error. Thus, the significance of this detector lies in the need to test for proper distance between icons. An example of this is shown in Figure 3-4, wherein the "Start" and "Privacy" elements are located too close to one another and may result in accidental, unintentional triggering of the components by a motor-impaired user.

4.2 MotorEase Approach

MotorEase is an automated approach that aims to detect motor-impairment accessibility guideline violations by analyzing UI metadata and screenshots collected via automated input generation (AIG) tools (i.e., automated app crawlers, UI testing tools). MotorEase operates in three

stages, and implements four guideline violation detectors, as depicted in Figure 4. First, an AIG tool is run on a target application to produce a set of screenshot and uiautomator XML files (i.e. UI metadata) before and after each AIG tool action. We tailor our approach to utilize UI metadata generated using the uiautomator framework, which captures UI layout information in a structured XML format, as this is most prevalent utility used by recent Android AIG tools [90, 78, 63, 101, 100, 129, 80, 82, 81, 146]. It should be noted that MOTOREASE does not require any pre-existing test cases, but instead can be used in conjunction with any of the AIG tools listed above. Second, MOTOREASE utilizes a series of four *violation detectors* to analyze the screenshots and UI metadata to determine if the target application failed to follow motor-impairment guidelines. Finally, MOTOREASE collects the information from the detectors and compiles an *accessibility report* that informs developers of accessibility guideline violations.

4.2.1 Detectors

The core components of MOTOREASE are its four accessibility guideline violation detectors. Detectors ①, ② and ④ operate upon *single* uiautomator XML files, screenshots, in the form of PNGs, or both. Detector ③ takes as input a series of *multiple* `XML` and screenshot pairs. In the remainder of this section, we describe the technical underpinnings of each of MOTOREASE's detectors.

Expanding Section Closure Detector

The expanding sections detector aims to identify pop up messages or slide-in views that lack a visible means to close the section. Objects or text that imply closing the section is what MOTOREASE aims to detect, if it cannot detect these, then a given screen with a dialog box or section is considered to be in violation of the guideline. This detector begins by determining whether the screen has an expanding section and then extracting it from the screenshot. This is done by identifying the largest element on the screen. MOTOREASE extracts the largest *android.widget.FrameLayout* and the largest *android.widget.ListView* on the screen whose size is not the entire screen as the pop up screen or the slide-in list menu. An example of this is shown in Figure 5.

Table 3: Mapping of Sample Lexical Patterns to detect Closure

Initial Closure Words
”close”, ”cancel”, ”dismiss”, ”done”, ”ok”, ”finish”, ”return”
GLoVE Embedding Words
”deny”, ”allow”, ”exit”, ”end”, ”terminate”, ”quit”, ”back”, ”stop”, ”ignore”, ”proceed”, ”save”,”apply”, ”submit”, ”confirm”, ”abort”, ”decline”, ”reject”, ”ignore”

Once MOTOREASE has extracted each of the expanding sections, it then aims to determine whether the pop-up or section provides a clear means of closing it. In order to offer a robust solution, MOTOREASE accomplishes this via two main procedures: (i) Semantic Text Matching

and (ii) Icon Detection. This is due to the fact that closure controls can have either textual (i.e. the word "exit") or visual (i.e. an $\times$ icon) signifiers that indicate functionality.

Semantic Text Matching: MOTOREASE's semantic matching technique defines a certain set of keywords that are likely to signify an element that can close an expanding section or pop-up. These keywords comprise common lexical patterns derived through two authors of the paper examining expanding sections that appear in 1500 randomly sampled screenshots from the RICO dataset [57]. The RICO dataset, comprising 9,000+ Android apps and 66,000+ screenshots, serves as a popular resource for mobile app research given its diverse set of screens and apps. We randomly sampled 1500 screens as it represents a statistically significant sample size of the 66k screenshots present in the RICO dataset (95% confidence level and 2.5% margin of error). These words were terms that implied a closure or completion action, i.e. "close", "dismiss", "cancel", "ok". To ensure that the selected words for semantic matching were relevant, we employed a manual process in which two authors reviewed the randomly sampled dataset for expanding elements and words that implied closure. The resulting set of words was agreed upon by the authors as suitable for the task. To further expand this set of words, we further utilized GloVe embeddings [114] to compare the original selected words to the entire dataset. Glove embeddings capture semantic relationships between words by considering global word co-occurrence patterns, resulting in dense vector representations that preserve meaningful similarities between words [114]. We extracted additional words that exhibited a cosine similarity of at least 0.95 to the GloVe embeddings of the original selected words. This approach enabled us to carefully curate a comprehensive set of words for semantic matching that accurately represented closure. We provide the complete list of 25 closure words in Table 3 – as our experiments illustrate, we found these words to generalize well to our experimental benchmark.

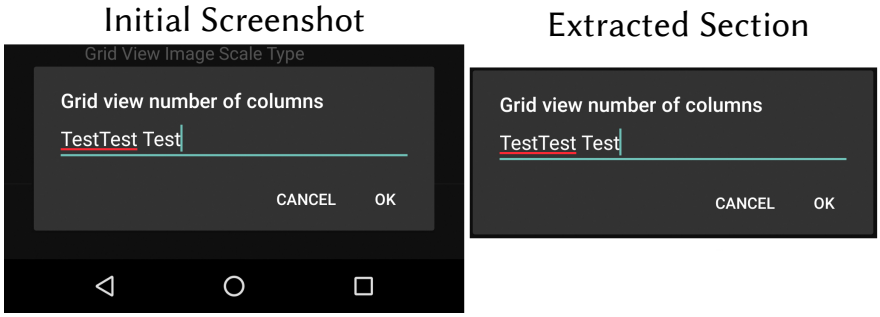

Figure 5: Extracted Expanding Sections

To extract the text from each image, MOTOREASE leverages a combination of Google's Optical Character Recognition (OCR) text extraction [4], which is based on the EAST OCR technique [148] and the text present in the `uiautomator XML` file. We use both methods of text extraction as text displayed on the screen via images is often not captured in the `uiautomator XML` files. This method provides a binary classification for the presence of text that indicates a means to close the pop-up. If there are no matching extracted terms, MOTOREASE then proceeds with icon detection.

Icon Detection: Icon detection is used to identify specific closing icons, i.e. $\times$, hamburger icon, checkmark, etc. Two authors examined the same set of 1500 screens from the RICO dataset discussed above and compiled a set of base icon types to train an image detection model in order to

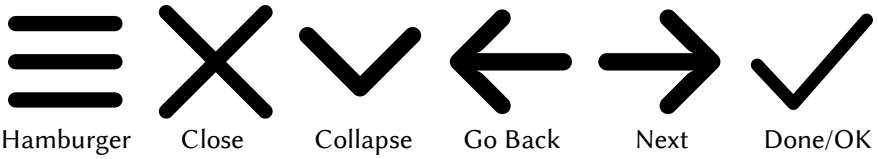

Figure 6: Base Closure Icons

detect these icons. The chosen base icons are shown in Figure 6. To accurately perform this detection, we trained a neural object detection model on a diverse set of examples of these identified icons, through a process we describe in detail below.

Training a neural object detection model typically requires a large-scale dataset with annotated examples of the target icons. To derive such a dataset, we *automatically* constructed a realistic, synthetic dataset of icons that represent menu/pop-up closing. To do this, we extracted icon images with transparent backgrounds from the Fontawesome [1] and Flaticon [2] image repositories until five icons for each icon type were identified that varied in color and style. Note that these icons do not directly appear in our evaluation dataset. We then superimposed these icons into random locations on screenshots derived from the RICO dataset [57]. During the analysis of the 1500 RICO screenshots two authors determined that a majority of icons on the screen were mathematically smaller than 10% of the total screen size and larger than 2% of the screen size. The screen size in question are the 1920x1080 pixel dimensions of the screen, limiting the maximum icon size to 192px and minimum icon size to 38px. Therefore, when an icon was superimposed on a screen, we varied their size between 2% and 10% of the screen area so that they remained relatively similar to the screens in the dataset. In this manner, we generated a dataset of 7,291 images (separate from the 1500 images sampled earlier) all with labeled and fully-localized icons on them (one per screen, spread evenly across the variations of our six icon types) and divided this into training and testing sets following an 80/20 split, 5,832 images for training, 1,458 for testing. We then used this dataset to train a Faster-RCNN object detection technique [118] using the the torch-vision API [21]. We generated $\approx$ 7k images as past work that uses a similar approach for icon detection was able to train an accurate model with this scale of data [42, 41, 64]. Our trained model was able to achieve over 95% accuracy on the test portion of our dataset.

MOTOREASE passes the cropped out expanding sections to the model in order to detect the icons. This detector works by first checking for semantic text matches and if there is no match, it then checks for icons. MOTOREASE performs text pattern matching first because of the potential for X icons on the expanding section *not* related to closing the icon. Had we done the icon detection first, the X icons to delete text in the text-fields would have been detected by the object detector. This means that this screen would have been classified as closable even though the X icons do not imply closing of the section. Therefore we use both the text and images on the expanding sections to try and classify if it can be closed. If neither technique can pick up on a pattern or icon, MOTOREASE reports the section as a violation to the expanding section guidelines, capturing the the screenshot name, FrameLayout/ListView name, and violation and make it known to the developer.

[1] `https://fontawesome.com`
[2] `https://www.flaticon.com`

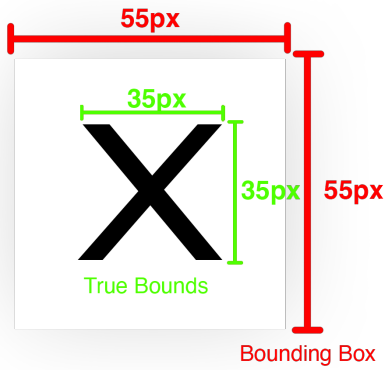

Figure 7: Example of Bounding Box vs. True Bounds

Visual Touch-Target Detector

This detector uses both the screenshot and its corresponding XML file. It starts by processing the XML file and extracting the bounds for all elements which are clickable. XML has various properties in its metadata to describe an element, and if an element has a *"True"* in the *"clickable"* field, we determined that it is meant to be clicked on or interacted with. This detector aims to identify elements that have a visible area that is smaller than their tappable/clickable bounding box. Hence with this detector, MOTOREASE aims identify elements whose *visual* size are under 48x48 pixels [10], even when the reported touchable area (as indicated by `uiautomator`) may be larger than 48x48. The bounding boxes in `uiautomator XML` files can show a bounding box whose size is larger than the actual size of the icon. An example of this is shown in Figure 7. The true bounds of an icon corresponds to the visible area occupied by the icon. This can make these bounding boxes appear to be guideline abiding since they are generally larger than the true, visible bounds of the icon they hold. In order to identify these cases, MOTOREASE adds 15 pixels to the width and height of the bounding boxes of each clickable item to extract the entire icon, before cropping the enlarged icon from the image.

After MOTOREASE extracts the element, it is used as input to an edge detection algorithm, implemented in the UIED tool [103], which is an approach that combines both neural object detection and unsupervised edge detection to effectively segment mobile app UI screens. This edge detection procedure is able to derive the *visual* bounds of a given UI element, and would identify the 35x35px edges for the "X" as shown in Figure 7. This procedure is applied to clickable icons extracted on each screen. Once the true edges are identified, MOTOREASE is then able to then able to compare these bounds to the reported element size in the `uiautomator XML` file. If it is determined that the true element width or height is less than 48 pixels, MOTOREASE labels that individual icon as a violation.

Persisting Elements Detector

The persisting elements detector aims to identify an icon on a screen whose functionality remains the same, but location changes across multiple screens. An example of this is the navigation bar at the bottom of most applications. We expect that if the navigation bar is at the bottom of the

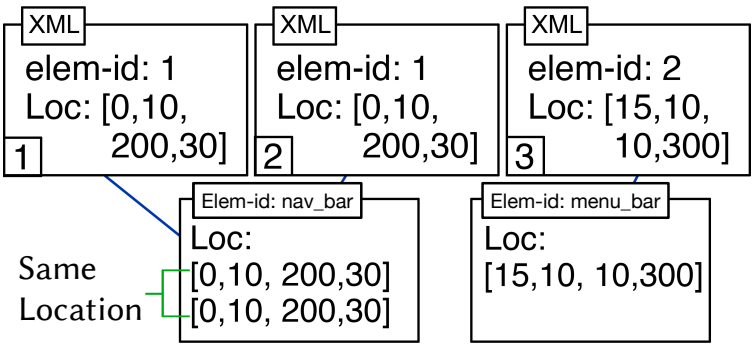

Figure 8: Detection of Persisting Elements

screen on one screen, if a second screen has a navigation bar it should also be at the bottom of the new screen. This detector requires the use of both screenshots and `uiautoamtor XML` files from multiple screens, as MOTOREASE directly analyzes properties of UI elements in the `uiautoamtor XML` files and compares the visual UI element similarity across multiple files.

This detector parses all of the `XML` files for a given application and records all of the UI element IDs. Then, it collects the location(s) across all `XML` files for each individual element ID. If there was more than 1 instance of a given ID across multiple screens, MOTOREASE checks to determine whether the location bounds were the same as well as checking if the elements within those bounds are visually similar (95% similar according to pixel-based mean squared error). If they are not, it deems it a violation of the persisting element guideline. This ensures that MOTOREASE is checking every element in the application while checking to see if the icons with similar IDs have similar visual properties. If there is an element that appears across the `XML` files more than once, MOTOREASE examines them to see if the location is the same for the element. It relays this information back to the developer by providing the ID of the violating element in the generated Accessibility report.

Visual Icon Distance Detector

The icon distance detector is designed to analyze icons on a screen and determine whether the visual distance between any two icons is less than 8 pixels, which combines findings from Kong et al. [71] regarding visual icon size/spacing and recommendations from Google's accessibility Guidelines [17]. T

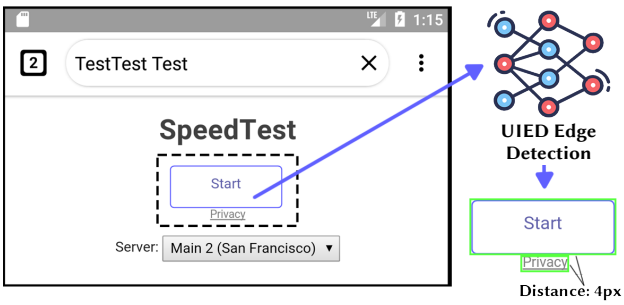

Figure 9: Visual Icon Distance Detection Process

his detector analyzes a XML and screenshot pair, and extracts all of the *clickable* elements on the screen using the `uiautomator` metadata. Once the clickable elements are identified, MOTOREASE crops out the icons and sends them as input to UIED [103] to find the true visual bounds of the icon via edge detection. The true bounds are then used to subtract the padding from the initial touch target to the visual touch target. These modified bounding box values are then stored, and the detector iterates through all of the other icons on the screen to calculate the distance between the target UI elements and all other elements. The distance between the bounding boxes is calculated by determining the horizontal, vertical, or diagonal distance in pixels between two bounding boxes on a screen.Once the distance between the icons is computed, MOTOREASE checks whether there is a distance is less than 8 pixels. If a violation is detected, MOTOREASE includes a description of the UI elements in the generated accessibility report allowing the developer to address the issue and improve the accessibility of their application.

4.2.2 Accessibility Report Generation

The accessibility report is generated using the target app screenshots that contain violations and a generated markdown file, with filenames to specific images of violations for each detector. We developed textual templates for each type of violation that are used by the report generation engine to automatically describe the accessibility violations by filling in the templates with information the MOTOREASE analysis. This accessibility report aims to provide a comprehensive account of accessibility issues within a given app so that developers are able to take this information to make any changes they may need in order to make their apps more accessible.

