Automated and Context-Aware Repair of Color-Related Accessibility Issues for Android Apps

Yuxin Zhang
College of Intelligence and Computing, Tianjin University
Tianjin, China
yuxinzhang@tju.edu.cn

Sen Chen*
College of Intelligence and Computing, Tianjin University
Tianjin, China
senchen@tju.edu.cn

Lingling Fan
College of Cyber Science, Nankai University
Tianjin, China
linglingfan@nankai.edu.cn

Chunyang Chen
Monash University
Australia
chunyang.chen@monash.edu

Xiaohong Li
College of Intelligence and Computing, Tianjin University
Tianjin, China

ABSTRACT

Approximately 15% of the world's population is suffering from various disabilities or impairments. However, many mobile UX designers and developers disregard the significance of accessibility for those with disabilities when developing apps. It is unbelievable that one in seven people might not have the same level of access that other users have, which actually violates many legal and regulatory standards. On the contrary, if the apps are developed with accessibility in mind, it will drastically improve the user experience for all users as well as maximize revenue. Thus, a large number of studies and some effective tools for detecting accessibility issues have been conducted and proposed to mitigate such a severe problem.

However, compared with detection, the repair work is obviously falling behind. Especially for the color-related accessibility issues, which is one of the top issues in apps with a greatly negative impact on vision and user experience. Apps with such issues are difficult to use for people with low vision and the elderly. Unfortunately, such an issue type cannot be directly fixed by existing repair techniques. To this end, we propose Iris, an automated and context-aware repair method to fix the color-related accessibility issues (i.e., the text contrast issues and the image contrast issues) for apps. By leveraging a novel context-aware technique that resolves the optimal colors and a vital phase of attribute-to-repair localization, Iris not only repairs the color contrast issues but also guarantees the consistency of the design style between the original UI page and repaired UI page. Our experiments unveiled that Iris can achieve a 91.38% repair success rate with high effectiveness and efficiency. The usefulness of Iris has also been evaluated by a user study with a high satisfaction rate as well as developers' positive feedback. 9 of 40 submitted pull requests on GitHub repositories have been accepted and merged into the projects by app developers, and another 4 developers are actively discussing with us for further repair. Iris is publicly available to facilitate this new research direction.

*Corresponding author

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616329

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Maintaining software**.

KEYWORDS

Mobile accessibility, Accessibility issue repair, Color-related accessibility issue, Android app

ACM Reference Format:
Yuxin Zhang, Sen Chen, Lingling Fan, Chunyang Chen, and Xiaohong Li. 2023. Automated and Context-Aware Repair of Color-Related Accessibility Issues for Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616329

1 INTRODUCTION

Nowadays, mobile applications (apps) are ubiquitous [17, 18, 82, 84]. In addition to providing various functional services for users, the importance of mobile accessibility has gained increasing attention from both industry and academia [14, 68, 75, 79]. Mobile accessibility refers to making apps more accessible to people with disabilities when they are using mobile phones [76]. Besides its special significance for the disabled, if the developers design a mobile app that has more accessible features such as screen readers (TalkBack for Android [78], VoiceOver for iOS [6]), they will be able to reach a larger number of audiences. To this end, governments have established laws to help eliminate barriers in electronic and information technology for people with disabilities [42, 43] and leading IT companies (e.g, Google, Apple, Microsoft, Meta, and IBM) have established their accessibility teams to improve app accessibility [5, 27, 35, 46, 57].

In recent years, a large number of empirical studies have been conducted to investigate the characteristics of app accessibility [3, 16, 23, 58, 63, 64, 75, 80]. These studies unveiled that almost all apps are suffering from accessibility issues [3, 16, 49]. To mitigate such a severe problem, a series of effective automated approaches for detecting app accessibility issues have been proposed such as Android Lint [33], Espresso [39], Robolectric [41], Google Accessibility Scanner [32], Google Accessibility Testing Framework (ATF) [38], MATE [25], LabelDroid [15], AccessiText [4], Latter [66], Groundhog [67], and Xbot [16]. Yet, too many accessibility issues make it difficult to effectively repair in practice. For example, the result in [16] shows that there are 40 issues for each app and 6.5 issues for each page on average. In addition to the bottlenecks from a large number of issues, the various categories of issues further limit repair efficiency. In other words, it is unrealistic for app developers to repair these issues within apps even with substantial human effort (also proved by the developers' feedback in § 4.3).

To address this problem, researchers tried to fix these issues by leveraging automated repair approaches, but such effort is in its infancy. Specifically, Alotaibi et al. [1] proposed a genetic algorithm guided by a fitness function to automatically repair size-based accessibility issues in apps. Moreover, the issues related to item labels (i.e., missing content labels) can be mitigated by different existing approaches such as social annotation techniques [83], deep learning algorithms [15, 56], and crowd-sourcing techniques [12]. However, in addition to the above two issue categories with a large proportion, the issues of text or image contrast are also very serious, which is one of the most prevalent accessibility issues that affect mobile apps [3, 13, 16, 64]. As shown in Figure 1, text or image contrast, also known as *color-related accessibility issues*, occurs when the color contrast between the text/image and the background is less than the minimum ratio specified by the accessibility guidelines [13, 77]. Such issues make the apps difficult to use, not only for people with low vision but also for all users. Linares-Vásquez et al. [48] proposed a method to generate a brand-new color scheme for the UI, intending to reduce the energy consumption of the GUI in Android apps. Although it also changes the color of the UI page, it is completely different from its original color scheme, and the scenario of their work is completely different from this paper.

There are also techniques for repairing accessibility issues of web pages [50–53, 60]. However, they focused more on Mobile Friendly Problems, such as Font sizing, Tap target spacing, Content sizing, Viewport configuration (i.e., sizing issue), and Flash usage (i.e., rendering issue), which can inspire the repair of size-related issues, however, cannot benefit color-based accessibility issues. Moreover, to help meet contrast requirements on web pages, several works [45, 69] focused on recommending color pairs by simply adjusting the color values of texts, however, they did not pay attention to the original design style of the web page. Additionally, the implementation mechanisms are significantly different for web apps and Android apps, which directly distinguish repair solutions.