4.3 Evaluation Methodology

In this section, we describe the procedure we used to evaluate MOTOREASE. The goal of our empirical study is to assess the accuracy, and practical utility of the approach. The main *quality focus* of our study is to determine the extent to which MOTOREASE is able to detect accessibility violations in applications. To achieve our study goals, we formulated the following five research questions:

- **RQ**$_1$ *How accurate is the Expanding Section detector?*
- **RQ**$_2$ *How accurate is the Visual Touch Target detector?*
- **RQ**$_3$ *How accurate is the Persisting Element detector?*
- **RQ**$_4$ *How accurate is the UI Element Distance detector?*
- **RQ**$_5$ *Does* MOTOREASE *identify a limited number of false positive and negative violations?*

4.3.1 RQ$_1$ - RQ$_4$: Violation Detection Capability

MOTOREASE is one of the first tools to support the detection of violations of accessibility design guidelines targeting motor-impaired users, and accomplishes this via understanding the visual and textual modalities Android UI screens. Given that prior techniques are not able to explicitly detect the design violations that MOTOREASE targets, by using the visual comprehension of the screen,

Table 4: Expending Sections Detector Test Dataset

Total Files	Close: Icon	Close: Text	Cannot Close
483	27	214	242

we both derived an entirely novel benchmark and designed an evaluation methodology that tests each of its four detectors individually to determine their accuracy, precision, and recall. In addition, we also examine the false positive and false negative rates to better understand the practical utility of MOTOREASE. The evaluation metrics used in this study provide insights into the ability of each detector to identify true positives and true negatives. Accuracy gives us an overall ability to deduce each detectors ability to detect true positive (TP) and true negative (TN) values accurately. Note that we balance the positive and negative samples in the MOTORCHECK benchmark to allow for an informative accuracy measurement.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

In the context of our study, a True Positive (TP) is defined as the detection of an existing design guideline violation as defined in our ground-truth dataset. A false positive (FP) is defined as the detection of a violation when one does not exist. A False Negative (FN) occurs when our approach does not report a violation, but one exists in the ground truth. Finally, a True Negative (TN) occurs when the approach does not report a violation and one does not exist. In addition to accuracy, we also measure the precision and recall (as defined below) to provide a more complete picture of MOTOREASE's performance. The results of MOTOREASE were manually validated (i.e. two authors compared MotorEase's output to the ground truth.)

$$Precision = \frac{TP}{TP+FP} \quad Recall = \frac{TP}{TP+FN}$$

Given these two evaluation metrics, we can determine how accurately each detector works. MOTOREASE's overall effectiveness can be derived by averaging the the accuracy/precision/recall across all four detectors, providing a comprehensive understanding of MOTOREASE's ability to detect accessibility guideline violations.

4.3.2 RQ$_5$: MOTOREASE's Practical Utility

In order to investigate MOTOREASE's practical utility, we also report both the false positive and false negative rate, as these reflect the need for a developer to sift through incorrect violation reports, or lost quality in terms of miss unreported violations. This helps to provide a more holistic picture of MOTOREASE's performance.

$$FalsePositiveRate = \frac{FP}{FP+TN} \quad FalseNegativeRate = \frac{TN}{FP+TN}$$

Table 5: Visual Touch-target, Persisting Elements, and Visual Icon Distance Test Datasets

Detector	Total Files/Apps	Violations	Non-Violations
Touch-Target	400 files	176	224
Persisting Elements	49 apps	24	25
Icon Distance	400 files	42	358

4.3.3 Derivation of the MOTORCHECK Benchmark

Given that no prior approach has targeted the motor-impairment design violations targeted by MOTOREASE, we develop a novel benchmark called MOTORCHECK which we discuss below. It should be noted that all of the accessibility violations of this benchmark are real, no synthetic violations were injected in it's construction – instead real violations were rigorously manually annotated.

To derive the initial set of screenshots and xml files for MOTORCHECK we applied the CRASHSCOPE [100] automated testing tool to 70 popular Android apps that are cross-listed on both FDroid and Google Play. To do this, we gathered a list of apps from F-Droid [24] and considered only those apps that were cross-listed on Google Play [25] and had at least 1000 downloads – providing some confidence in the popularity of the chosen applications. We provide a full list of these applications with download statistics and links in our online appendicies [5, 6, 7]. During this process, we used one of CRASHSCOPE's exploration strategies to extract 2,864 screenshot/XML pairs. Note that the goal of our study in assessing MOTOREASE's capabilities is independent of the coverage provided by the underlying testing tool, although CRASHSCOPE has been illustrated to be competitive with other tools [101]. It should be noted that the screen coverage of MOTOREASE is dependent upon the Android AIG tool that the approach is paired with. Given recent advances in AIG tools [134], MOTOREASE can integrate with these new tools and take advantage of the improved coverage. Next, we describe the dataset derivation process for each detector. Note, given that the data labeling process is quite objective for violations of our identified guidelines, for each dataset, we had one author manually label each instance, and another author verified the results. During this process, no instances of conflicts were noted, again due to the largely objective nature of the labeling procedure. We provide an overview of the MOTORCHECK benchmark data in Tables 4 & 5.

Expanding Section Closure Detector Dataset

In order to detect expanding sections, this detector requires an input screenshot and its corresponding XML file. In order to evaluate the detector and remove any bias, one author labeled expanding sections without closure elements until all CrashScope files were exhausted, resulting in 241 screens. Of the 241 screens that had an expanding section, there were 121 screens were FrameLayouts and the remaining 120 were ListViews. Then, in order to balance the dataset, an additional 242 screenshots without violations were randomly selected to complete the dataset, for a total of 483 screens. The additional screens without violations consisted of expanding sections that could be closed. Table 4 provides more information on how the dataset was split between violations and non-violation samples. Labeling was done manually using LabelStudio [26]. For the screenshots

Table 6: Overall Results for MOTOREASE Detectors & Baselines

Approach	Precision	Recall	Accuracy	F1-Score
MOTOREASE (Viz T.-Target)	1.0000	0.6648	0.8525	0.7986
G-Accessibility Scanner (T.-Target)	0.5556	0.5085	0.6025	0.5310
MOTOREASE (Exp. Sec)	0.9042	0.9205	0.9123	0.9129
Groundhog (Exp. Sec)	0.6849	0.8659	0.7207	0.7648
MOTOREASE (Pers. Elem)	0.8214	0.9583	0.8776	0.8846
MOTOREASE (Icon Dist)	0.7119	1.0000	0.9575	0.8317
MOTOREASE (All Detectors)	**0.8594**	**0.8859**	**0.8999**	**0.8570**

without violations, icon types and closure word types were labeled. If neither the icon nor text clearly showed a means of closing the section, it was labeled as a violation.

Visual Touch-Target and UI Element Distance Detector Dataset
The touch-target detector requires screenshot and XML pairs to determine if the screens had a touch-target violation (i.e. XML) and visual bounds differed. In order to evaluate the detector and remove any biases, we randomly chose 400 screenshots and XML pairs generated by CrashScope stratified across our 70 applications. This sample size was used as it represents a statistically significant sample of the total number of extracted CrashScope screens (99% confidence interval). Table 5 provides the dataset splits between violations and non-violation samples. Labeling for these screenshots was performed manually. One author analyzed each of the screenshots and set a bounding box on each of the interactive elements on the screen using Label Studio [26]. If the size of the bounding box was less than 48 in width or height, it was labeled it as a violation, else it was labeled as a screen without violations. Additionally, the author checked the distance between all components on these screens and labeled any instances where UI elements were less than 8 pixels apart. This was done for all 400 images.

Persisting Elements Detector

The persisting elements detector requires multiple app XMLs in order to detect violations. The CrashScope dataset [100] contains 70 applications and their screenshots. We filtered out apps that only contained screenshots of similar screens, resulting in 49 applications, 24 of which had persisting elements that violated our guideline, and 25 of which adhered to our guideline. One author labeled each app as having a violation or not having a violation. During the labeling, this author also specified which specific screenshot exhibited the violation.

4.3.4 Comparison to Baseline Techniques

While the Accessibility issues that MOTOREASE targets have not been explicitly targeted by past tools, there are two tools which are capable of detecting a subset of the accessibility violations identified by MotorEase. These two baselines are Groundhog [123] and Google Accessibility Scanner [2]. We ran these two tools on the same MOTORCHECK benchmark used to evaluate MOTOREASE to keep the comparison fair and consistent, and we report the same metrics for

both MOTOREASE and the baseline techniques. Upon careful analysis of these baselines, Google Accessibility Scanner and Groundhog are only capable of detecting Touch-target size violations and Expanding Sections violations, respectively.

The first baseline tool we compared MOTOREASE to in our updated evaluation is Google's Accessibility Scanner [2]. This tool operates directly upon the dynamic representation of the GUI as reported by uiautomator, and checks to whether the bounds of screen elements fall below the 48x48 dp threshold. To apply this tool, we launched the each app included in the MotorEase dataset on an emulator of the same screen dimension (1920x1080) and Android version as the screens in the MotorCheck benchmark and manually navigated to the screen in question, and triggered the Accessibility Scanner tool. One author then manually compared the output results from Accessibility Scanner to the ground-truth visual touch-target size violations defined in MOTORCHECK. We then reported the Precision/Recall/Accuracy and F1 Score Metrics.

The second baseline tool that we compared MOTOREASE to is the Groundhog Accessibility tool [123]. We worked directly with the authors of the Groundhog tool, who were quite helpful after some initial issues initializing the tool, in order to apply Groundhog to the MOTORCHECK benchmark. Groundhog functions by first sending touch-based actions to a given Android app screen running on an emulator or real device, and then attempts to exercise the same actions using one of Google's accessibility services, such as Talkback. Given the manner in which Groundhog works, the only Accessibility issue in MOTORCHECK that it was applicable to is the Expanding Section guideline violations. To apply Groundhog to detect Expanding Section violations in MOTORCHECK we launched a target on an Android emulator configured to the 1920x1080 screen size and Android version of the MotorCheck screens, and then ran Groundhog on the screen with an expanding section. Groundhog then navigated the screen both with and without assistive services to determine whether it could close the expanding section, if there was a discrepancy, this was reported by Groundhog. Then one author manually checked the output of Groundhog to determine if it was able to detect an expanding section violated the MOTORCHECK guideline and could not be closed by a tappable element. We then reported the metrics seen in Table 6

4.4 MOTOREASE Evaluation Results

4.4.1 RQ_1: Expanding Section Detector Accuracy?

The expanding section detector performed well across nearly all of our studied metrics, as indicated in Table 6. With a precision of .9042, F1-Score of 0.9129, and an accuracy of 0.9123, this detector shows promising results that it is capable of identifying a section's "collapsibility" accurately. This indicates that, by using MOTOREASE, developers will have an increased chance of identifying sections that were designed without a method of closure which may impede motor-impaired users.

Importantly, as per Table 6 our approach surpassed the performance of Groundhog's ability to detect closable sections, with a precision of 0.6849, demonstrating its effectiveness in comparison. The difference between the two is attributable to the fact that MOTOREASE and Groundhog do not have the same objective. MotorEase aims to detect the presence or absence of closing icons, while Groundhog aims to determine whether a closing icon can be accessed using an assistive service. If a closing icon is missing, Groundhog cannot detect it. Since there is no closing icon, Groundhog does not even attempt to close it using an assistive service. Groundhog relies on an accessibility service to detect icons on the screen, However, if the icon does not have any metadata indicating

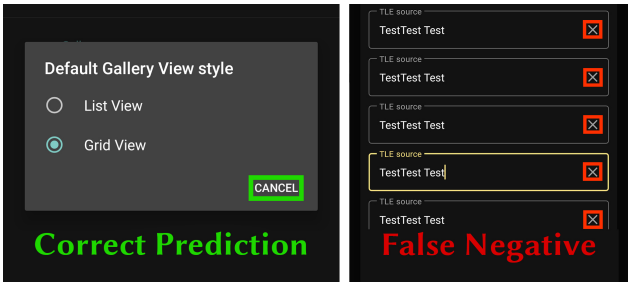

Figure 10: Expanding Section Closure Detection

that it is an interactive, it is unable to interact with the icon and close it. MOTOREASE's novelty lies in its ability to consider the visual presence of the icon independent of its metadata description. However, MOTOREASE's detector does struggle to detect certain instances in the MOTOREASE benchmark. An example of a successful and unsuccessful prediction is shown in Figure 10. The example on the left side is an expanding section that the detector correctly identified as collapsible. It is correctly identified by MOTOREASE as a closing section because of the semantic matching's ability to generalize the word "cancel" as a means of closing the section. The image on the right is not collapsible but the detector deduced that it was collapsible. This was due to the use of X icons in the training data for the object detection model. The "X" icons used in this image imply the deletion of text, but this detectors object detection model is also trained to identify "X" icons that may be used to close the expanding section. It should be noted that this was an outlier in our dataset, and that the pattern for detecting "X" icons generally worked as expected.

4.4.2 RQ_2: Visual Touch Target Detector Accuracy?

The touch-target detector performed well as illustrated in Table 6. The detector exhibited perfect precision, an F1-Score of 0.7986, and an accuracy of 0.8525, showing encouraging results of its ability to detect and classify screens with touch target violations well. Given the results, this detector is successfully able to give developers an insight into smaller icons that may inconvenience users with tremors and inaccurate touches. Importantly, our approach surpassed Google Accessibility Scanners's ability to detect visually small elements on the screen, which only had a precision of 0.5085, demonstrating its effectiveness in comparison. The primary reason for the gap in performance is that Google Accessibility Scanner is not able to check the *visual* touch-target size, and can only parse the reported size from the `uiautomator` framework, which may not necessarily correspond to the visual touch target size. However, while nearly all violations returned by this detector are correct, it does tend miss certain types of violations. For instance, it cannot detect icons on the screen which are not labeled as clickable in the XML. By default, all elements on an android device have a clickable property with a boolean True/False label. If that property is labeled as False, MOTOREASE does not consider the object to be clickable. Dynamically generated screenshots may not always have the metadata information for each element on the screen, and this absence of information was the main reason for incorrect or missed violations. This detector could be augmented in the future with work from the HCI community aimed at assessing icon tap-ability [130].

4.4.3 RQ_3: Persisting Element Detector Accuracy?

This results of this detector are presented in Table 6. With a precision of 0.8214, F1-Score of 0.8846, and accuracy of 0.8786, this detector shows encouraging signs of viability. This detector's recall rate of 0.9583 suggests that the detector identifies true positives at a high rate.

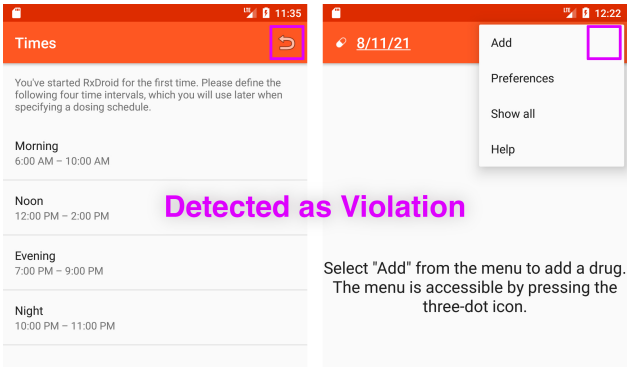

Figure 11: Persisting Elements Detection

This detector, however, relies heavily on the XML to locate elements on the screen, which can lead to mis-classifications. One such example is shown in Figure 11. The example shows the undo icon on the screenshot on the left and a menu on the right side where the undo icon would be. The XML for this second screen has the undo icon in the data, but its bounds and information are missing since they are not visible on the screen. This was classified as a violation though it is not a violation.

4.4.4 RQ_4: Visual Icon Distance Detector Accuracy?

This results of this detector are presented in Table 6. With a precision of 0.7119, F1-Score of 0.8317, and accuracy of 0.9575. This detector achieved a perfect recall rate, suggesting that this detector provides developers with a reliable tool that is capable of accurately detecting closely placed icons, prompting potential UI design and icon placement adjustments.

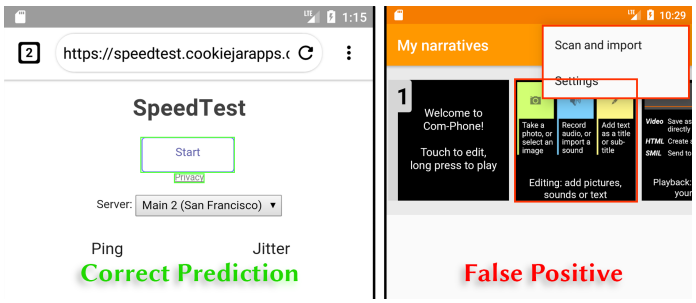

Figure 12: Visual Icon Distance Detection

Like the Visual Touch-Target Violation detector, this detector relies heavily on the `uiautomator` metadata specifying clickable components, and the accuracy of the UIED element bound detector. The latter led to certain cases of inaccurate reporting of violations, due to incorrect overlapping

bounds. Both of these examples are shown in Figure 12. The first example shows a correct prediction where MOTOREASE correctly identifies two icons on the screen and determines that they are not a minimum of 8 pixels in distance. However, Figure 12 also has an example of a false positive prediction which shows an overlap of two elements on the screen. Due to the visual bounds of each overlapping, the distance between the two is 0, therefore predicting a false positive.

4.4.5 RQ_5: False Positives and Negatives

The confusion matrices for the detectors are shown in Figure 13, where green boxes illustrate predictions that matched the ground truth, and red boxes illustrate predictions that did not match the ground-truth. These figures provide a visual representation of false positive and negative rates.

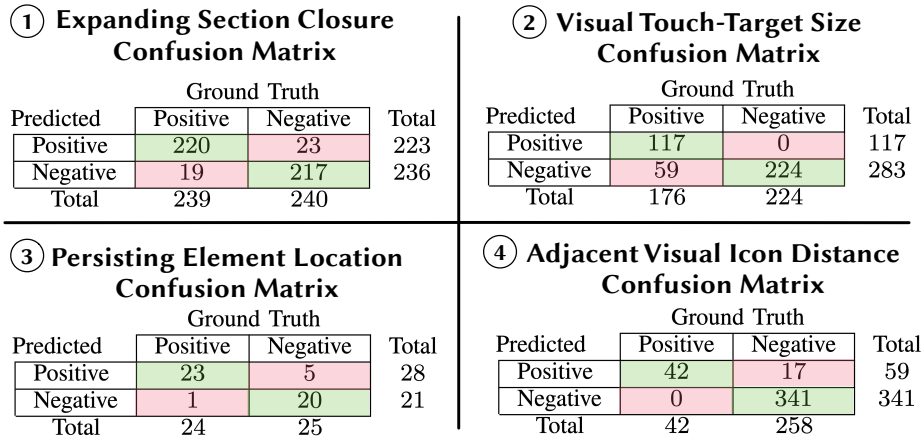

Figure 13: Detector Confusion Matrices

Figure 13-② illustrates that the visual touch target detector never produced a false-positive outcome. This is due to the fact that the detector specifically extracts elements labeled as clickable in the `XML`, therefore once they are extracted and have their edges analyzed, the detector is able to detect violations with certainty. The confusion matrix for the expanding section detector is shown in Figure 13-①, there were 23 false positive predictions, mainly due to limitations related to lexical pattern matching. Finally, MOTOREASE's persisting element detector identified only 8 false positives, mainly due to inconsistencies in matching elements across screens due to unexpected changes in `uiautomator XML` files. In evaluating the effectiveness of MOTOREASE, we also considered the impact of false negative predictions, as they signal violations that are not flagged by MotorEase, and hence could reach end-users. However, our evaluation revealed that the recall rate of MOTOREASE was relatively high, indicating that it is a dependable tool with a high true positive rate, thus demonstrating its practical applicability. In regards to MOTOREASE's low false negative prediction rate, The confusion matrix analysis for the icon distance detector shown in Figure 13-④ shows that MOTOREASE predicted 0 false negatives. Moreover, the confusion matrix analysis for the persisting elements detector shown in Figure 13-③ showed only one false negative prediction. Similarly, the confusion matrix for the expanding section detector presented in Figure 13-① demonstrated a false negative rate similar to its false positive rate, further supporting the viability of MotorEase. Overall, our evaluation demonstrates that MOTOREASE is likely a generally practical tool, exhibiting a relatively low rate of false positives and negatives. MOTOREASE leverages

it's ability to visually comprehend the visual and textual contents of a screen to determine accessibility violations. This approach offers distinct advantages. For instance, MOTOREASE showcases its adaptability by effectively detecting accessibility guideline violations regardless of variations in UI design or format. In contrast, traditional metadata-based approaches might be limited in their ability to analyze physical elements on the screen.

4.5 Accessibility Developer Tools Related Works

This section will look at previous accessibility related research. Our approach introduces a novel way to determine motor accessibility guideline violations within apps. In our exploration for previous work, we found that motor impaired users were seldom the targeted demographic in accessibility testing research. We also discovered that testing for accessibility based on guidelines is a relatively young area of research with many open problems. However, previous work has been done in accessibility testing and is presented below.

4.5.1 Accessibility Studies on Mobile Apps

There exists a large body of work that aims to understand how users with disabilities use their devices, and the potential accessibility issues that exist in current software applications [92, 29, 139, 55, 124, 91, 28, 128, 108, 96]. Such studies tend to take two forms, user studies [28, 91] and empirical analyses of software [55, 139]. Blind and low-vision users, like a majority of the population, have found themselves depending on their smartphones or touch screen devices recently [75].Touch devices have improved the life of people with disabilities, but some applications are still inaccessible to LV users [110]. A study by Park et al. [110] examined mobile application accessibility through a user study with four participants with an average of 1.9 years of experience with smartphones running IOS. The study aimed to produced a list of design guidelines for app developers to better support LV users some of which include text associations to UI elements, press action support (which is a way to make every interaction with a tap and no gestures), and under 30 items in a page [110]. This study successfully produced a list of ten guidelines that the participants had the most issues with. The previous study looked at accessibility from an IOS standpoint and looked to create a set of guidelines for LV users [91, 31, 68]. Alshayban et al. [31] examined the current state of accessibility issues in Android applications [31] using a Google-provided accessibility testing framework [17],they found that Text Contrast, Touch Target, Image Contrast, and Speakable Text are the most frequent accessibility issues [31].

Vendome et al. [133] also examined the prevalence of accessibility issues in Android apps. In this study, the authors mined thousands of android applications and analyzed the usage of accessibility APIs and whether or not applications adhered to accepted guidelines. In addition, they mined thousands of messages on Stack Overflow and other interaction platforms to understand the sentiment of developers and the types of questions they were asking. They found that most accessibility based conversations were centered around LV features while a lesser number focused on DHH features. Surprisingly though, they found that accessibility APIs imported into the programs were used as a way to resolve non-accessibility issues such as, retrieving notifications from other apps or automating touch interactions in the device [133].

Our approach is motivated by and complements what researchers have discovered in the above studies. These studies have shown the importance of the problem that MOTOREASE tackles

through illustrating the widespread prevalence of accessibility issues in mobile apps and illustrating a comparative lack of awareness of motor-impairment design guidelines and the needs of such users. Hence, these studies both motivate and validate our work on MOTOREASE.

4.5.2 Accessibility Testing

Software testing for accessibility aids developers in identifying violations of guidelines set forth by companies and governments [106, 17, 1, 110]. A wide range of research has been carried out to automate this process [117, 44, 122, 37, 106, 59, 123]. We discuss the most closely related approaches below.

Eler et.al. introduced the MATE tool [59] that uses automated dynamic analysis to check for accessibility issues that affect users with visual impairments in mobile apps, and generate detailed reports that facilitate developers fixing identified issues. Similar to MATE, MOTOREASE also leverages automated dynamic analysis and is able to generate reports that aid developers in fixing accessibility issues. However, our tool is differentiated by the ML-based analyses employed, and by it's focus on motor-impaired users.

Latte [122] is an accessibility testing framework introduced by Salehnamadi et.al. for android applications that aims to provide a deeper analysis compared to testing frameworks provided by Google [10, 17] by testing how accessibility services, such as VoiceOver, function in conjunction with feature-based use cases. The authors carried out an evaluation of their tool using the `switchAccess` and `TalkBack` services [1, 17] and found that several applications did not accommodate for both forms of accessible interactions. Latte is one of the only tools or studies that explicitly considers accessibility issues for users with motor impairments, given that it is capable of integrating with the `switchAccess` service in Android. However, given Latte's use case driven nature it both (1) requires pre-existing test cases, which many mobile apps have been shown to lack [79], and (ii) cannot detect violations of the specific accessibility guidelines targeted by MOTOREASE, as it attempts to carry out actions of a use case using an assistive service, and does not analyze the UI for specific patterns. In short, MOTOREASE and Latte serve largely *complementary* purposes, that is MOTOREASE provides UI design guidance to developers to avoid common pitfalls related to motor-impaired accessibility issues, and Latte can point out issues specific to given use cases and accessibility services.

AXERAY [37] provides an accessibility testing tool that tests for accessibility design violations in web applications identified by prior work [75, 128, 29]. This tool infers a semantic connection between different elements on the screen and attempts to group them together, then identifies whether the `HTML`/`CSS` markup on the page matches the semantic grouping. While MOTOREASE shares some elements of the visual and textual analyses employed by AXERAY, it does so to target the detection of specific guideline violations, as opposed to forming semantic groups of UI elements. Furthermore, MOTOREASE is focused on analyzing mobile apps as opposed to web apps, and is further focused specifically on accessibility issues that affect motor impaired users. We believe future work could translate techniques in this paper to the web domain.

Chen et al. [51] introduced Xbot, an accessibility testing tool that is capable of identifying accessibility issues within an app using a combination of dynamic and static program analysis. Xbot is not able to uncover any of the accessibility issues targeted by MOTOREASE. MOTOREASE exhibits novelty as compared to Xbot as it utilizes semantic understanding of the visual and textual elements of UI screens to detect new issues that affect motor-impaired users.

Finally, recently Salehnamadi et.al introduced the Groundhog approach [123], which is an accessibility crawler for mobile apps. Groundhog implements an automated UI crawler that explores an app both with and without assistive services, such as `TalkBack`, and notes any cases where an action can be performed through traditional touches, but cannot be performed via an assistive service. Again, MOTOREASE is largely *complementary* to this work, in that Groundhog targets general issues related to actionability and locatability of UI elements more broadly, but does not target the specific motor-impairment accessibility issues addressed by MOTOREASE. In fact, MOTOREASE aims to address two specific classes of issues identified in the Groundhog paper as being important for future work, (i) counterintuitive navigation (e.g., persisting elements) and (ii) inoperative actions (e.g., expanding sections).

4.5.3 Accessibility-Based UI Comprehension

Given that users interact with software through a UI, and past work has illustrated accessibility issues present in UIs [122, 75, 38, 113], there is a body of work dedicated to automatically comprehending and augmenting UIs to identify and circumvent accessibility and UI issues [84, 85, 47, 62, 127, 95, 145, 99, 102].

UI elements and icons typically need to be labeled in order for screen readers to be able to properly describe their appearance and functionality, however, such metadata is often missing from apps [49]. Zhang et.al.[145] and Chen et.al. [49] designed machine learning models trained from both existing UI labels and annotated labels from developers. Follow-up work by Mehralian et. al. introduced COALA [93], which aimed to improve upon the automated icon labeling by considering context related to screen text and region to build a multi-modal model for predicting icon labels.

Mansur et.al. introduced the AIDUI [89] tool, which uses semantic screen understanding to automatically identify and localize Dark Patterns in mobile app and web user interfaces. This technique uses similar techniques for semantic screen understanding as MOTOREASE, but in different ways. For instance, while AidUI uses element size and distance between elements as a factor in determining certain dark patterns, MOTOREASE aims to compare to the programmatic element size and visual element size to detect accessibility violations. Further, MOTOREASE analyzes *both* visual UI screenshots and `uiautomator` metadata, whereas AidUI operates only upon visual UI screenshots – with the use of multimodal data serving as a source of novelty for MOTOREASE.

Wu et al. [138] developed a screen parser that is able to automatically reverse-engineer the hierarchical layout of elements a given UI screenshot using purely computer vision techniques. Chen et.al. developed a technique that uses multi-modal neural networks to generate a UI skeleton for mobile apps [47]. Nguyen et.al. [104] and Moran et.al. [97] created approaches for automatically prototyping the UIs of applications using both supervised and unsupervised computer vision techniques to comprehend screen structure from pixels. While Wu et.al. illustrated that UI understanding and generation techniques could be used to bolster accessibility of mobile apps [138], these techniques target different accessibility issues as compared to MOTOREASE.

4.6 Summary

In this paper, we presented MOTOREASE, an approach for detecting, classifying, reporting motor-impairment accessibility violations. We measured the performance, generalizability, and applicability of MOTOREASE to various open source applications. Our results indicate that MOTOREASE

is effective in practice and offers a novel approach for developers to identify accessibility issues affecting motor-impaired users. Future work will examine the potential to detect accessibility issues in web apps and conduct user studies.

5 Structurally motivated UI Embedding

As society trends toward increasing digitization, large portions of individuals' daily lives are governed by software and the user interfaces (UIs) that drive them. As such, the design, programming, and adaptation of UIs has never been more important. However, the creation and construction of user interfaces is a challenging task due to the need to reason between the affordances offered by UIs and the underlying logic of software [**?**]. To help tame these challenges, the research community has increasingly invested in techniques for automatically and computationally understanding and adapting UIs [77, 35, 76]. These techniques underlie powerful new tools that enable new forms of programming [**?**], adaptions for accessibility [**?**], and refinement of UI design [89].

For example, VUT [77] and UIBert [35] are composed of an ensemble of multi-modal models that jointly encode screen text, visual pixel-based information, and structural information related to UI hierarchies. Other techniques, such as ScreenVec [76] encode purely textual information from both app descriptions and text from the screen, and UI hierarchy information. The main challenge that these techniques aim to solve is the following, *How can patterns unique to UI screens be learned such that they can support automated tasks for creating, understanding, and adapting user interfaces?*

The crux of the above challenge is adapting *generalized* models, which have been shown to effectively learn patterns across a varied range of data domains, to the specific domain of user interfaces. As such, the methods mentioned often *attempt* to capture rich contextual information conveyed through the visual elements of the screen. UI's are designed as graphical interfaces with specific layouts and visual properties that capture important semantic information about the affordances of the UI. That is, the structural, visual, and lexical properties of elements on the screen provide contextual information about each screen's function.

These visual elements are valuable for techniques like Erica's [58] self-supervised clustering method, which identifies similar icons across multiple apps, and Liu et al.'s [83] approach utilizing metadata and hierarchical UI structure to uncover structural patterns within interfaces.

Another underlying challenge in learning patterns from user interfaces is the *variability* in designs that convey similar semantic meanings. For example, even a screen as simple as a login screen, which is typically comprised of two text fields and a button, can be instantiated through a wide variety of different visual designs and lexicon. While current UI embedding techniques, such as VUT and UIBert aim to address these challenges through embedding UI hierarchy information, this information is often flattened into representations that make it difficult to deal with the large variety of semantically similar, yet characteristically different patterns that are found across UI designs.

However, recent research demonstrates the advantages of multi-modal embeddings such as BEiT [39] and CLIP [**?**] that use visual and visual-language pairs respectively. These two embeddings are capable of embedding any image regardless of whether it is a UI screen or not. Recent work has shown that CLIP is a strong performing image embedding because of its visual-language paired embedding method [135, 46, 30, 142]. CLIP has also established itself as an excellent em-

bedding in retrieval tasks [30, 116, 75]. This has led to many works to use CLIP as the preferred image embedding technique in UI tasks as well. However, CLIP lacks the ability to consider the structural composition of user interfaces (UIs) and the potential advantages this information could offer in screen similarity tasks.

To address these limitations in the existing research, we propose FRAME: A Rein**F**orced Use**R** Interf**A**ce Screen E**M**bedding With Graphical Structural Compr**E**hension. FRAME is a UI screen-specific embedding that enhances the existing CLIP embedding by incorporating structural information. FRAME operates through five main steps to achieve this. Initially, it preprocesses the input image by converting it to black-and-white and enhancing contrast. Next, it utilizes a computer vision edge detection model to extract UI elements and generate embeddings for them. These embeddings form a positional graph representation of the screen's elements, integrating text and visual embeddings. The embeddings within the graph nodes are propagated to effectively weigh graph node embeddings according to their proximity within the graph, revealing underlying relations. The propagation of these embeddings employs a method derived from the first-order approximation of localized spectral filters on graphs [69, 56]. Subsequently, a triangulation approach called the Rips Complex consolidates the augmented graph embedding into a unified embedding using triangles formed in the embedding space as weights. The Rips Complex is a mathematical concept that is known to preserve structural information of the graph within an embedding space. This structural embedding is then combined with the augmented input image to embed the screen's structural properties into the augmented CLIP embedding of the input image. Although the final embedding is noisy, FRAME mitigates this by applying PCA to reduce it to a size of 116. The resulting embedding encapsulates a comprehensive representation of the screen's structural characteristics, aiding in screen similarity and retrieval tasks.