Thus, it is actually a non-trivial task due to the following challenges: **C1:** The color-related changes should ensure to maintain the consistency of the design style between the original UI pages and repaired UI pages. **C2:** The UI components with issues and their corresponding attribute to be modified need to be accurately located and determined. Meanwhile, it should be determined whether the involved image files that are relevant to the image contrast issues really need to be modified. **C3:** The repaired results need to be evaluated and confirmed by real users and developers.

To address these above challenges, we proposed Iris, an automated and context-aware approach to repa<u>i</u>ring the colo<u>r</u>-related

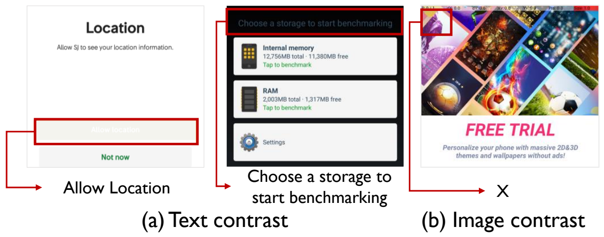

Location
Allow SJ to see your location information.
Allow location
Not now
Allow Location
Choose a storage to start benchmarking
Internal memory
12,756MB total · 11,380MB free
Tap to benchmark
RAM
2,003MB total · 1,317MB free
Tap to benchmark
Settings
Choose a storage to start benchmarking
FREE TRIAL
Personalize your phone with massive 2D&3D themes and wallpapers without ads!
X
(a) Text contrast
(b) Image contrast

Figure 1: Examples of color-related accessibility issues in apps.

access<u>i</u>bility i<u>s</u>sues for Android apps. Specifically, to address **C1**, we design a novel context-aware technique that resolves the optimal colors by leveraging the well-designed criteria for color value resolving and the color reference DB construction. The context-aware technique can ensure the design style consistency between the repaired UI pages and original UI pages for apps. For **C2**, taking the selected colors as input, we use an attribute-to-repair localization method by analyzing the source code of relevant layout files to determine the component attributes that should be modified. Meanwhile, the repair constraints parsing step helps to determine if the image contrast issues need to be really fixed. Based on these key phases, the issues within the apps can be effectively repaired and validated before a new repaired APK (Android Application Package) file is released. Last but not least, to address **C3**, we carry out a comprehensive and well-designed user study to help evaluate our repaired results. We also submit a number of pull requests for real GitHub projects to help improve the accessibility of their apps.

To evaluate the effectiveness, efficiency, and usefulness of Iris, we designed a series of experiments on 100 real-world apps, including both closed-source and open-source apps. The results show that Iris performs a high repair success rate of 91.38% and costs 2.27 minutes per app on average. Finally, based on the user study results and the developers' feedback, we highlight that Iris can really help developers fix color-related accessibility issues and practically improve the app accessibility. The user study results also show that the consistency of the original design style is well maintained from the perspective of both the UI page and the app level. Till now, we have submitted 40 pull requests on GitHub repositories and 9 projects have merged our repaired results, and another 4 developers are actively discussing with us for further repair.

In summary, we make the main contributions as follows.

- To the best of our knowledge, Iris is the first automated approach proposed to repair color-related accessibility issues for Android apps. We make the tool and relevant data public available on GitHub,[1] to facilitate new research areas for improving app accessibility.
- We propose a novel context-aware technique that resolves the optimal colors to ensure the consistency of design style between the required UI pages and the original UI pages. Additionally, a color reference DB collected from 9,978 apps has been constructed and released to help resolve the optimal color value for the color selection.

[1] https://github.com/iris-mobile-accessibility-repair/iris-mobile

```
1  // Implemented by XML code
2  <TextView android:layout_width="fill_parent"
3                  android:layout_height="wrap_content"
4                  android:textColor="#80ff0000"
5                  android:text="Hello"/>
6  <Button android:background="#80ff0000"/>
7
8  // Implemented by Java code
9  TextView tv = (TextView)this.findViewById(R.id.tv);
10 tv.setTextColor(0xff000000);
```

Figure 2: Examples of color-related layout implementation.

```
// Details of detection report
Text contrast: "a2dp.Vol:id/pi_tv_name"
The item's text contrast ratio is 1.04. This ratio is based on an estimated
foreground color of #FFFFFF and an estimated background color of #FAFAFA.
Consider increasing this item's text contrast ratio to 3.00 or greater.
// Part of layout file provided by Xbot
<node bounds="[131,362][1080,493]" selected="false"
content-desc="" package="a2dp.Vol" class="android.widget.TextView"
resource-id="a2dp.Vol:id/pi_tv_name"
text="A2DP Volume" index="1"/>
```

Figure 3: The example of report generated by Xbot [16].

- The experiments on 100 real-world apps including both closed-source apps and open-source apps demonstrate the effectiveness and efficiency of our approach. A well-designed user study clearly demonstrates the usefulness of our approach. Moreover, the positive feedback from real developers has also highlighted the practicality of Iris.

2 PRELIMINARY

2.1 Color-related Accessibility Issue

As shown in Figure 1, color-related accessibility issue includes two types: *text contrast issue* and *image contrast issue*. The former corresponds to visible text, where there is a low contrast ratio between the text color and background color. The latter refers to images with a low contrast ratio between the foreground and background colors. The visual presentation of text and text images has a contrast of at least 4.5:1 (text below 18 point regular or 14 point bold), while the contrast of large-scale text (18 point and above regular or 14 point and above bold) and large-scale text image is at least 3:1 (required by Web Content Accessibility Guidelines (WCAG) [77] and Google Accessibility Guidelines for Android [37]). These two types of issues frequently occur in apps [3, 13, 16] and significantly reduce the app accessibility. Specifically, the examples shown in Figure 1 have a big visual problem even for all people, not just for people with disabilities (e.g., people with low vision and the elderly).

2.2 Color-related Layout Implementation

There are 2 main ways to implement the color setting of UI components [34]. As shown in Figure 2, we can use XML layout code to set the color property for different UI components. For example, we can use "android:textColor" (Line 4) to draw the color for the text of *TextView* and "android:background" (Line 6) to config the color for the background of *Button*. The same functionalities can be completely replaced by Java code by using the corresponding APIs like *#setTextColor()* (Line 10).

2.3 Reports Collected from Detection Tools

The input required for repair needs to be obtained from the Google official accessibility testing framework (ATF) [38], a library collecting variable accessibility-related checks on View objects as well as AccessibilityNodeInfo objects. Among the existing accessibility issue detection tools using ATF [4, 25, 32, 39–41, 66, 67], Xbot [16], which is a fully automated approach for detecting all types of accessibility issues based on the ATF and Google Accessibility Scanner [32], has the ability to explore the app UI with high activity coverage, and can effectively and efficiently collect a relatively comprehensive dataset of accessibility issues. Therefore, we finally choose Xbot as our issue detection and collection tool, which takes as input an APK file and outputs its exploration and detection results. Figure 3 is a partial example of the detection report and layout file provided by Xbot. The detection report prompts the type of the accessibility issue (i.e., text or image contrast), the unique identification of the component (i.e., *resource-id* and *node bounds*), and the specific information of the issue. Moreover, Xbot also provides the rendered UI screenshot for each activity.

2.4 Default Solution in Android

Android OS provides support for addressing color-related accessibility issues [31]. The repair strategy is that the system setting of "High Contrast Text" will change the UI components of used apps to black or white according to the original color, trying to make the text on the device easier for the user to read. However, after this setting is turned on, the text of all colors in the app will be changed into white or black, which will completely change the design style of the UI pages.

3 APPROACH

Figure 4 shows an overview of Iris, which takes as input an APK file along with the issue reports, and outputs a repaired APK file without color-related accessibility issues. There are no special restrictions on the input APK, while the required reports are provided by ATF [38]. Iris consists of three main phases: ***(1)*** *Reference DB construction*, which analyzes the UI components without color-related issues from a large number of detection results, and constructs a reference database to further help select optimal color. ***(2)*** *Context-aware color selection*, which is a novel context-aware technique that resolves the optimal value of the color replacement through two strategies based on our well-designed criteria for color value resolving. ***(3)*** *Attribute-to-repair localization*, which is used to locate the position of relevant UI components and further determine the attributes of components that need to be modified. After that, Iris replaces the attributes of the problematic UI component with the resolved optimal color value and updates the corresponding layout files or source code to repair the app.

3.1 Reference DB Construction

To provide reference values for selecting the optimal value of the replacement colors, the goal of this phase is to construct a

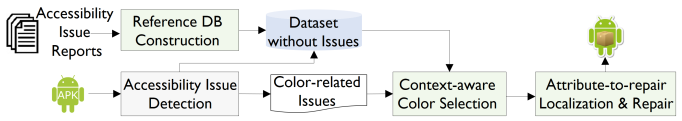

Accessibility Issue Reports
Reference DB Construction
Dataset without Issues
APK
Accessibility Issue Detection
Color-related Issues
Context-aware Color Selection
Attribute-to-repair Localization & Repair

Figure 4: Overview of Iris.

reference database containing a dataset without color-related issues (i.e., meet the requirements of standard color contrast). *We highlight the replacement color selected from the database that has been used in real apps and accepted by the app designers.*

Based on our investigation of the detection reports, we notice that if we only consider the color used by the app itself, it may be difficult to find a replacement color that can be applied, since most UI components in an app generally share the same color value. Therefore, it is essential to construct a database composed of multiple APKs instead of only relying on an app itself. Meanwhile, we investigated that different types of UI components (e.g., *Button, EditText,* and *TextView*) have different display styles. Therefore, the influence of the category of components needs also be thought about in the reference database construction. Based on these primary investigations, ***(1)*** Owing to a significant number of accessibility issue reports detected by Xbot [16], we first collect all UI components without contrast issues for all selected APKs. Meanwhile, we categorize this dataset by the UI component type. ***(2)*** In a similar manner, for each APK, we also collect the UI components without issues.

The purpose of building the database is to obtain the color pairs composed of the foreground color and background color of each UI component. For the component dataset without issues, the reports do not involve the specific value of the color pair of UI components, so we need to make further efforts to compute the value of the color pairs. In this process, we extract the values of the key foreground and background colors from the screenshot of each UI component. The color composition of a single component is relatively simple and usually consists of only the background color and the color of the text or image on it. Therefore, we use *#getcolors()* [61] in the image module to return the two most used colors in the screenshot. After judging that the result meets the required contrast, the color pair is returned as a replacement value for reference.

3.2 Context-aware Color Selection

Given a component with color-related accessibility issues, the goal of this phase is to determine the optimal replacement color for components, which do not violate the standard color contrast on the premise of maintaining the style of the original design as much as possible (i.e., context-aware). *This is also the innovation of our approach, that is, to ensure style consistency at all design levels.* When resolving the optimal replacement color, two aspects need to be considered: ***(1)*** For a component with low contrast, how to decide whether to change the foreground color or background color, or both? ***(2)*** What is the strategy of color selection and what are the criteria for color value resolving under the selection strategy?