5.1 Approach

FRAME is a structural-based approach to creating an image embedding that will aid in various screen similarity and retrieval tasks. To achieve this, FRAME requires only the screenshot of a mobile app screen to generate the embedding. This is done as a way to maximize the use cases for FRAME in any process that may require the embedding of a UI screen. FRAME operates in five main stages to create the most structurally accurate representation of the screen. When a screen is given to FRAME, it preprocesses the image using a series of image alteration techniques. We use the images and use edge detection to extract visual elements of the screen while retaining their positional information within the context of the screen. We create a graph-based representation of these elements and contextually weight the graph nodes based on their neighborhoods. Once that is done, FRAME will take the new node embeddings and consolidate them using the Vietoris-Rips complex to consolidate the graph information into one embedding. The process of creating this embedding is visualized in Figure 14. This embedding will be a structurally rich embedding which may aid in tasks requiring a numerical UI screen representation.

5.1.1 Pre Processing the Image

Screens are designed to be visually pleasing and provide an intuitive placement of the elements such as icons and text boxes. In order for the embedding to abstract the elements on the screen and mitigate noise on the screen, FRAME utilizes two screen preprocessing techniques. Noise on the

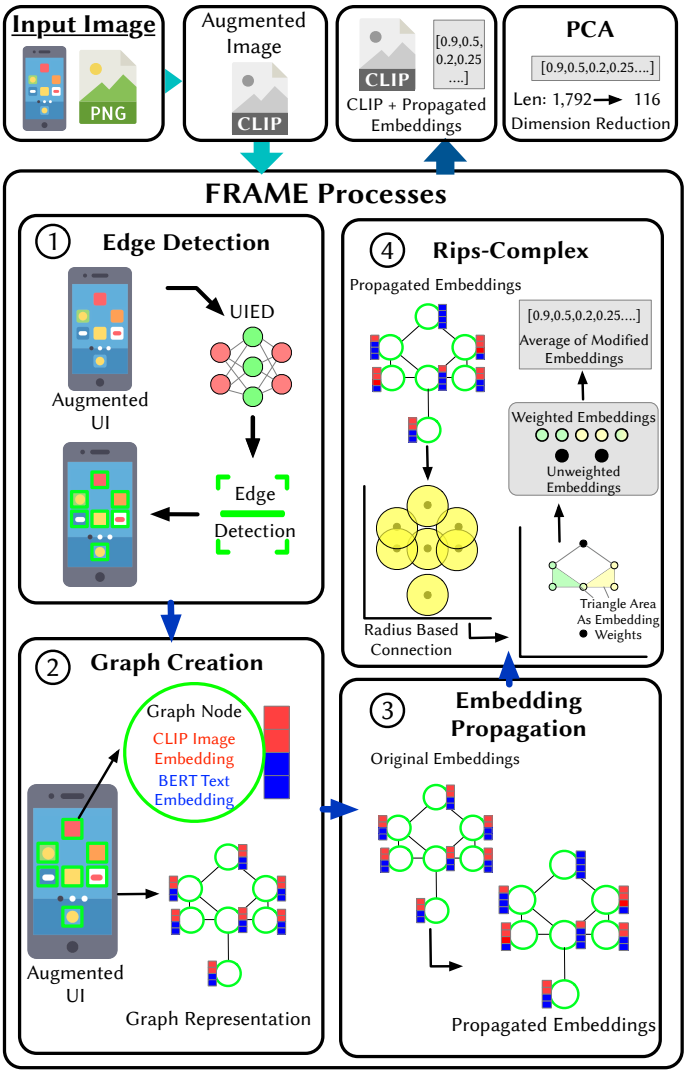

Figure 14: Overview of The FRAME Approach

screen can take many forms such as color palettes, customized icon designs, and different fonts. To mitigate these types of noise on the screen, FRAME uses black-and-white as well as high contrast alterations to the screen.

black-and-white Modification

FRAME's goal is to provide skeletal-based similarity of screens rather than color. For example, Figure 15 shows three images that were the most similar when using a CLIP embedding [**?**]. The three screenshots have very similar color palettes and dominating colors upon visual inspection. On top of being similarly colored, these images do not share the same structural components and are three different types of screens. The leftmost screen is a Splash screen, the middle is a Search screen, and the rightmost screen is a password screen. Visual intuition suggests that these three screens provide different functions, but the CLIP embedding, which considers the colors of the screen, is unable to properly identify this discrepancy in the visual structure of the elements on the

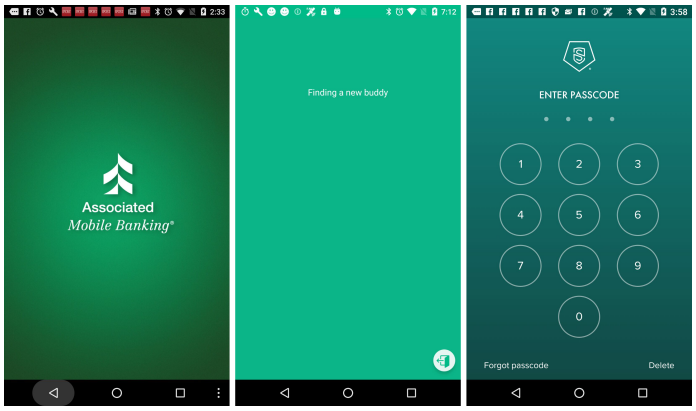

Figure 15: 3 Similar Images using CLIP Embeddings

screen. To diminish the identified issues experienced by CLIP, we opt to remove color as a factor from consideration. We use the Pillow library's [**?**] grey scale conversion to convert the image to black-and-white first to ensure that no color is present in the extraction of the screen elements in the process of making the embedding. This can be seen in Figure 16 as the input image is made black-and-white.

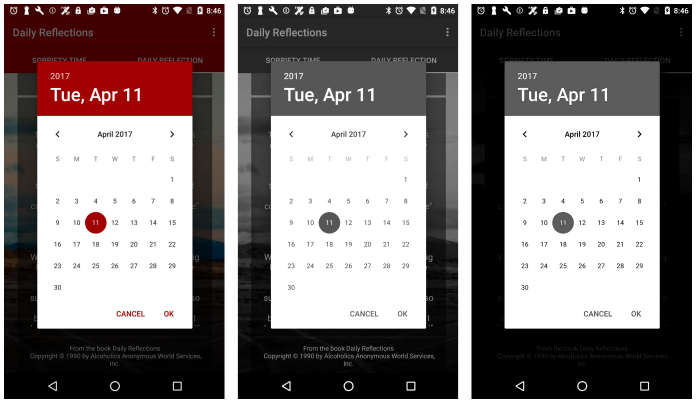

Figure 16: Preprocessing and augmenting the input image

High-Contrast

In addition to making each input screenshot black-and-white, we alter the images by increasing the contrast by 2X in the image. Contrast allows FRAME to darken the dark pixels in the screen and brighten the brighter pixels in the screen. The resulting image has two main characteristics: (i) enhanced edge visibility; (ii) reduced noise influence. Enhanced edges are created because the colors on either side of the edge are modified, resulting in a more obvious and harsh edge. This aids the edge detection algorithm in finding accurate edges to accurately represent the screen's elements. Reduced noise is achieved by merging shades that are very close to each other, such as drop shadows, to create an abstract screen without many distractions for the edge detection algorithm. We use the Pillow library's [**?**] "ImageEnhance" sublibrary which has a contrast function

to increase the contrast in the image. An example of this is shown in Figure 16. We can see that the initial black-and-white image has less pronounced shades of greyscale colors and edges, while the high-contrast image has very clear boundaries between light and dark which can help facilitate edge detection.

5.1.2 Graph Creation

To ingrain the visual structure of a UI screen, FRAME utilizes a graph to plot nodes in similar areas as they appear on the screen. These nodes are created by the individual elements of the screen and their positional information. This positional information combined with elemental embedding information provides for a rich node within the graph to define relations between elements within the graph neighborhoods. There are three stages to developing a graph for each screen.

Edge Detection

FRAME's goal is to create an embedding with the same intuitive structural pattern humans notice in their UIs. This is motivated by the examples of screens shown in Figure 15. We identify that passcode screens traditionally have the keypad in the center and that splash screens traditionally have the logo of the app in the center while surrounded by color. Being able to extract the icons and elements of the screen that suggest these patterns is a key element in creating an embedding that is capable of identifying these patterns. The edge detection approach used is an open-source edge detection model called UIED [103]. UIED is specialized to work with UI Screens and provides high-quality element detection. This approach is further aided by the preprocessed image. FRAME takes the edge detector's output and crops out each image. FRAME only considers images larger than 48x48px since that is the standard for minimum accessible icons on an Android screen as per Google's Accessibility Guidelines [17]. UIED gives the bounding box for each element as well, which can be used to find the centralized (x,y) point where the element is located on the screen. This is the positional information for each extracted element on the screen.

Graph Creation

Icons on the screen have contextual information to offer to a screen. For example, a magnifying glass icon can indicate search, or a gear wheel icon may suggest settings. In addition to the positional information obtained through UIED, FRAME takes into account the visual and textual aspects of each icon on the screen. To do this, we take the cropped icon provided by UIED and create 2 embeddings from it, an image embedding using CLIP, and a text embedding using BERT [**?**].

Image Embedding: Though CLIP may not capture structure that well, CLIP is excellent at identifying stylistically similar screens as shown in Figure 15. We use this property of CLIP to create embeddings for each icon on the screen. We create the icon embedding and add it to the node in the graph that represents the element on the screen. As shown in Figure 17, each node has a visual representation of the icon that is cropped out of the screen.

Text Embedding: BERT is currently the most advanced text embedding available and we utilized it to create textual embeddings for each icon. Icons may have text in them, for example, the

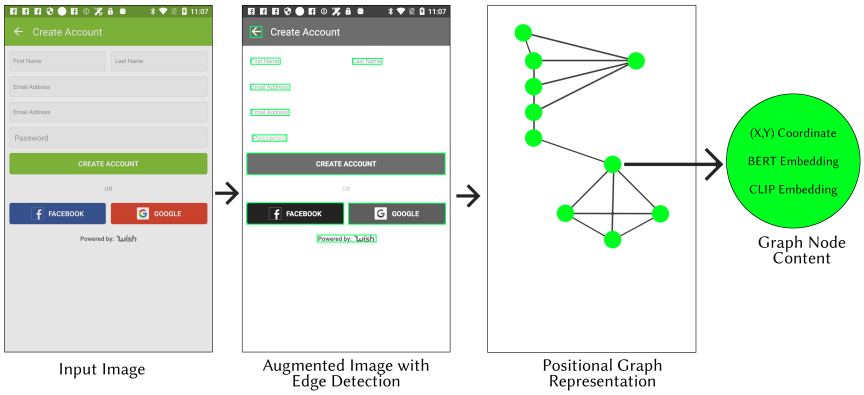

Figure 17: Process of creating the graph

Facebook icon has the letter ”F” and the word ”Facebook” under the logo. We can leverage this text to add more depth to each node within the graph. We use PyTesseract to extract text from each cropped icon. We take the extracted text and clean it using regex functions to make it clean text. FRAME takes the extracted, cleaned text and creates the BERT embedding to add to the graph node. As shown in Figure 17, each node has a textual representation of the icon that is cropped out of the screen. In the case there is no text, a single space character is used as default.

With the inclusion of the image and text embeddings within each node as well as the (x,y) positional coordinate for each icon, the nodes in the graph are filled with niche data that is localized to each icon. This gives each node a unique set of features that sets it apart from other nodes within the graph and its neighborhoods. We can utilize this diversity of node information to create a graph that can map relations between nodes based on their embedding data.

Graph Edges: Once nodes for the graph are created, FRAME takes into account the positional distance between the nodes in the graph and connects nodes that are within a certain distance to each other on the screen. The graph connects nodes that are a maximum of 300 pixels in distance from each other when using Manhattan distance. On average, the distance between icons on the screens is 300 pixels as observed by the authors. Therefore, we elected to use the average distance as the threshold for edge connection to ensure that the resulting graph does not lack edges or appear overly complete. This approach allows for the formation of distinct neighborhood clusters within the graph. The graph is built with nodes that have position coordinates, BERT embedding, and CLIP embedding and are connected via edges to form neighborhoods. This is shown in Figure 17. FRAME aims to find patterns between close elements, so having the connections be a maximum distance of 300 pixels allowed the graph to have distinct neighborhoods of nodes on different parts of the screen. This helps localize icon clusters and elements on the screen that may closely be related to each other.

5.1.3 Graph Embedding Propagation

Unlike ordinary images, GUI screens exhibit a more structural nature, with closer relationships existing between adjacent components. Therefore, we augment the image/text embedding of each component by its neighbor component embeddings based on the constructed structural graph. This is inspired by a recent work for code embedding augmentation by leveraging the program depen-

dence (i.e. structural) graph constructed for a software system [140]. Specifically, our embedding propagation strategy is derived from the first-order approximation of localized spectral filters on graphs [69, 56], which can be represented as follows:

$$S' = (I_N + wD^{-\frac{1}{2}}AD^{-\frac{1}{2}})S. \tag{1}$$

$S \in \mathbb{R}^{N \times M}$ represents the matrix of all the image/text embeddings of nodes from G and S' represents the updated matrix by incorporating the information from the neighbor nodes. M denotes the dimension of each image/text embedding (i.e. 768) and N denotes the number of nodes in the graph. A is the adjacency matrix of G without self-connections and D is the degree matrix of A so that the adjacency matrix is normalized by D with respect to both the row and the column. w is a constant to balance the information from the original node/component with structural information from the neighbor nodes/components. By leveraging the embedding propagation strategy, the image/text embedding for each component will encompass more structural information from its adjacent components, which is expected to further enhance GUI screen understanding. A visual representation of this process is shown in Figure 14-3.

5.1.4 Rips-Complex

We realize the propagated UI embeddings as a set of points in Euclidean space to capture information regarding the topological relationship of the embedded data. This is achieved by triangulating the points to create a structure known as a simplicial complex as shown in Figure 18.

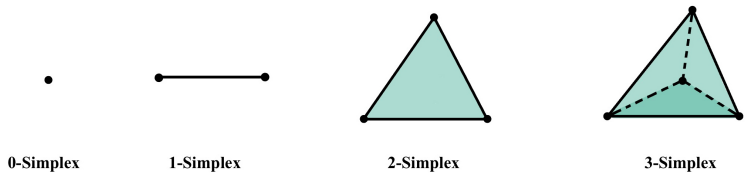

Figure 18: Visualization of low dimensional simplicies

A simplicial complex is created by plotting the propagated embeddings on a graph in Euclidian space. We use our embeddings to create an object known as the Vietoris-Rips complex (often called the Rips complex) using GUHDI [**?**]. Given a parameter $\epsilon > 0$, the Rips complex of embeddings $\mathcal{X}$ is made into a simplicial complex that captures the spatial and topological relationships between the points in space using the following definition:

Definition 1. *Given a set of embeddings $\mathcal{X} = \{x_1, x_2, \ldots, x_k\} \subset \mathbb{R}^n$ and an $\epsilon > 0$, a k-simplex $\sigma = [x_{i_1}, x_{i_2}, \ldots x_{i_k}]$ is in the Vietoris-Rips Complex Rips$_\epsilon(\mathcal{X})$ if and only if:*

$$\mathbb{B}_\epsilon(x_{i_j}) \cap \mathbb{B}_\epsilon(x_{i_{j'}}) \neq \emptyset$$

where $\mathbb{B}_\epsilon(x_{i_j})$ is an open ball at x_{i_j}.