For question ***(1)***, many UI components usually share an area with the same background color or call the same resource defining the background color, on one page of an app. Therefore, choosing

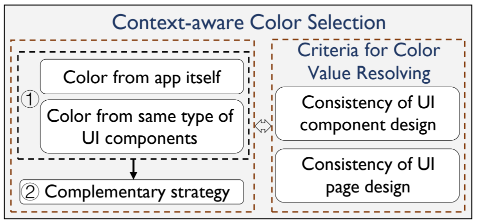

Context-aware Color Selection
①
Color from app itself
Color from same type of UI components
② Complementary strategy
Criteria for Color Value Resolving
Consistency of UI component design
Consistency of UI page design

Figure 5: The workflow of context-aware color selection.

to modify the value of the background color is too easy to have a chain reaction, which may change the color contrast between the foreground and background of other UI components, or even lower than the standard value, and introduce new accessibility issues. The risk caused by this modification is high. Compared with the background color, the foreground color is relatively independent, usually expressed as the color of the text in the component or the main color of a picture. When the foreground color changes, it would probably not trigger changes in other parts of the UI page. Therefore, our approach takes priority to modifying the foreground color of the problematic component with the goal of obtaining the optimal solution for the replacement color.

After determining the target to be modified, the next problem to be solved is to find the replacement color required for modification. The most important thing is to keep the original UI style unchanged. Therefore, for question ***(2)***, as shown in Figure 5, we propose two strategies of color selection.

3.2.1 Color Selection Strategy. Two situations: ① the color replacement based on the reference DB. Through this strategy, we can get the replacement color from the app itself or the dataset filtered by the same component types in other apps. Specifically, if we can find alternative colors from the app itself, we can directly use the colors defined and used by this app itself for replacement, which is more in line with the color selection intention of the app designers. If no suitable replacement color can be found in the app itself, we will consider the same component types collected from other apps. Because the same types of components probably share similar design styles. ② Complementary strategy will be applied if we cannot find a replacement color from the reference DB. This strategy chooses the replacement color by directly modifying the original foreground color for the problematic component while maintaining the design style of the UI component and the coordination of the UI page.

3.2.2 Criteria for Color Value Resolving (Context-aware). Note that the replacement color matched from the reference DB by using the background color of the problematic component is a set of color candidates. Meanwhile, a reference is also required to directly modify the value of the original color when using the complementary strategy. Thus, to maintain the original design style of the app as much as possible, we define two criteria for color value resolving. ① Consistency of UI component design: the hue and saturation level are consistent with the original color of UI component. ② Consistency of UI page design: the hue is more harmonious with the overall hue of the UI page.

Consistency of UI component design. From the perspective of component design, the color matching of components represents the designer's intention. UI designers usually first consider the hue of components when setting colors for them, such as the color of red or blue. In addition, colors with the same hue but different saturations also have great visual differences. Actually, various saturations produce a variety of visual impacts and attractions. For example, a color with high saturation is bright, which can make the main body stand out from the background, while a low saturation color can give people a low-key and subtle feeling. Therefore, when considering the designer's intention to the greatest extent and ensuring that the replacement color is relatively consistent with the original color, it is crucial that the hue and saturation level of the replacement color and the original color should be consistent. We use HSV (hue-saturation-value) [59] to meet such a criterion since the HSV color model is consistent with the way humans describe colors, and allows independent control of hue, saturation, and intensity (value). The HSV values of colors can be calculated using their RGB values by formulas (1)~(3) [71].

$$H = \begin{cases} 0, \ Max = Min \\ 60 \times \frac{G-B}{Max-Min}, \ Max = R \\ 60 \times \frac{B-R}{Max-Min} + 120, \ Max = G \\ 60 \times \frac{R-G}{Max-Min} + 240, \ Max = B \end{cases} \tag{1}$$

$$S = \begin{cases} 0, \ Max = 0 \\ \frac{Max-Min}{Max}, \ Max \neq 0 \end{cases} \tag{2}$$

$$V = Max \tag{3}$$

where R, G, and B represent red, green, and blue values of the RGB of one color, and "Max" and "Min" represent the maximum and minimum values between R, G, and B values. These formulas operate on values in the form of decimal numbers. Based on this standard, according to the Practical color coordinate system (PCCS) [44, 65], we divide the calculated saturation value from 0 to 1 into three saturation levels: low (0~0.33), medium (0.34~0.67), and high (0.68~1). When filtering the replacement color values, we only retain the candidate values with the same hue and saturation level as the original color.

Consistency of UI page design. As shown in Figure 6, when considering the color coordination degree of the whole UI page, we leverage eight harmonic types defined on the hue channel of the HSV color wheel as the second criterion [22]. Each type shows the hue color distribution in the harmonic template (the size of the gray area is fixed, but the position is not fixed, and it can rotate around the center of the circle). In other words, if all the hues of a UI page fall in the gray area of a certain harmonic type, the color replacement of the page is considered to be harmonic.

Using the two strategies with two criteria defined in Figure 5, we can get the final optimal color for replacement. As shown in Algorithm 1, before calculating the optimal replacement color value, Iris will obtain the set of colors available for replacement from the reference DB ($ColorSet_{ref}$) and calculate the optimal harmonic type and deflection angle corresponding to the UI page ($best_T$ and $best_{Alpha}$). After obtaining the inputs, we calculate the HSV value of the original color first (Line 1). Then, by judging whether the hue and saturation level of the replaceable candidate color is consistent with the original color (*Consistency of UI component design*), a *ColorSet* (Line 2~Line 5) that meets the consistent color hue and saturation standard is selected from the set of replaceable candidate

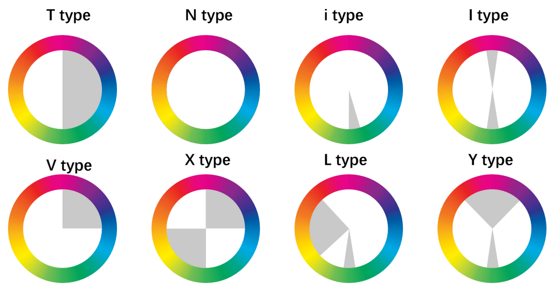


Figure 6: Eight harmonic types for color value resolving.

colors. If the *ColorSet* is not empty, the distance between each color in the *ColorSet* and the shadow part of the optimal harmonic type is calculated (Line 8). The smaller the distance is, the more consistent the hue consistency standard of the UI page is *(Consistency of UI page design)*. The color with the smallest distance is selected from the *ColorSet* as the optimal replacement color (Line 10).

If the *ColorSet* is empty, it means no appropriate color can be selected for replacement from the existing reference DB. To find a suitable replacement color, we use the complementary strategy to directly modify the HSV value of the original color. If the original color is black, white, gray, and other achromatic system colors (neutral colors), since there is only a difference in brightness between them, it is only necessary to adjust the brightness (V) value of the original color up and down until the changed color meets the standard color contrast (Line 13). At this time, since the changed color is still neutral, the hue (H) channel of the HSV color wheel of the whole UI page will not be affected *(Consistency of UI page design)*. The saturation (S) value of the original color is also not changed. But if the original color belongs to the color system (not neutral colors), adjust the value of H and S on the premise that the hue and saturation level is consistent with the original color *(Consistency of UI component design)*: (1) Adjust the value of H first, and then adjust the value of S to obtain the replaceable value $color_{HS}$ (Line 15). (2) Adjust the value of S first, and then adjust the value of H to obtain $color_{SH}$ (Line 16). Finally, the color closest to the original color is selected as the optimal replacement color among the two replaceable colors (Line 17).

3.3 Attribute-to-repair Localization

After obtaining the optimal color for repairing UI components, this phase aims to localize the components and determine the attributes to be repaired. Although the input detection report contains specific information about accessibility issues, the specific location and the attributes that need to be modified cannot be directly determined. There are several challenges. ***(1)*** First of all, color-related accessibility issues include two types of problems. These two types are different in repair objects and repair methods. For example, for text contrast issues, we need to repair the color attribute of the text in the UI components. For the issues of image contrast, we need to replace or modify the involved images. ***(2)*** Secondly, even for the same type of issues, the attributes and the repair conditions to be modified will be different. ***(3)*** In addition, how to decide the location of the layout code (or source code) of the relevant components in

Algorithm 1: Context-aware Color Selection Algorithm

```
Input: $Color_{org}$: The problematic color with contrast issue
       $ColorSet_{ref}$: The color candidates selected from reference DB
       $best_T$: The optimal harmonic type of the corresponding UI page
       $best_{Alpha}$: The deflection angle of the optimal harmonic type
Output: $Color_{opt}$: The selected optimal color
1  $H_0, S_0, V_0 \leftarrow getHSV(Color_{org})$
2  foreach $c \in ColorSet_{ref}$ do
3      $H, S, V \leftarrow getHSV(c)$
4      if $isConsistentHue(H_0, H)$ and $isConsistentSaturation(S_0, S)$ then
5          $ColorSet$.append($c$)
6  if $ColorSet$ is not null then
7      foreach $c \in ColorSet$ do
8          $d \leftarrow getDistance(best_T, best_{Alpha}, c)$
9          $DisSet[c] = d$
10     $Color_{opt} \leftarrow minDistance(DisSet)$
11 else
12     if $isNeutralColor(Color_{org})$ then
13         $Color_{opt} \leftarrow adjustV(V_0, Color_{org})$
14     else
15         $Color_{HS} \leftarrow adjustHS(H_0, S_0, Color_{org})$
16         $Color_{SH} \leftarrow adjustSH(H_0, S_0, Color_{org})$
17         $Color_{opt} \leftarrow minChanged(Color_{org}, Color_{HS}, Color_{SH})$
18 return $Color_{opt}$
```

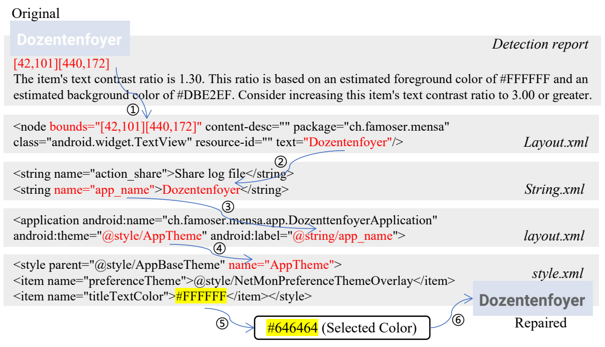
the UI layout files of the app through the existing detection report is also a big challenge. In this phase, Iris treats these two types of issues separately and decides the attributes to be modified by analyzing both the layout code and the source code.

3.3.1 *Localization of the related UI components.*

The input report contains two types of tips about the information of the components: "Component ID" and "Bounds" of a component in the layout. Note that the layout file here refers to the layout file gained by Xbot, denoted by *Xbot-Layout*, which is different from those obtained by decompiling the APK file, denoted by *Decompile-Layout*. To localize the relevant components to be repaired, we analyze these two different types of information. ***(1)*** For component ID, since ID is unique to objects, we can directly locate the property set of the corresponding component in the *Decompile-Layout* (or source code) according to the component ID (Figure 8). ***(2)*** For the bounds of a component, the report displays Bounds instead of ID because Xbot sometimes failed to detect the component ID. Meanwhile, the attribute "bounds" does not exist in the *Decompile-Layout* files. Thus, to locate the relevant components, we can find the text information of the component according to the bounds in *Xbot-Layout*, and choose to match through the *Android:text* attribute in *Decompile-Layout* (the steps 1~3 in Figure 7).