$\epsilon > 0$ is the radius of the open ball used to determine edge connections within the embedding space. For small parameters of ϵ, the open balls are too small to intersect and result in a complex with only the original embeddings. On the opposite end, if ϵ is too large, every feature in the input image is related to every other feature. We selected ϵ by performing a parameter study from

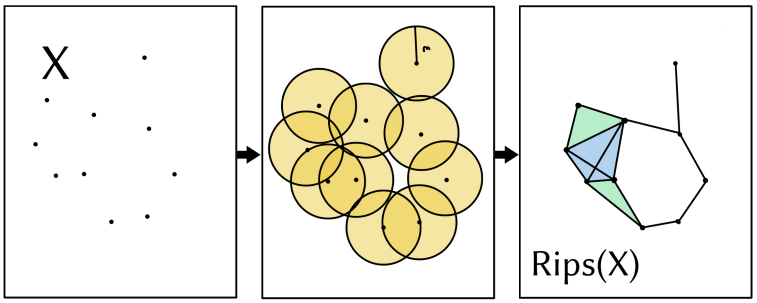

Figure 19: Process of creating the Rips Complex using a set of embeddings

$0.1 - 1$ in increments of 0.05. Our ideal value of ϵ captures the average 100-200 features present in an input image. This was chosen to be $\epsilon = 0.5$, as it resulted in an average of 100-200 2-simplicies generated in the Rips complex. The process of overlapping open balls and simplex creation is shown in Figure 19.

By employing the Rips complex on the propagated UI embeddings we are topologically capturing the relationship between an object in an image with all other objects in the image as shown in Figure 19. Once the simplex is created, we consider the embeddings that make 2-simplicies (2D triangles) and use the area of the resulting simplex as a weight for the involved embeddings. The area of a given 2-simplex can be calculated using Heron's formula [136]:

$$\text{Area} = \sqrt{S(S-A)(S-B)(S-C)} \tag{2}$$

where S is the semi-perimeter of the 2-simplex and A, B, and C are the lengths of its three edges. The area is then used to weigh the involved embeddings.

We employ the Area-based Triangulated Embedding method (ATE), proposed by Krishna Vajjala et al. [72], for the weighted average of the embeddings. This creates a weighted relation between embeddings that are close to each other, resulting in a weighted preservation of information between those three embeddings. We do this until we have a set of weighted and unweighted embeddings. We then average these embeddings together into one final embedding. A visual representation of this process is shown in Figure 14-4.

5.1.5 Final Embedding

FRAME embeddings reinforce existing embeddings by adding structural understanding to an image. The process of creating a distance-similar graph, propagating the embeddings within it, and consolidating them using the area-based triangulation creates a structure-encoded set of embeddings. This set of embeddings can then be added to the initial pre-processed CLIP image embedding. This results in an embedding that has the pre-processed version of the CLIP embedding along with the propagates structural relations between the icons on the screen which are a CLIP and BERT embedding combined. The resulting embedding is noisy and large with a size of 1,792 dimensions. To combat this, FRAME uses PCA [**?**], a dimension reduction technique to reduce noise in high-dimension vectors. PCA takes a list of vectors and identifies the directions (principal components) where the data varies the most. By projecting the original data onto these principal components, PCA effectively reduces the dimensionality of the dataset while preserving as much

of the original information as possible through linear transformations. We use Google's Scikit-Learn's [**?**] PCA which takes as input a series of vectors and the desired size of the vector. We considered the training data vectors from both of our datasets and ran a hyperparameter study to determine the best size for the resulting vector. The resulting vector size is reduced from 1,792 down to 116. To ensure that new embeddings result in size 116 without re-running PCA on the dataset, we use PCA's transformation matrix and do a dot product with the initial noisy vector and the transformation matrix to obtain the 116 size representation of the new embedding.

5.2 Evaluation Methodology

In this section, we describe the procedure we used to evaluate FRAME. To achieve our study goals, we formulated the following four research questions:

- **RQ**$_1$ *How does* FRAME *perform against other baselines in screen retrieval tasks*
- **RQ**$_2$ *How does* FRAME *perform on various types screens against the best baseline?*
- **RQ**$_3$ *How important is the structural embedding propagation between graphs in order enhance the embedding ?*
- **RQ**$_4$ *How well does* FRAME *leverage its ability to abstract the screen and disregard styling?*

5.2.1 FRAME Evaluation Database

FRAME is designed to understand structural aspects of the screen. Screen retrieval tasks are a suitable benchmark for FRAME since the embedding will need to effectively identify other screens with similar visual layouts regardless of colors and text differences. This means that the evaluation for FRAME has to focus on screen similarity and retrieval tasks that require FRAME to find screens labeled of the same category.

We evaluate FRAME with two different databases, Avgust [147] and Aurora [67].

Aurora Dataset

While creating the Aurora tool [67], authors created a labeled version of the RICO dataset with a small subset of the images. The RICO dataset is a set of 66,000 unique UI screenshots [**?**]. However, this dataset is just a set of screens and they are not labeled. The labeled Aurora dataset was made by selecting a 1% subset of RICO UI images randomly. Then they performed an unlabeled clustering of the images based on their visual properties. They used k-means clustering and clustered them into groups. The authors manually examined the clusters and collectively agreed to label them into 21 distinct categories. This process resulted in a high-quality dataset to evaluate screen retrieval and search algorithms. This dataset has 21 distinct categories and 1370 screens. It is a sparse dataset and can test FRAME's ability to find similar screens in a dataset with lots of variability. We divide this dataset using a 90 to 10 train-test split for testing. The resulting evaluation dataset is 1233 train images and 137 test images.

Avgust Dataset
The Avgust dataset is another subset of the RICO dataset. This dataset is obtained from the Avgust paper by Zhao et al. [146]. It has 25 labels with 2475 total screens. This dataset was manually labeled by 4 authors who mutually agreed on each screen's category. This is a dense daset with above 130 screens per category on average. For testing, we divide this dataset using a 90 to 10 train-test split. The resulting evaluation dataset is 2219 train images and 256 test images.

5.2.2 RQ$_1$ *How does* FRAME *perform against other baselines in screen retrieval tasks?*

To evaluate FRAME, we use two common information retrieval metrics to test its ability to find similar screens properly; we use Hits@k and Mean Reciprocal Rank (MRR).

Hits@k

Hits@K is a metric that evaluates the effectiveness of information retrieval systems by measuring how many relevant items appear within the top K results. It assesses the tool's ability to provide relevant content within a specified top K number of ranked items. For example, Hits@10 would quantify the percentage of relevant items found within the top 10 results. This metric is beneficial for evaluating search algorithms and retrieval tasks, as it prioritizes the most relevant results users will likely encounter.

$$\text{Hits@}k = \frac{\text{Number of relevant items in top } k \text{ results}}{k} \tag{3}$$

We perform Hits@k with three different K values, 1, 5, and 10. When using Hits@K, a higher K value can make for a more lenient evaluation, having smaller numbers results in a rigid evaluation of the embedding that can provide true insight into its performance. In our assessment of FRAME, we input a test image labeled with a specific category, then examine the top K images most similar to the test image to determine how many of them share the same label. This gives us an accurate representation of how FRAME is able to find similarly structured screens. We perform this evaluation on both the Aurora and the Avgust datasets to test its ability to find similar screens in sparse and dense datasets.

Mean Reciprocal Rank (MRR)

Mean Reciprocal Rank (MRR) is a metric commonly used in information retrieval and ranking tasks. It calculates the average of the reciprocals of the ranks at which the relevant items are retrieved. For instance, if a relevant item is found at rank 3, its reciprocal would be $\frac{1}{3}$. If another relevant item is found at rank 5, its reciprocal would be $\frac{1}{5}$. The MRR is then calculated as the average of these reciprocals.

$$\text{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i} \tag{4}$$

MRR provides a single numerical value that summarizes the performance of the system across multiple queries, making it useful for evaluating and comparing different ranking algorithms or

search models. In our assessment of FRAME, we input a test image labeled with a specific category, then examine the list of images most similar to the test image to find the first image with the same label as the test image. This evaluation provides insight into how well FRAME identifies similar images as higher in the list of similar images. We perform this evaluation on both the Aurora and the Avgust datasets to test its ability to find similar screens in sparse and dense datasets.

It is important to understand how well FRAME performs on its own, but having it perform the same tasks as other, common, baselines provides an insight into how well FRAME is able to perform in information retrieval tasks. We take the same test and train data from both of our datasets and perform the same, rigorous, evaluation on three baselines to measure FRAME's performance in comparison to the baselines. We compare FRAME to three key baselines: CLIP, Screen2Vec, and BERT. We use these baselines to measure FRAME's efficacy compared to popular tools.

CLIP

OpenAI's CLIP embeddings [**?**] are currently the most popular embedding for image-related tasks. CLIP embeddings are capable of creating an embedding for any image regardless of it being a screen or not. However, CLIP has emerged as the prominent embedding given that they can represent both images and text in a shared embedding space, enabling cross-modal understanding. FRAME takes into account the visual and textual aspects of the screen as well, so having CLIP embeddings given their popularity and their construction is a valuable baseline to FRAME.

Screen2Vec

Li et al.'s Screen2Vec [76] is designed as an embedding specifically for screens. This serves as a valuable baseline in evaluating the performance of FRAME because Screen2Vec is designed to embed UI screens and so is FRAME. This baseline allows us to test FRAME against another screen-specific embedding.

BERT

Google's BERT embeddings [**?**] serve as a good baseline for FRAME because it is a widely-used and well-understood model for natural language understanding tasks. Despite being primarily designed for text processing, BERT can still encode textual information about images, providing a simple and readily available baseline for cross-modal tasks.

We use these three baselines and the same information retrieval metrics presented above to evaluate FRAME's performance in comparison to the baselines.

5.2.3 RQ_2 *How does* FRAME *perform on various types of screens against the best baseline?*

To determine FRAME's ability to detect various screens, we present a case study aimed at evaluating the performance of FRAME across various types of screens, comparing its performance against the best baseline. The study focuses on different screen functionalities such as login, search, etc., to provide a comprehensive understanding of FRAME's efficacy in diverse contexts.

To conduct the case study, we identify the best baseline against which to compare FRAME's performance. This baseline is chosen based on the most similar performance to FRAME when considering the metrics presented in Section 5.2.2, considering its robustness and reliability in similar contexts. Additionally, we implemented both FRAME and the best baseline in the same test and train splits across both Avgust and Aurora datasets to ensure there is a thorough, fair comparison.

5.2.4 RQ_3 *How important is the structural embedding propagation between graphs in order enhance the embedding ?*

Embedding propagation is the process of modifying embeddings within a neighborhood to weigh nodes differently. This process aids in weighting embeddings based on their structural proximity. This process provides the structural-based enhancement to the base CLIP embedding. To determine FRAME's ability to structurally enhance the base embedding, it is important to consider FRAME's performance with and without its embedding propagation. There are six variants of embedding propagation that we test against the final variant of FRAME. In the interest of space and simplicity, we have elected to create codes for each variant of FRAME. The structure is as follows: C-PB-PC. This indicates the augmented CLIP, the propagated BERT, and the propagated CLIP embeddings are all present within the variant. To omit an embedding to create a variant, we add an **'N'** next to the embedding. For example: NC-PB-PC indicates that the augmented CLIP embedding is **not** present, but the propagated BERT and CLIP embeddings are present. Below are the descriptions of each variant along with their codes for reference.

Propagated Embedding: NC-PB-PC

The propagated embedding has both the CLIP and BERT propagated embeddings that are concatenated with the original modified CLIP embedding. This set of concatenated propagated embeddings provides the foundation to enhance the modified CLIP embedding with structural properties. To demonstrate the need for propagated embedding in the final embedding, we run the same set of tests that the FRAME embedding is evaluated with, but we omit the augmented CLIP embedding and only leave the propagated embedding. In return, this will demonstrate the performance impact that the CLIP has within the FRAME embedding.

CLIP + No Propagated CLIP: C-PB-NPC

The propagated embedding has both the CLIP and BERT propagated embeddings that are concatenated with the original modified CLIP embedding. The propagated CLIP embedding is a weighted CLIP embedding that is created by the propagation between graph nodes within a neighborhood. This relationship is crucial to adding image-based structural properties to the image since the approach weighs images that are closer to each other. To demonstrate the need for propagated CLIP embeddings in the final embedding, we run the same set of tests that the FRAME embedding is evaluated with, but we omit the propagated CLIP embedding and leave in propagated BERT embedding and augmented CLIP embedding. In return, this will demonstrate the performance impact that CLIP embedding propagation has within the FRAME embedding.

CLIP + No Propagated BERT: C-NPB-PC

The propagated embedding has both the CLIP and BERT propagated embeddings that are concatenated with the original modified CLIP embedding. The propagated BERT embedding is a weighted BERT embedding that is created by the propagation between graph nodes within a neighborhood. This relationship is crucial to adding textual-based structural properties to the image since the approach weighs text within icons that are closer to each other. To demonstrate the need for propagated BERT embeddings in the final embedding, we run the same set of tests that the FRAME embedding is evaluated with, but we omit the propagated BERT embedding and leave in propagated CLIP embedding and augmented CLIP embedding. In return, this will demonstrate the performance impact that BERT embedding propagation has within the FRAME embedding.

CLIP + No Propagated Embeddings: C-NPB-NPC

The propagated embedding has both the CLIP and BERT propagated embeddings that are concatenated with the original modified CLIP embedding. This set of concatenated propagated embeddings provides the foundation to enhance the modified CLIP embedding with structural properties. To demonstrate the need for propagated embedding in the final embedding, we run the same set of tests that the FRAME embedding is evaluated with, but we omit the propagated embeddings and only leave the augmented CLIP embedding. In return, this will demonstrate the performance impact that the propagated embedding has within the FRAME embedding.

Only Propagated BERT: NC-PB-NPC

Propagated embeddings carry intrinsic spatial data with weights depending on their proximity to other elements on the screen. However, it is important to consider the performance of the propagation itself. We do this by running the same tests that the FRAME embedding is evaluated with, but only consider the propagated BERT embedding.

Only Propagated CLIP: NC-NPB-PC

Propagated embeddings carry intrinsic spatial data with weights depending on their proximity to other elements on the screen. However, it is important to consider the performance of the propagation itself. We do this by running the same tests that the FRAME embedding is evaluated with, but only consider the propagated CLIP embedding.

5.2.5 RQ_4 *How well does* FRAME *leverage its ability to abstract the screen and disregard styling?*

To determine FRAME's ability to disregard stylistic elements in a UI, it is important to consider FRAME's performance with and without its various screen augmentations. FRAME performs a black-and-white augmentation first and then adds on a high contrast augmentation to remove color bias in the screen and create more defined objects on the screen. To test the efficacy of these techniques, we run an augmentation study to determine how much each augmentation impacts the

resulting performance for FRAME. We do this by testing three different variants of FRAME. In the interest of space and simplicity, we have elected to create codes for each variant of FRAME in the augmentation study. The structure is as follows: B-C. This indicates the augmented input image has both black-and-white and contrast augmentations. To omit an augmentation to create a variant, we add an **'N'** next to the augmentation. For example: B-NC indicates that the augmented input image has a black-and-white augmentation but the high contrast augmentation is **not** present. Below are the descriptions of each variant along with their codes for reference.

No Contrast: B-NC

FRAME uses contrast enhancement to aid the computer vision in identifying elements on the screen. This is necessary to avoid extracting elements that could be slight color variations or small on the screen. This gives FRAME the highest quality elements to use in its graph and reduces the number of unrelated elements in the propagation. To demonstrate the need for contrast enhancements in the image, we run the same set of tests that the FRAME embedding is evaluated with, but we omit the contrast augmentation and leave in the other augmentations to the embedding including the embedding propagation and the black-and-white augmentation. In return, this will demonstrate the performance impact that contrast provides within the FRAME embedding.

No black-and-white: NB-C

FRAME converts the input image to black-a nd-white to remove a color bias in the similarity measure. This is important since it eliminates the ability for the embedding to make similar embeddings solely off color and allows FRAME to enhance the embedding with structural properties. To demonstrate the need for black-and-white enhancements to the image, we run the same set of tests that the FRAME embedding is evaluated with, but we omit the black-and-white conversion and leave in the other augmentations to the embedding including the embedding propagation and the contrast enhancements. In return, this will demonstrate the performance impact that black-and-white provides within the FRAME embedding. Thus proving the motivation that color-independent screen similarity tasks are structurally influenced.

No Contrast and No black-and-white: NB-NC

To demonstrate the need for augmentation, it is important to consider the performance of the embedding without any augmentation. FRAME embeddings are motivated by the idea that the embedding should have no color and style bias, unlike popular image embeddings. To do this, we built FRAME to remove colors from the image being processed. This intuition serves well for the idea behind FRAME, but it is important to show that this intuition results in higher performance. To demonstrate that these augmentations are necessary, we run the same set of tests that the FRAME embedding is evaluated with, but we omit all initial enhancements while retaining the embedding propagation. In return, this will demonstrate the performance impact that the preprocessing steps provide within the FRAME embedding. Thus strengthening the motivation behind the augmentation decisions.