3.3.2 *Acquisition of the related attribute-to-repair.*

Iris uses static data-flow analysis to extract relevant attribute sets shown in Figure 7 and Figure 8. Firstly, Iris parses the app to extract the layout files and *Smali code*. Then, through the above analysis, Iris completes the localization of the relevant components and obtains the attribute sets for each component. For the two types of accessibility issues, the attributes-to-repair are different. ***(1)*** For text contrast issues, the components are generally *TextView*, *EditText*, and *Button*, which are prone to such problems. At this time, the



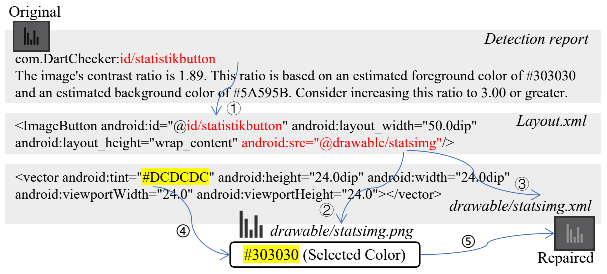
```
[42,101][440,172]
The item's text contrast ratio is 1.30. This ratio is based on an estimated foreground color of #FFFFFF and an
estimated background color of #DBE2EF. Consider increasing this item's text contrast ratio to 3.00 or greater.

<node bounds="[42,101][440,172]" content-desc="" package="ch.famoser.mensa"
class="android.widget.TextView" resource-id="" text="Dozentenfoyer"/>

<string name="action_share">Share log file</string>
<string name="app_name">Dozentenfoyer</string>

<application android:name="ch.famoser.mensa.app.DozenttenfoyerApplication"
android:theme="@style/AppTheme" android:label="@string/app_name">

<style parent="@style/AppBaseTheme" name="AppTheme">
<item name="preferenceTheme">@style/NetMonPreferenceThemeOverlay</item>
<item name="titleTextColor">#FFFFFF</item></style>

#646464 (Selected Color)
```

Figure 7: Example of attribute-to-repair localization for text contrast.

foreground color refers to the color of the text in the component, and the background color is the color of the component background. Therefore, for such issues, the object to be repaired is mainly the color of the text in these components. However, since the text in *EditText* is mainly used for input hints and the text color is generally light, the color contrast of the component does not have to meet the standard requirements, and such components need to be filtered out in the localization stage. To conclude, the main attributes involved include *Android:textcolor, Android:textcolorlink, Android:titletextcolor*, and other attributes, or set the attributes in the style file accordingly (the step 5 in Figure 7). Figure 7 shows an example of attribute-to-repair localization for text contrast. ***(2)*** For image contrast issues, the frequently occurring components mainly include *ImageButton* and *ImageView*. At this time, the issue with low contrast is the contrast between the image color and the background color of the component. Therefore, to prevent the impact on other components in the UI, we choose the image in the component as the repair object. The first step is to get the image files that need to be repaired. Among them, the source file of an image can be directly associated with the UI component through the attribute definition in the layout file. For example, as shown in Figure 8, images can be associated by setting the property values of *Android:src* or *Android:background* related to the components. *Android:src="@drawable/statsimg"* can be mapped directly to the file name in the resource folder (*res/drawable/statsimg.png*). Figure 8 shows an example of attribute-to-repair localization for image contrast. At this time, the goal of repair is to change the color setting of the image in the component. However, some images are actually not suitable for modification, so we need to filter out them and conduct repairing, which is elaborated as follows.

3.3.3 *Repair constraints of images.*

To distinguish the images suitable to be repaired, we divide these images into "*functional images*" and "*ornamental images*" according to the display intention. We choose to modify the images or icons that focus more on using functions, such as return, add, cancel, and share icons, or images representing these meanings, which we refer to as "functional images". For the "ornamental images", if we change the color of the image, the display function of the image may be different from before. Therefore, we only choose to repair the "functional images". Iris distinguishes these two types of images by a lightweight



```
com.DartChecker:id/statistikbutton
The image's contrast ratio is 1.89. This ratio is based on an estimated foreground color of #303030
and an estimated background color of #5A595B. Consider increasing this ratio to 3.00 or greater.
```

```
<ImageButton android:id="@id/statistikbutton" android:layout_width="50.0dip"
android:layout_height="wrap_content" android:src="@drawable/statsimg"/>
```

```
<vector android:tint="#DCDCDC" android:height="24.0dip" android:width="24.0dip"
android:viewportWidth="24.0" android:viewportHeight="24.0"></vector>
```

Figure 8: Example of attribute-to-repair localization for image contrast.

static analysis technique that determines whether the images are associated with event handler methods. The trigger behavior of a component is usually associated with the "clickable" attribute in the layout file. Therefore, we judge whether the corresponding image is classified as a "functional image" by checking the relevant attributes of the problematic component. Then we execute the optimal color selection algorithm and change the color composition of the original image by changing the value of the color model [47] of the pixel of the image, such as RGB [73] and RGBA [74], which applies to vector images in resources and other types of user-uploaded non-vector images such as PNG or JPG.

After obtaining the replacement color and locating the attribute to be modified, Iris will directly replace the original values that have issues, including the replacement of color values and image files. Iris then repackages the replaced file to get a new APK. Moreover, during repair, we also consider many other aspects such as the judgment of foreground and background color, to solve the possible detection errors of foreground and background color in the detection reports. Sometimes, the set of the foreground color and the background color is reversed in the detection process, such as when the detection report indicates the foreground color as #298670 and the background color as #EDF064, when in reality, they are exactly the opposite. Before repair, we capture screenshots of the UI pages and extract the color information from them to ascertain the true color composition of each component.

4 EXPERIMENTS

To make the experiments we designed better evaluate our approach, we raise the following questions:

- **RQ1:** How effective and efficient is Iris in repairing the color-related accessibility issues?
- **RQ2:** How much does each key strategy of Iris contribute to the overall performance?
- **RQ3:** How useful is Iris from the perspective of mobile users and app developers?