5.3 Evaluation Results

This section analyzes the results of our empirical evaluation of FRAME.

5.3.1 RQ_1 *How does* FRAME *perform against other baselines in screen retrieval tasks?*

Dataset	Metrics	Screen2Vec	BERT	CLIP	FRAME
Aurora	HR@1	0.3750	0.3194	0.4722	**0.5625***
	HR@5	0.3027	0.2486	0.4013	**0.4652***
	HR@10	0.2604	0.2187	0.3590	**0.4166***
	MRR	0.4853	0.4363	0.5963	**0.6729***
Avgust	HR@1	0.3922	0.8270	0.8549	**0.8784***
	HR@5	0.3185	0.7788	0.7788	**0.8266***
	HR@10	0.2739	0.7141	0.7364	**0.7799***
	MRR	0.5116	0.8908	0.8913	**0.9138***

Table 7: Results over 2 datasets, where the best results are in **bold**, and * indicates a p-value less than 0.05 from two-tailed paired t-test between FRAME and best baseline (CLIP).

FRAME performs significantly better in screen retrieval tasks compared to the state-of-the-art baseline methods. Specifically, FRAME outperforms Screen2Vec, BERT, and CLIP in every metric for both datasets. It is important to note that Screen2Vec is designed as an embedding for UI screens, BERT is used in image embeddings by extracting the text on the screen to identify semantic similarities between screens, and CLIP is the leading image embedding method that leverages textual and visual properties of images. From the results in Table 7, we can see that, compared to the best baseline, FRAME shows an average of 15.9% improvement on the Aurora dataset and an average of 4.3% improvement on the Avgust dataset. The Aurora dataset contains fewer images and is more sparse overall, making the screen retrieval task difficult. On the other hand, the Avgust dataset is very dense and contains a large number of screens. Given that the two datasets are very different in nature, FRAME is able to show superior performance in screen retrieval tasks across both datasets along with statistically significant results. In addition, for the Aurora dataset, the MRR improvement between FRAME and the best baseline is roughly 12.8% and the improvement between FRAME and the best baseline for the Avgust dataset is 2.5%. The MRR metric gives higher scores to screens that appear closer to the top of a ranked list. Based on these results, we show FRAME's impressive ability to rank similar UI screens towards the top of the ranked list.

5.3.2 RQ_2 *How does* FRAME *perform on various types of screens against the best baseline?*

It is important to evaluate FRAME on fine grained datapoints in addition to the averaged metrics. We evaluated FRAME and its best baseline, CLIP, to gain a finer understanding of their performance on various screen types. We present a head-to-head comparison against the two embeddings using Hits@10. Hits@10 strikes a balance between precision (the proportion of relevant results among the retrieved items) and recall (the proportion of relevant results that are successfully retrieved).

Dataset	Metrics	NC-NPB-PC	NC-PB-NPC	C-NPB-NPC	NC-PB-PC	C-PB-NPC	C-NPB-PC	FRAME
Aurora	HR@1	0.2777	0.3750	0.4722	0.4166	0.5138	0.4513	**0.5625**
	HR@5	0.1944	0.2972	0.4013	0.2750	0.4486	0.4180	**0.4652**
	HR@10	0.1631	0.2791	0.3590	0.2458	**0.4180**	0.3618	0.4166
	MRR	0.3880	0.5026	0.5963	0.5136	0.6448	0.6077	**0.6729**
Avgust	HR@1	0.7960	0.8352	0.8549	0.8666	0.8588	0.8705	**0.8823**
	HR@5	0.6901	0.7850	0.7788	0.8023	0.8282	0.8219	**0.8329**
	HR@10	0.5929	0.7360	0.7364	0.7498	0.7721	0.7674	**0.7827**
	MRR	0.8380	0.8724	0.8913	0.9012	0.9014	0.9114	**0.9184**

Table 8: Ablation analysis over 2 datasets between FRAME and its variants, where the best results are in **bold**.

By considering only the top 10 results, Hits@10 emphasizes precision while still providing a reasonable measure of recall, giving a holistic view of each embedding's capabilities.

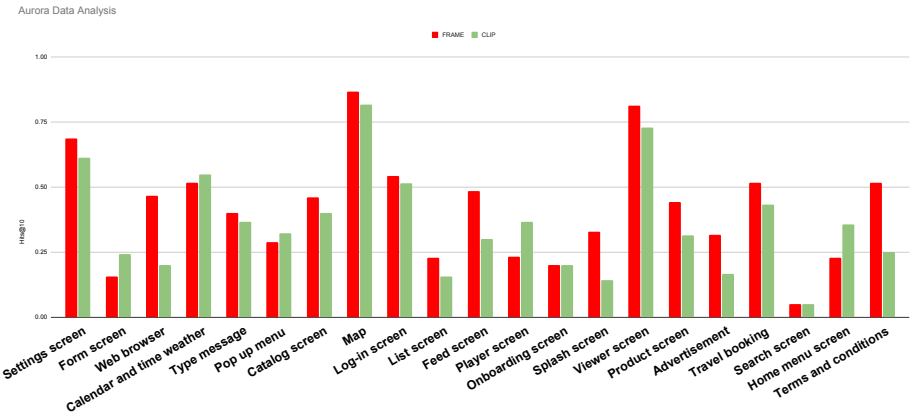

Figure 20: Visualization of FRAME vs CLIP Hits@10 on different screen types in the Aurora dataset

Figure 20 presents the difference between FRAME and CLIP embeddings on various screen types in the Aurora dataset. The bar chart shows FRAME in red and CLIP in green. We see that FRAME outperforms CLIP in every category except for four screen types. This gives FRAME an advantage above CLIP in 17/21 screen categories. In the context of UI screen retrieval, the FRAME embedding's advantage over CLIP embedding's performance across 17 out of 21 screen categories highlights FRAME's efficacy in capturing nuanced visual features specific to UI elements. FRAME's performance in comparison to CLIP suggests that Frame embeddings are better suited for tasks requiring precise understanding and representation of static UI screens, potentially leading to improved retrieval accuracy and user experience in UI-related applications.

Figure 20 presents the difference between FRAME and CLIP embeddings on various screen types in the Avgust dataset. The bar chart shows FRAME in red and CLIP in green. We see that FRAME outperforms CLIP in every category except for four screen types. This gives FRAME an advantage above CLIP in 20/25 screen categories. In the context of UI screen retrieval, the FRAME embedding's advantage over CLIP embedding's performance across 20 out of 25 screen categories highlights FRAME's efficacy in capturing nuanced visual features specific to UI ele-

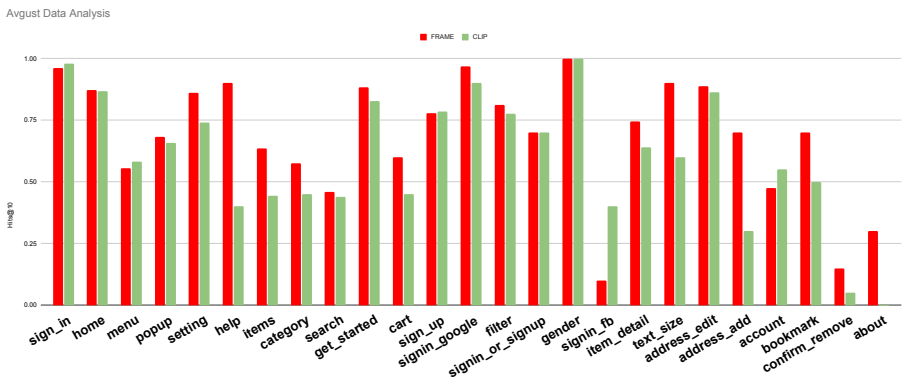

Figure 21: Visualization of FRAME vs CLIP Hits@10 on different screen types in the Avgust dataset

ments. FRAME's performance in comparison to CLIP suggests that Frame embeddings are better suited for tasks requiring precise understanding and representation of static UI screens, potentially leading to improved retrieval accuracy and user experience in UI-related applications.

5.3.3 RQ$_3$ *How important is the structural embedding propagation between graphs in order to enhance the embedding?*

FRAME creates a graph representation of the screen and augments weights within the nodes based on proximity and neighborhoods. This approach is designed to provide a set of weighted embeddings that can ingrain structural information into the embedding. To demonstrate the need for this propagation, we conduct an ablation study with six different variants of FRAME, including and excluding various properties from the embedding. Table 8 has the results for the ablation study for both datasets with each variant of FRAME. FRAME outperforms all variants of frame in every metric except for the version of frame that has the modified CLIP embedding and the propagated CLIP embedding in Hits@10 on the Aurora dataset. As described in Section **??**, there are 6 variants of FRAME. Table 8 shows a comparison of FRAME's results to all the variants. In the variants that do not have the larger augmented CLIP embedding and have one of the two possible propagated embeddings, Variants: NC-NPB-PC and NC-PB-NPC, we see poor performance. This poor performance can be attributed to the lack of broader screen information. These variants are solely dependent on the propagation between the icons or the text on the screen.

In the variants that do have the larger augmented CLIP embedding and have one of the two possible propagated embeddings, Variants: C-NPB-PC and C-PB-NPC, we see an improved performance to the previous two variants. However, these variants still underperform when compared to the results for FRAME. Given the lower performance of these two variants, it is assumed that the presence of both propagated is important to provide a higher structural context for the overall embedding.

Given the results of the other four embeddings, FRAME has two more configurations which offer a direct insight into the impact of the modified CLIP embedding and propagated embedding inclusion. The two main configurations to consider are NC-PB-PC and C-NPB-NPC.

- **NC-PB-PC**: This variant of FRAME only has the propagated CLIP and BERT embedding.

This is the information that ingrains the modified CLIP with structural information. However, on its own, the propagated embeddings are the fourth best variant of FRAME. It outperforms the augmented version of CLIP which is just an image with no structural enhancements. This is shown in Table 8.

- **C-NPB-NPC**: This variant of FRAME only has the augmented CLIP embedding without the propagated CLIP and BERT embeddings. This is the base image that is going to be enhanced with structural context. This variant of FRAME is only the third best variant of FRAME, only beating out the individual propagated CLIP and BERT embeddings, as shown in Table 8.

However, together, FRAME can combine the two variants and leverage the properties of the augmented CLIP embedding and the structural properties of the propagated embeddings to create a structure-biased embedding. This results in FRAME outperforming every variant of itself, demonstrating the impact of each element within the ablation process.

5.3.4 $\mathbf{RQ}_4$ *How well does* FRAME *leverage its ability to abstract the screen and disregard styling?*

Dataset	Metrics	NB-C	NB-NC	B-NC	FRAME
Aurora	HR@1	0.4930	0.5347	0.5208	**0.5625**
	HR@5	0.4069	0.4250	0.4527	**0.4652**
	HR@10	0.3638	0.3805	0.4138	**0.4166**
	MRR	0.6171	0.6484	0.6492	**0.6729**
Avgust	HR@1	0.8505	0.8784	0.8784	**0.8823**
	HR@5	0.8119	0.8298	0.8266	**0.8329**
	HR@10	0.7658	0.7827	0.7799	**0.7827**
	MRR	0.9094	0.9118	0.9138	**0.9184**

Table 9: Results over 2 datasets, where the best results are in **bold**, between FRAME and its variants including both contrast and black and white.

FRAME addresses color and style bias limitations in prior work by augmenting the image. There are three image preprocessing variants for the FRAME embedding. The results for this augmentation ablation study are presented in Table 9.

NB-C
This variant of FRAME only increases the contrast on the input image. The results show that increasing the contrast on the input image is the least effective form of augmentation. As shown in Table 8, this variant of FRAME is the least effective.

B-NC
This variant of FRAME does not increase the contrast in the input image and only converts it to greyscale. The results show a noticeable increase in the performance of this variant, making it the

highest-performing variant behind FRAME. This variant benefits from the lack of color and allows the embedding to take on the structural aspects of the propagated embedding without allowing color bias to play a factor.

NB-NC
This variant of FRAME does not increase the contrast or make the image black-and-white. The results for this variant are interesting since it outperformed the contrast only variant. This gives an insight into how important the black-and-white is for the high contrast augmentation to define clear edges.

Together with the inclusion of the black-and-white and high contrast modification, FRAME can create an embedding that outperforms its variants while excluding color and increasing edge detection capabilities to define elements clearly on the screen.

FRAME is a novel approach to enhance existing image embeddings with structural information to embed UIs. There has been limited work in this area, so we present related work that has similar concepts to what FRAME tries to achieve. We looked at previous work in UI Embeddings, UI Search and similarity, and Graph Based UI Representations.

5.4 Embedding Related Works

There exists a limited body of work when considering embeddings for UI screens [76], however, there have been increased efforts to provide a screen-specific embedding.

One such effort is Screen2Vec, proposed by Li et al. [76]. Screen2Vec tackles the lack of UI Screen embeddings and proposes its own neural embeddings. Screen2Vec uses BERT text embeddings to take into account the text on the screen. Additionally, they use a layout embedding to embed the images with some layout information. Screen2Vec differs by creating their own embedding while using the Rico data [**?**] and uses the extracted screen hierarchy to embed screen hierarchy. This layout-based screen embedding is also employed by Deka et al. in Rico [**?**]. Both of these approaches use the same layout-based screen embedding which differs from the FRAME proposes. These two embedding techniques aim to create a new embedding while FRAME aims to enhance an existing embedding to be suitable for screen structure. FRAME uses a computer vision approach to find the elements on the screen and use their positional information on the screen as opposed to finding elements within the screen hierarchy. FRAME also utilizes a graph-based approach to creating layouts while considering both element visual and text characteristics within the graph.

Li et al. [77] proposed VUT, a UI transformer for multimodal multi-task user interface modeling. This approach considered the hierarchical structure, image, and language properties of the screen to create their screen representations. VUT has two Transformer architectures, an Image-Structure model, and a Question-Answer model. The image-structure model combines screenshot and view hierarchy data for UI encoding while the question-answer model encodes questions while attending to UI encodings from the Image-Structure model, generating text answers for language tasks and serving as the question encoder for grounding tasks to locate UI elements for action. However, FRAME is a lightweight approach that takes existing embeddings and enhances them with structural properties and does not create its embedding using a transformer architecture. The structural properties of the screen in FRAME are graph and visual position-based as opposed to

hierarchy-based representation in VUT. VUT is an excellent baseline comparison for FRAME, however, the model and architecture to use VUT are currently unavailable to the public.

Bai et al. [35] proposed UIBert, a transformer-based joint image-text model to learn generic feature representations for a UI and its components [35]. This embedding is similar to VUT in its goal to create an embedding using image, text, and structural properties of the screen. Similar to VUT, UIBert also considers the view hierarchies for each screen and the leaf nodes. UIBert uses the element information on the screen in conjunction with the visual text to create an embedding with both textural and structural information. However, as discussed with VUT, UIBert is a new transformer-based embedding framework, whereas FRAME enhances existing embeddings. Additionally, FRAME interprets structure as visual positions on the GUI rather than an image's position in the hierarchy. UIBert may have provided an insight into the performance of FRAME as a baseline, however, the models for UIBert are not available to the public.

UI Search and Similarity

This paper presents a retrieval-motivated image embedding. It is important to consider other ways screen retrieval has been done [65, **?**, 50] and how FRAME can be used to enhance search based on its retrieval-based evaluation. Guigleproposed by Bernal-Cardenas et al. [**?**] is a UI search engine. They use metadata information to facilitate a filter-based search. This method of finding embeddings is functional but does not provide and mbedding based search. Search is done through filters to find an image.

GeminiScope proposed by Mao et al. [**?**] proposed a GUI similarity metric that uses the leaf nodes in the hierarchy tree to determine the position of the elements on the screen. They compute the absolute values of UI-related features, such as position and size, and use these metrics as their numerical representation for the screen. This allows them to perform distance calculations on the different images based on UI feature locations. FRAME, like GeminiScope, considers the position of UI elements on the screen. However, GeminiScope does not consider the relations between the embeddings, unlike FRAME. Additionally, FRAME considers the text on the screen along with the actual visual representation of the icons.

UI Comprehension

To modify or test a UI, it is necessary to have a semantic understanding of the UI. Many tools use the android screen hierarchy [35, 77, 76, 126], however, the hierarchy is not always available. We present a prior work that has used computer vision techniques to gain a semantic understanding of the elements on the screen. Spotlight, a tool introduced by Li et al. [74] is a tool that uses computer vision to create a vision-language model architecture that can assist in screen comprehension tasks. Spotlight can generate text to suggest a screen or icon's function. This helps create lightweight icon detection for screens without a hierarchy. Following this line of thinking, we designed FRAME to be hierarchy-free and use a computer vision approach to detect icons on the screen, resulting in an embedding that only needs a screenshot as input.

Additionally, Schoop et al. [126] proposed a technique to classify tappable icons on a screen given an image. This work is done by using XRAI. XRAI identifies and emphasizes influential areas within the input screenshot, which affect the prediction of capability for the chosen region. Additionally, it employs k-Nearest Neighbors to showcase mobile UIs from the dataset that exhibit contrasting influences on the perception of tappability. Though the intended application was not re-

lated to screen embedding creation, tappable icons can uncover important relations within the way the icons on a screen are laid out. FRAME does not consider the tappability of the icon, however, it is a direction that can be considered. FRAME is concerned with identifying niche relationships between icons and where they are positioned on the screen as opposed to their functionality.

Graph Based UI Representation
Li et al. proposed Sugilite [**?**], a graph-based UI element search for UI screens. Sugilite creates a UI snapshot graph using element metadata and position to identify each element on the screen. This allows users to write test scripts to test certain elements of the screen using natural language queries. WHile this is not an embedding, it utilizes a similar intuition to FRAME by considering the graph-based approach to mapping elements. This method differs from FRAME in two ways. Sugilite does not perform any structural relationship analysis between the different elements, they use the graph as a means to query the UI and find elements. Sugilite also uses metadata from the UI screens to create the graph. FRAME aims be more versatile, therefore it creates an embedding for a screen from only the image of the screen.

5.5 Summary

In this paper, we presented FRAME, an approach for enhancing UI screen embeddings with structural information. We measured the performance and generalizability of FRAME to various open source UIs. Our results indicate that FRAME is effective in practice and outperforms key baselines, offering a novel approach to create vectors for UIs. Future work will examine the potential create embeddings for web apps and focus on creating an entirely new embedding as opposed to enhancing current ones.

6 Accessible UI Search

We take the work done in the previous two projects and combine them in SeachAccess. In UI design, initial UI mockup sketches serve as the starting point for developers, offering an image that can be easily modified. However, addressing accessibility concerns within these designs often proves challenging. To bridge this gap, we introduce SearchAccess, a UI search engine designed to identify accessibility issues in mockups and find similar screens which are more accessible.

SearchAccess goes beyond traditional search functionalities, providing developers with a visual interface to identify and address accessibility issues within their UI mockups. I leverage computer vision techniques to enable developers to locate accessibility issues within the mockup screens. I use an image embedding representation of the screen to facilitate search between other screens.

SearchAccess is built with 6 custom detectors. Each detector is built using computer vision techniques that are able to detect accessibility violations given only a screenshot. This makes it easy for developers to check for accessibility and make changes as necessary. Additionally, the search functionality is built to be multimodal, offering developers the opportunity to find similar screens with either the mockup or a text description and mockup. The screens displayed as most similar to the input screens are screens that are both similar in style, but more accessible than the input screen. The similar, more accessible, screens can provide an insight to developers on how to improve the accessibility within their own screens.

This project aims to give developers the tools so they can make informed accessibility driven decisions early in the design process.

6.1 Current Progress

Currently, the search engine has been implemented along with the search functionality. SearchAccess is fully functioning locally and we aim to host the tool soon. We ideally intend to perform a mixed evaluation of qualitative and quantitative measures. These are the proposed research questions. These questions are subject to change:

RQ$_1$: *How does SearchAccess perform in screen retrieval tasks?*

RQ$_2$: *How accurate are the detectors in SeachAccess?*

RQ$_3$: *Are developers able to identify accessibility issues within their UI designs?*

RQ$_4$: *Do developers benefit from UI search when looking to make their apps more accessible?*

We intend to evaluate RQ1 and RQ2 quantitatively using retrieval metrics and accuracy metrics respectively. RQ3 and RQ4 will examine the qualitative impact this tool will have on developers. This study will be a performance study to analyze how well SearchAccess is able to supplement the accessibility focused design process and how it can expedite the process of making UIs more accessible.

7 Literature Review

In order to fully capture the current landscape of accessibility guidelines that may impact various populations of users, and to aid in selecting the most impactful guidelines that aim to assist motor impaired users we conducted a systematic literature reviews on research at the intersection of software engineering, human-computer interaction, and accessibility. To conduct this review, we followed the methodology set forth by Kitchenham et al. [70]. We defined a single research question that asked *"What accessibility guidelines have been identified and discussed in prior research?"*. We used the relatively simple search string of "accessibility" to search DBLP, the ACM Digital Library, and IEEE Xplore, for work at the intersection of accessibility and software engineering for the date range of January 2010 - December 2022. The purpose of using such a simple search string was to "cast a wide net" and ensure that we did not miss important work. We defined inclusion criteria as follows: (i) must have been published in our studied date range, (ii) must have been published at one of 16 conference venues (ICSE, FSE, ASE, ICSME, MSR, ICPC, ISSTA, ICST, SANER, UIST, CHI, SPLASH, OOPSLA, PLDI, CSCW, ASSETS) or 5 journal venues (TSE, TOSEM, EMSE, JSS, ASE) that cross cut software engineering, HCI, and accessibility, (iii) the paper must describe a study or developer tool directly related to an accessibility issue that impacts end-users. The scope of our search was limited to these venues and digital libraries as they provide the highest quality of research in all matters including accessibility. Our search results returned 2948 papers from our selected conferences within our given date range. Then, two authors manually checked each paper for adherence to the final inclusion criteria, resulting in 20 papers that intersect our desired research areas *and* discuss developer guidelines for addressing accessibility issues. In addition to these 20 identified primary studies, we also examined Apple's and Google's design guidelines related to accessibility [1, 17], as several of our primary studies referenced these sources.

7.1 Current Progress

Currently I have found almost 2,948 different papers and and narrowed it down to about 100 papers to begin the data extraction process. I intend to extract the data from these papers present a comprehensive study about the current state of developer tools focused on accessibility. Below are a list of initial research questions that may be subject to change barring further discussion.

RQ$_1$: *Is the research targeted at automating developer activities, enhancing existing software, or creating guidelines for developers?*

RQ$_2$: *What software domains does research on accessibility typically target?*

RQ$_3$: *Which populations of users with accessibility needs has software engineering targeted?*

RQ$_4$: *What type of data do studies use and how can the quality of data and its collection suggest an impact on the research in the field?*

RQ$_5$: *What are the primary means of evaluation for research that targets users with accessibility needs.?*

8 Research Plan

During the Fall 2023, I will continue working on the proposed work and measure how likely the findings are motivated to investigate into new areas. In terms of writing my dissertation, I intend to defend it during 2024.

References

[1] Accessibility: Apple Human Interface Guidelines. https://developer.apple.com/design/human-interface-guidelines/foundations/accessibility/.

[2] Accessibility Scanner. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor.

[3] Android Switch Access Service. https://support.google.com/accessibility/android/answer/6122836?hl=en.

[4] Google Cloud Vision API. https://cloud.google.com/vision/docs/ocr.

[5] MotorEase GitHub Repository. https://github.com/SageSELab/MotorEase.

[6] MotorEase Website. https://sagelab.io/MotorEase.

[7] MotorEase Zenodo Archive. https://zenodo.org/doi/10.5281/zenodo.10460700.

[8] Universal Design Definition and Guidelines. https://www.section508.gov/blog/Universal-Design-What-is-it/.

[9] Web content accessibility guidelines (wcag) 2.1.

[10] "Design for Android: android developers," https://developer.android.com/design.

[11] World Report on Disability http://www.who.int/disabilities/world_report/2011/en/, 2011.

[12] ADA Web Accessibility Lawsuit Recap Report https://blog.usablenet.com/2018-ada-web-accessibility-lawsuit-recap-report, 2018.

[13] ADAlaws https://www.ada.gov/cguide.htm, 2019.

[14] Adaptivity and layout - visual design - ios - human interface guidelines - apple developer https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/adaptivity-and-layout/, 2019.

[15] Adaptivity and layout - visual design - IOS - human interface guidelines - apple developer https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/adaptivity-and-layout/, 2019.

[16] Accessibility Guide https://accessibility.18f.gov/checklist/accessibility, 2022.

[17] Accessibility https://developer.android.com/guide/topics/ui/accessibility, 2022.

[18] Accessibility https://support.google.com/accessibility, 2022.

[19] Accessibility https://www.apple.com/accessibility/, 2022.

[20] Motor Impairment https://accessibility.huit.harvard.edu/disabilities/motor-impairment?page=1, 2022.

[21] Torchvision- torchvision main documentation. https://pytorch.org/vision/stable/index.html, 2022.

[22] Web content accessibility guidelines (WCAG). https://www.w3.org/TR/WCAG21/, journal=W3C, 2022.

[23] "live stream your virtual event online: Vimeo enterprise," https://vimeo.com/enterprise/stream-virtual-events, 2022.

[24] https://f-droid.org/, F-droid.

[25] https://play.google.com/store, Google Play Store.

[26] https://labelstud.io, LabelStudio.

[27] Julio Abascal, Amaia Aizpurua, Idoia Cearreta, Borja Gamecho, Nestor Garay-Vitoria, and Raúl Miñón. Automatically Generating Tailored Accessible User Interfaces for Ubiquitous Services. In *The Proceedings of the 13th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '11, page 187–194, New York, NY, USA, 2011. ACM.

[28] Amaia Aizpurua, Myriam Arrue, Simon Harper, and Markel Vigo. Are Users the Gold Standard for Accessibility Evaluation? In *Proceedings of the 11th Web for All Conference*, W4A '14, New York, NY, USA, 2014. ACM.

[29] Leonelo Dell Anhol Almeida and Maria Cecília Calani Baranauskas. Universal Design Principles Combined with Web Accessibility Guidelines: A Case Study. In *Proceedings of the IX Symposium on Human Factors in Computing Systems*, IHC '10, page 169–178, Porto Alegre, BRA, 2010. Brazilian Computer Society.

[30] Tayfun Alpay, Sven Magg, Philipp Broze, and Daniel Speck. Multimodal video retrieval with CLIP: a user study. *Information Retrieval Journal*, 26(1–2), September 2023.

[31] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. Accessibility Issues in Android Apps: State of Affairs, Sentiments, and Ways Forward. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1323–1334, New York, NY, USA, 2020. ACM.

[32] Douglas Astler, Harrison Chau, Kailin Hsu, Alvin Hua, Andrew Kannan, Lydia Lei, Melissa Nathanson, Esmaeel Paryavi, Michelle Rosen, Hayato Unno, Carol Wang, Khadija Zaidi, Xuemin Zhang, and Cha-Min Tang. Increased Accessibility to Nonverbal Communication through Facial and Expression Recognition Technologies for Blind/Visually Impaired Subjects. In *The Proceedings of the 13th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '11, page 259–260, New York, NY, USA, 2011. ACM.

[33] Shiri Azenkot, Jacob O. Wobbrock, Sanjana Prasain, and Richard E. Ladner. Input finger detection for nonvisual touch screen text entry in ¡i¿perkinput¡/i¿. In *Proceedings of Graphics Interface 2012*, GI '12, page 121–129, CAN, 2012. Canadian Information Processing Society.

[34] Chongyang Bai, Xiaoxue Zang, Ying Xu, Srinivas Sunkara, Abhinav Rastogi, Jindong Chen, and Blaise Aguera y Arcas. Uibert: Learning generic multimodal representations for ui understanding. 2021.

[35] Chongyang Bai, Xiaoxue Zang, Ying Xu, Srinivas Sunkara, Abhinav Rastogi, Jindong Chen, and Blaise Aguera y Arcas. UIBert: Learning Generic Multimodal Representations for UI Understanding, 2021.

[36] Mohammad Bajammal and Ali Mesbah. Semantic web accessibility testing via hierarchical visual analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1610–1621, 2021.

[37] Mohammad Bajammal and Ali Mesbah. Semantic Web Accessibility Testing via Hierarchical Visual Analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1610–1621, 2021.

[38] Mohammad Bajammal and Ali Mesbah. Semantic Web Accessibility Testing via Hierarchical Visual Analysis. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1610–1621. IEEE Press, 2021.

[39] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. BEiT: BERT Pre-Training of Image Transformers, 2022.

[40] Larwan Berke, Christopher Caulfield, and Matt Huenerfauth. Deaf and hard-of-hearing perspectives on imperfect automatic speech recognition for captioning one-on-one meetings. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '17, page 155–164, New York, NY, USA, 2017. Association for Computing Machinery.

[41] C. Bernal-Cardenas, N. Cooper, M. Havranek, K. Moran, O. Chaparro, D. Poshyvanyk, and A. Marcus. Translating Video Recordings of Complex Mobile App UI Gestures into Replayable Scenarios. *IEEE Transactions on Software Engineering*, 49(04):1782–1803, apr 2023.

[42] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. Translating Video Recordings of Mobile App Usages into Replayable Scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 309–321, New York, NY, USA, 2020. ACM.

[43] Danielle Bragg, Naomi Caselli, John W. Gallagher, Miriam Goldberg, Courtney J. Oka, and William Thies. Asl sea battle: Gamifying sign language data collection. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[44] Giorgio Brajnik, Chiara Pighin, and Sara Fabbro. Model-Based Automated Accessibility Testing. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '15. ACM, 2015.

[45] Rocio Calvo, Faezeh Seyedarabi, and Andreas Savva. Beyond Web Content Accessibility Guidelines: Expert Accessibility Reviews. In *Proceedings of the 7th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-Exclusion*, DSAI 2016, page 77–84, New York, NY, USA, 2016. ACM.

[46] Chang Che, Qunwei Lin, Xinyu Zhao, Jiaxin Huang, and Liqiang Yu. Enhancing Multimodal Understanding with CLIP-Based Image-to-Text Transformation. In *Proceedings of the 2023 6th International Conference on Big Data Technologies*, ICBDT '23, page 414–418, New York, NY, USA, 2023. Association for Computing Machinery.

[47] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 665–676, New York, NY, USA, 2018. ACM.

[48] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 322–334, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 322–334, New York, NY, USA, 2020. ACM.

[50] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 322–334, New York, NY, USA, 2020. ACM.

[51] Sen Chen, Chunyang Chen, Lingling Fan, Mingming Fan, Xian Zhan, and Yang Liu. Accessible or Not? An Empirical Investigation of Android App Accessibility. *IEEE Transactions on Software Engineering*, 48(10):3954–3968, 2022.

[52] Paul T. Chiou, Ali S. Alotaibi, and William G. J. Halfond. Detecting and Localizing Keyboard Accessibility Failures in Web Applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 855–867, New York, NY, USA, 2021. ACM.

[53] Paul T. Chiou, Ali S. Alotaibi, and William G.J. Halfond. BAGEL: An Approach to Automatically Detect Navigation-Based Web Accessibility Barriers for Keyboard Users. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, 2023. ACM.

[54] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? In *ASE*. IEEE, 2015.

[55] Henrique Neves da Silva, Andre Takeshi Endo, Marcelo Medeiros Eler, Silvia Regina Vergilio, and Vinicius H. S. Durelli. On the Relation between Code Elements and Accessibility Issues in Android Apps. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, SAST 20, page 40–49, New York, NY, USA, 2020. ACM.

[56] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 3844–3852, Red Hook, NY, USA, 2016. Curran Associates Inc.

[57] 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, page 845–854, New York, NY, USA, 2017. ACM.

[58] 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, page 767–776, New York, NY, USA, 2016. Association for Computing Machinery.

[59] Marcelo Medeiros Eler, Jose Miguel Rojas, Yan Ge, and Gordon Fraser. Automated Accessibility Testing of Mobile Apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126, 2018.