Dataset. For reference DB construction, we randomly collected 9,978 real apps, including 5,081 open-source apps from F-Droid [26] and 4,897 closed-source apps from Google Play. For RQ1 and RQ2, considering the time cost of testing, we randomly selected 100 apps with an average size of approximately 6MB as experimental subjects, of which the number of open-source apps and closed-source apps is 50 and 50, respectively, and used Iris to automatically repair them.

Table 1: Results for effectiveness evaluation of Iris.

Issue Type	# Real Issues	# Repaired Issues	Success Rate (%)
Text Contrast	660	618	93.6
Image Contrast	71	50	70.4
Total	731	668	91.38

It should be noticed that these 100 apps do not appear in the dataset (i.e., 9,978) used in reference DB construction.

4.1 RQ1: Effectiveness and Efficiency Evaluation

4.1.1 ***Setup.*** To answer RQ1, we compare the detection results of 100 apps before and after repair. First, we use the Xbot [16] tool to conduct a preliminary detection on the tested apps and count the color-related accessibility issues of those apps before repair. Then, we take the detection results and APK files as input, use the Iris to repair the color-related accessibility issues of these apps one by one, repackage them, and output new APK files. To get the results after repair, we use the Xbot tool to detect the repaired APK files again and obtain the detection results. We evaluate the effectiveness of Iris by comparing the number of repaired issues and the issues in the same original app. The success rate for the i^{th} app mentioned here is denoted by *RepairR*.

$$RepairR_i = \frac{N_i^{Repaird_{issue}}}{N_i^{All_{issue}}} \times 100\% \tag{4}$$

Additionally, to evaluate the efficiency of Iris, we record the execution time for these 100 apps and compute the average time to demonstrate the performance.

4.1.2 ***Result.*** The results of the effectiveness evaluation (RQ1) are shown in Table 1. The column "# Real Issues" represents the number of issues of the text and image contrast contained in all 100 apps. Note that, we removed a part of issues when counting the number of real issues for repairing. The removing parts include ***(1)*** the *EditText* components as we mentioned in § 3.3.2, ***(2)*** the false positive cases caused by wrong screenshots in the detection reports, and ***(3)*** the causes altered by the system design style (the problem also introduced in [3]). Finally, we have 660 real issues with text contrast and 71 real issues with image contrast, which owns the most issues among all the issue types, accounting for 30.17%. Remarkably, because a component, uniquely marked by ID, may be applied to different pages or the same page multiple times, there may be two or three issues caused by the same component. At this time, the property only needs to be modified once, which belongs to components with the same ID. As for text contrast, Table 1 shows that the number of issues of 100 apps before the repair is 660. Among them, there are 618 issues that have been repaired successfully, accounting for a 93.6% success repair rate and including 413 different UI components. The repair rate unveils that our tool can effectively reduce the number of text contrast issues. The root causes of the remaining 42 unresolved issues are as follows. ***(1)*** For some components, although we have located their locations and initial attribute sets during the repair process and have tried to repair them, the user's actions may cause the components to

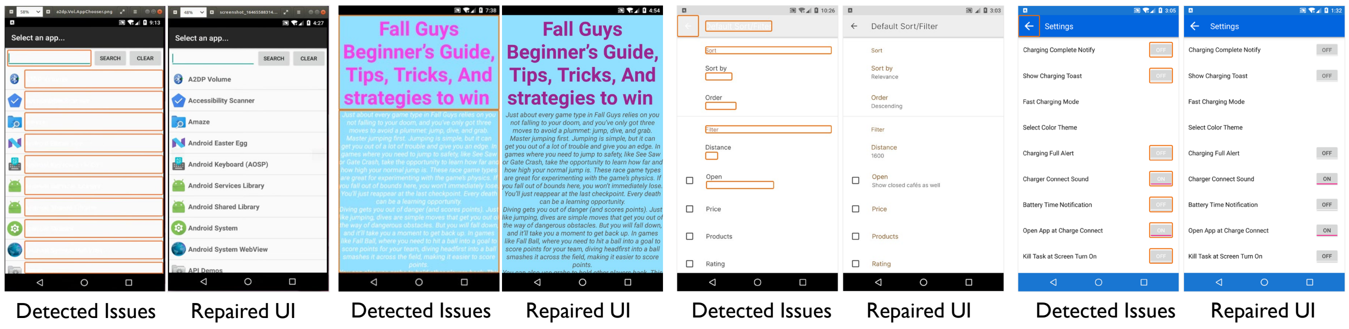


Figure 9: Examples of repaired issues.

Table 2: Evaluation of the key phrase (context-aware color selection).

Category	# Repaired issues by reference DB	# Repaired issues by direct modification
Text Contrast	569	49
Image Contrast	39	11
Total	608	60

appear in different states during the running of the app, failing the repair. Because the color rendering is implemented in complex source code supported by new reconstructed API interfaces of third-party libraries, rather than the official APIs introduced in Figure 2. ***(2)*** Errors are due to the shortcomings of using bounds to locate attributes. In the process of issue detection, the detection tool cannot obtain the ID of some components, so we use the *text* information of the component according to the *bounds* to achieve positioning, and the positioning effect is not as good as using ID.

For the issues of image contrast, Table 1 shows that there are 71 image issues before repair, and 50 problems were repaired by Iris, including 35 different components. Therefore, the success rate of image contrast is 70.4%. There are still 21 issues that have not been solved for the reason of the shortcomings of using bounds to locate. Finally, the overall success rate of the two types of issues is 91.38%, and the total number of issues repaired is 668. Figure 9 presents several examples of the repair results by Iris. More examples can be found on our website [11].

Apart from evaluating the effectiveness of Iris in the number of repairs, we also record the execution time of Iris in practice. When repairing an APK, Iris will first use Xbot to detect the accessibility issues in it, update the reference DB, and then start to automatically repair. Finally, the average time of issue detection and DB updating is 100.7 seconds, and the average time of repair is 136.2 seconds. Compared with manual repair, Iris can greatly shorten the time of attribute localization and provide a feasible reference value for color replacement. Thus, Iris has high time efficiency.