[60] Leandro Flórez-Aristizábal, Sandra Cano, César A. Collazos, Andrés F. Solano, and Stephen Brewster. DesignABILITY. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, May 2019.

[61] Raymond Fok, Harmanpreet Kaur, Skanda Palani, Martez E. Mott, and Walter S. Lasecki. Towards more robust speech interactions for deaf and hard of hearing users. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '18, page 57–67, New York, NY, USA, 2018. Association for Computing Machinery.

[62] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. Automatically Generating User Interfaces Adapted to Users' Motor and Vision Capabilities. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, page 231–240, New York, NY, USA, 2007. ACM.

[63] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical GUI Testing of Android Applications via Model Abstraction and Refinement. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 269–280. IEEE Press, 2019.

[64] Madeleine Havranek, Carlos Bernal-Cárdenas, Nathan Cooper, Oscar Chaparro, Denys Poshyvanyk, and Kevin Moran. V2S: A Tool for Translating Video Recordings of Mobile App Usages into Replayable Scenarios. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 65–68, 2021.

[65] Forrest Huang, John F. Canny, and Jeffrey Nichols. Swire: Sketch-based User Interface Retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–10, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Shaun K. Kane, Meredith Ringel Morris, Annuska Z. Perkins, Daniel Wigdor, Richard E. Ladner, and Jacob O. Wobbrock. Access overlays. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, October 2011.

[67] Safwat Ali Khan, Wenyu Wang, Yiran Ren, Bin Zhu, Jiangfan Shi, Alyssa McGowan, Wing Lam, and Kevin Moran. AURORA: Navigating UI Tarpits via Automated Neural Screen Understanding, 2024.

[68] Hyun K. Kim, Changwon Kim, Eunyoung Lim, and Hyunjin Kim. How to Develop Accessibility UX Design Guideline in Samsung. In *Proceedings of the 18th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct*, MobileHCI '16, page 551–556, New York, NY, USA, 2016. ACM.

[69] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[70] Barbara Ann Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 07 2007.

[71] Junhan Kong, Mingyuan Zhong, James Fogarty, and Jacob O. Wobbrock. New Metrics for Understanding Touch by People with and without Limited Fine Motor Function. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '21, New York, NY, USA, 2021. ACM.

[72] Ajay Krishna Vajjala, D Meher, Shrunal Pothagoni, Ziwei Zhu, and D Rosenblum. Vietoris-rips complex: A new direction for cross-domain cold-start recommendation. 2024.

[73] Arun Krishnavajjala, SM Hasan Mansur, Justin Jose, and Kevin Moran. Motorease: Automated detection of motor impairment accessibility issues in mobile app uis. 2024.

[74] Gang Li and Yang Li. Spotlight: Mobile UI Understanding using Vision-Language Models with a Focus, 2023.

[75] Jiasheng Li, Zeyu Yan, Ebrima Haddy Jarjue, Ashrith Shetty, and Huaishu Peng. TangibleGrid: Tangible Web Layout Design for Blind Users. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, UIST '22, New York, NY, USA, 2022. ACM.

[76] Junchen Li, Garreth W. Tigwell, and Kristen Shinohara. Accessibility of High-Fidelity Prototyping Tools. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, May 2021.

[77] Yang Li, Gang Li, Xin Zhou, Mostafa Dehghani, and Alexey Gritsenko. VUT: Versatile UI Transformer for Multi-Modal Multi-Task User Interface Modeling, 2021.

[78] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. DroidBot: a lightweight UI-guided test input generator for Android. In *ICSE-C*. IEEE, 2017.

[79] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. Test Automation in Open-Source Android Apps: A Large-Scale Empirical Study. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1078–1089, New York, NY, USA, 2021. ACM.

[80] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. Mining Android app usages for generating actionable GUI-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 111–122. IEEE Press, 2015.

[81] Mario Linares-Vásquez, Cárlos Bernal-Cardenas, Kevin Moran, and Denys Poshyvanyk. How do Developers Test Android Applications? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 613–622, 2017.

[82] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–410, 2017.

[83] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. Learning Design Semantics for Mobile Apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, page 569–579, New York, NY, USA, 2018. Association for Computing Machinery.

[84] Zhe Liu. Discovering UI Display Issues with Visual Understanding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1373–1375, New York, NY, USA, 2021. ACM.

[85] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 398–409, New York, NY, USA, 2021. ACM.

[86] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for Android apps. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *ESEC/FSE'13*, pages 224–234. ACM, 2013.

[87] I. Scott MacKenzie, R. William Soukoreff, and Joanna Helga. 1 thumb, 4 buttons, 20 words per minute: Design and evaluation of h4-writer. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, page 471–480, New York, NY, USA, 2011. Association for Computing Machinery.

[88] I. Scott MacKenzie, R. William Soukoreff, and Joanna Helga. 1 Thumb, 4 Buttons, 20 Words per Minute: Design and Evaluation of H4-Writer. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, page 471–480, New York, NY, USA, 2011. ACM.

[89] SM Hasan Mansur, Sabiha Salma, Damilola Awofisayo, and Kevin Moran. Aidui: Toward automated recognition of dark patterns in user interfaces. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1958–1970. IEEE, 2023.

[90] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for Android applications. In *ISSTA*. ACM, 2016.

[91] Delvani Antônio Mateus, Carlos Alberto Silva, Marcelo Medeiros Eler, and André Pimenta Freire. Accessibility of Mobile Applications: Evaluation by Users with Visual Impairment and by Automated Tools. In *Proceedings of the 19th Brazilian Symposium on Human Factors in Computing Systems*, IHC '20, New York, NY, USA, 2020. ACM.

[92] Thomas B. McHugh and Cooper Barth. Assistive Technology Design as a Computer Science Learning Experience. In *Proceedings of the 22nd International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '20, New York, NY, USA, 2020. ACM.

[93] Forough Mehralian, Navid Salehnamadi, and Sam Malek. Data-Driven Accessibility Repair Revisited: On the Effectiveness of Generating Labels for Icons in Android Apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 107–118, New York, NY, USA, 2021. ACM.

[94] Lauren R. Milne and Richard E. Ladner. Blocks4All. In *CHI'18*. ACM, April 2018.

[95] Kyle Montague, Vicki L. Hanson, and Andy Cobley. Designing for Individuals: Usable Touch-Screen Interaction through Shared User Models. In *Proceedings of the 14th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '12, page 151–158, New York, NY, USA, 2012. ACM.

[96] Kyle Montague, Hugo Nicolau, and Vicki L. Hanson. Motor-Impaired Touchscreen Interactions in the Wild. In *Proceedings of the 16th International ACM SIGACCESS Conference on Computers ; Accessibility*, ASSETS '14, page 123–130, New York, NY, USA, 2014. ACM.

[97] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering*, 46(2):196–221, 2020.

[98] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. Automated reporting of gui design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 165–175, New York, NY, USA, 2018. Association for Computing Machinery.

[99] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. Automated reporting of GUI design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 165–175, New York, NY, USA, 2018. ACM.

[100] Kevin Moran, Mario Linares-Vasquez, Carlos Bernal-Cardenas, Christopher Vendome, and Denys Poshyvanyk. CrashScope: A Practical Tool for Automated Testing of Android Applications.

[101] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 33–44, 2016.

[102] Kevin Moran, Cody Watson, John Hoskins, George Purnell, and Denys Poshyvanyk. Detecting and summarizing GUI changes in evolving mobile apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 543–553, New York, NY, USA, 2018. ACM.

[103] MulongXie. UIED - UI element detection, detecting UI elements from UI screenshots or drawnings, 2021.

[104] Tuan Anh Nguyen and Christoph Csallner. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 248–259, 2015.

[105] Hugo Nicolau, Kyle Montague, Tiago Guerreiro, André Rodrigues, and Vicki L. Hanson. Typing performance of blind users: An analysis of touch behaviors, learning effect, and in-situ usage. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility*, ASSETS '15, page 273–280, New York, NY, USA, 2015. Association for Computing Machinery.

[106] Kirk Norman, Yevgeniy Arber, and Ravi Kuber. How Accessible is the Process of Web Interface Design? In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '13, New York, NY, USA, 2013. ACM.

[107] Francisco Nunes, Paula Alexandra Silva, João Cevada, Ana Correia Barros, and Luís Teixeira. User interface design guidelines for smartphone applications for people with Parkinson's disease. *Universal Access in the Information Society*, 15(4):659–679, October 2015.

[108] Uran Oh, Shaun K. Kane, and Leah Findlater. Follow That Sound: Using Sonification and Corrective Verbal Feedback to Teach Touchscreen Gestures. In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '13, New York, NY, USA, 2013. ACM.

[109] Pekka Parhi, Amy K. Karlson, and Benjamin B. Bederson. Target Size Study for One-Handed Thumb Use on Small Touchscreen Devices. In *Proceedings of the 8th Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '06, page 203–210, New York, NY, USA, 2006. ACM.

[110] Kyudong Park, Taedong Goh, and Hyo-Jeong So. Toward Accessible Mobile Application Design: Developing Mobile Application Accessibility Guidelines for People with Visual Impairment. In *Proceedings of HCI Korea*, HCIK '15, page 31–38, Seoul, KOR, 2014. Hanbit Media, Inc.

[111] Amy Pavel, Gabriel Reyes, and Jeffrey P. Bigham. Rescribe. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, October 2020.

[112] Yi-Hao Peng, Muh-Tarng Lin, Yi Chen, TzuChuan Chen, Pin Sung Ku, Paul Taele, Chin Guan Lim, and Mike Y. Chen. Personaltouch: Improving touchscreen usability by personalizing accessibility settings based on individual user's touchscreen interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–11, New York, NY, USA, 2019. Association for Computing Machinery.

[113] Yi-Hao Peng, Muh-Tarng Lin, Yi Chen, TzuChuan Chen, Pin Sung Ku, Paul Taele, Chin Guan Lim, and Mike Y. Chen. PersonalTouch: Improving Touchscreen Usability by Personalizing Accessibility Settings Based on Individual User's Touchscreen Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–11, New York, NY, USA, 2019. ACM.

[114] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global Vectors for Word Representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014.

[115] Lauren Race, Amber James, Andrew Hayward, Kia El-Amin, Maya Gold Patterson, and Theresa Mershon. Designing sensory and social tools for neurodivergent individuals in social media environments. In *The 23rd International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '21, New York, NY, USA, 2021. Association for Computing Machinery.

[116] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning Transferable Visual Models From Natural Language Supervision, 2021.

[117] Nagendra Prasad Kasaghatta Ramachandra and Christoph Csallner. Testing Web-Based Applications with the ¡u¿v¡/u¿oice ¡u¿c¡/u¿ontrolled ¡u¿a¡/u¿ccessibility and ¡u¿t¡/u¿esting Tool (VCAT). In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 208–209, New York, NY, USA, 2018. ACM.

[118] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[119] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. Examining Image-Based Button Labeling for Accessibility in Android Apps through Large-Scale Analysis. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*. ACM, October 2018.

[120] Lisa Jo Rudy. Overview of assistive technology for autism, Feb 2021.

[121] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. Latte: Use-case and assistive-service driven automated accessibility testing framework for android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[122] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. ACM.

[123] Navid Salehnamadi, Forough Mehralian, and Sam Malek. Groundhog: An Automated Accessibility Crawler for Mobile Apps. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, 2022.

[124] Mara Taynar Santiago and Anna Beatriz Marques. Are User Reviews Useful for Identifying Accessibility Issues That Autistic Users Face? An Exploratory Study. In *Proceedings of the 21st Brazilian Symposium on Human Factors in Computing Systems*, IHC '22, New York, NY, USA, 2022. ACM.

[125] Zhanna Sarsenbayeva, Niels van Berkel, Eduardo Velloso, Jorge Goncalves, and Vassilis Kostakos. Methodological Standards in Accessibility Research on Motor Impairments: A Survey. *ACM Comput. Surv.*, may 2022. Just Accepted.

[126] Eldon Schoop, Xin Zhou, Gang Li, Zhourong Chen, Bjoern Hartmann, and Yang Li. Predicting and Explaining Mobile UI Tappability with Vision Modeling and Saliency Analysis. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery.

[127] Brent N. Shiver and Rosalee J. Wolfe. Evaluating Alternatives for Better Deaf Accessibility to Selected Web-Based Multimedia. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '15, page 231–238, New York, NY, USA, 2015. ACM.

[128] Giovanna Magda S. Silva, Rossana M. de C. Andrade, and Ticianne de Gois R. Darin. Design and Evaluation of Mobile Applications for People with Visual Impairments: A Compilation of Usable Accessibility Guidelines. In *Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems*, IHC '19, New York, NY, USA, 2019. ACM.

[129] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, Stochastic Model-Based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 245–256, New York, NY, USA, 2017. ACM.

[130] Amanda Swearngin and Yang Li. Modeling Mobile Interface Tappability Using Crowdsourcing and Deep Learning. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–11, New York, NY, USA, 2019. ACM.

[131] Sarit Felicia Anais Szpiro, Shafeka Hashash, Yuhang Zhao, and Shiri Azenkot. How People with Low Vision Access Computing Devices: Understanding Challenges and Opportunities. In *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '16, page 171–180, New York, NY, USA, 2016. ACM.

[132] Christopher Vendome, Diana Solano, Santiago Liñán, and Mario Linares-Vásquez. Can everyone use my app? an empirical study on accessibility in android apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–52, 2019.

[133] Christopher Vendome, Diana Solano, Santiago Liñán, and Mario Linares-Vásquez. Can Everyone use my app? An Empirical Study on Accessibility in Android Apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–52, 2019.

[134] Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. Vet: identifying and avoiding UI exploration tarpits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 83–94, Athens Greece, August 2021. ACM.

[135] Yixuan Wei, Han Hu, Zhenda Xie, Ze Liu, Zheng Zhang, Yue Cao, Jianmin Bao, Dong Chen, and Baining Guo. Improving CLIP Fine-tuning Performance. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, October 2023.

[136] E. Weisstein. Heron's formula. *MathWorld*, 2003.

[137] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. Screen parsing: Towards reverse engineering of ui models from screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, page 470–483, New York, NY, USA, 2021. Association for Computing Machinery.

[138] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. Screen Parsing: Towards Reverse Engineering of UI Models from Screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, page 470–483, New York, NY, USA, 2021. ACM.

[139] Shunguo Yan and P. G. Ramachandran. The Current Status of Accessibility in Mobile Apps. *ACM Trans. Access. Comput.*, 12(1), feb 2019.

[140] Yanfu Yan, Nathan Cooper, Kevin Moran, Gabriele Bavota, Denys Poshyvanyk, and Steve Rich. Enhancing Code Understanding for Impact Analysis by Combining Transformers and Program Dependence Graphs. In *The ACM International Conference on the Foundations of Software Engineering (FSE)*, July 2024.

[141] Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *ICSE*, pages 250–265. Springer, 2013.

[142] Jiarui Yu, Haoran Li, Yanbin Hao, Bin Zhu, Tong Xu, and Xiangnan He. CgT-GAN: CLIP-guided Text GAN for Image Captioning. In *Proceedings of the 31st ACM International Conference on Multimedia*, MM '23, page 2252–2263, New York, NY, USA, 2023. Association for Computing Machinery.

[143] Xiao (Cosmo) Zhang, Kan Fang, and Gregory Francis. Optimization of switch keyboards. In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '13, New York, NY, USA, 2013. Association for Computing Machinery.

[144] Xiao (Cosmo) Zhang, Kan Fang, and Gregory Francis. Optimization of Switch Keyboards. In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '13, New York, NY, USA, 2013. ACM.

[145] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, Aaron Everitt, and Jeffrey P Bigham. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. ACM.

[146] Yixue Zhao, Saghar Talebipour, Kesina Baral, Hyojae Park, Leon Yee, Safwat Ali Khan, Yuriy Brun, Nenad Medvidović, and Kevin Moran. Avgust: automating usage-based test generation from videos of app executions. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 421–433, New York, NY, USA, 2022. ACM.

[147] Yixue Zhao, Saghar Talebipour, Kesina Baral, Hyojae Park, Leon Yee, Safwat Ali Khan, Yuriy Brun, Nenad Medvidović, and Kevin Moran. Avgust: automating usage-based test generation from videos of app executions. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 421–433, New York, NY, USA, 2022. Association for Computing Machinery.

[148] Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 5551–5560, 2017.