4.2 RQ2: Ablation Study

4.2.1 ***Setup.*** In RQ2, we aim to evaluate the key phases (i.e., context-aware color selection and attribute-to-localization). For the localization, the accuracy of this phase is consistent with *RepairR* (i.e., 91.38%). For context-aware color selection, there are two strategies to obtain the optimal replacement color: ***(1)*** selected from reference DB; ***(2)*** direct modification based on the complementary strategy. We also use Xbot to detect the accessibility issues of APK before and after repair and investigate the number of repaired issues by each strategy.

4.2.2 ***Result.*** As shown in Table 2, for the issue of text contrast and image contrast, the proportions of the two strategies are 569:49 and 39:11, respectively. We can see that 91.02% of the issues in the selected 100 apps can be repaired by the reference DB, indicating the reference color DB plays an important role in the phase of context-aware color selection. Since the color pairs in reference DB are all from real apps, their color matching is recognized and loved by the designers and users of those apps, which also proves the rationality of our replacement. That is also the reason why we prefer to utilize the reference DB to repair issues as many as possible at first. Meanwhile, there are also some issues (i.e., 60) that cannot be repaired by the reference DB, while they can still be fixed by the complementary strategy. Although the replacement colors obtained using the complementary strategy do not appear in the apps in the existing DB, they also meet the standards, including the color contrast and two designed criteria.

4.3 RQ3: Usefulness Evaluation

To evaluate the usefulness of Iris, on the one hand, it is necessary to conduct a user study on the repaired UI page and repaired app. Meanwhile, the consistency of the design style between the original UI pages and repaired UI pages as well as the design style of the app level also should be evaluated by the validation from real users. On the other hand, the pull requests that are accepted by real app developers on GitHub will make our repair results more convincing in a real scenario. During this process, the feedback from real developers facilitates the iterative improvement of Iris.

4.3.1 ***User study from app users.*** **Dataset.** To compare the effects before and after repair from the app users' perspective, ***(1)*** we randomly selected 12 pairs of UI pages from the repaired UIs. Each pair of pages consists of the original UI and the repaired UI. Then, we divided the 12 pairs of pages into two groups as a control experiment (for *Task 1*). ***(2)*** To facilitate the scoring of participants, we randomly selected other 7 pairs of pages (for *Task 2*). In each pair, besides the original UI page and the repaired UI page, we

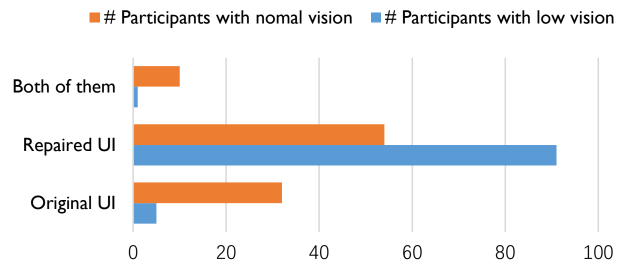


Figure 10: Participants' preference between the original UI pages and repaired UI pages.

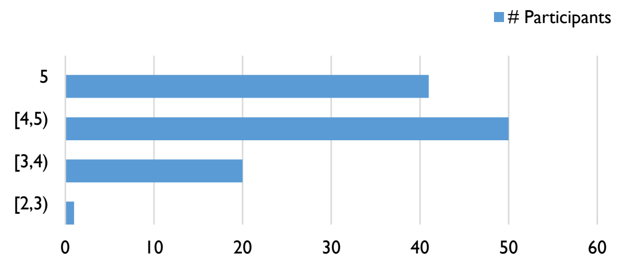


Figure 11: Distribution of participants' scores.

also added the UI page with the circled issues. ***(3)*** To evaluate the overall coordination of the design style of the repaired apps, we randomly selected 3 apps according to the different levels of the UI page number, whose UI page numbers are 4, 8, and 14 (for *Task 3*). Among them, the number of issues in each app is not less than 10.

Participant recruitment. We recruited 32 participants from our university, including 20 who are near-sighted (16 participants for *Task 1* and *Task 2* (both of them are near-sighted), and 16 participants for *Task 3*). None of them has used the repaired apps, which excludes the potential bias. The participants are from different countries, including Singapore, the United States, Germany, and China. These participants include undergraduates, postgraduates, PhD students, and staff.

Setup. For app users, we design user research from two levels (i.e., the UI level and the app level). At the UI level, we investigate the user's preference for the UI pages before and after repair in *Task 1* and the user's satisfaction with the repair effect of Iris in *Task 2*. The purpose of *Task 1* is to demonstrate the usefulness of Iris for different users. To achieve it, we divided the page screenshots before and after repair into two groups, looked for people who are nearsighted with over 2.0 diopters as participants in this part, and asked them to make preference choices for the two groups of screenshots without glasses (simulating people with special vision) and with glasses (normal vision). In this step, the order of each pair of screenshots is random and marked separately. We request participants to replicate the real-world usage conditions as closely as possible when making their selections, which aims to accurately simulate the authentic experiences of diverse user groups.

The purpose of *Task 2* is to test the repair effect of Iris through the score of participants. We showed the users the original UI page, the detection UI page with the issue identification (the problematic component is circled), and the repaired UI page. Meanwhile, we also explained to the participants what is the color-related accessibility issues and the goal of Iris. In this part, we randomly look for participants to rate the repair effect of Iris and explain the shortcomings of the repair or their suggestions. The main evaluation criteria are: ① Whether the issues are successfully repaired. ② Whether the design style of the UI page after the repair is consistent with the original design style. ③ Whether the color matching of the whole UI page after the repair is coordinated. ④ Whether the color changes are minimal. ⑤ Score on the Likert scale ranging from 1 to 5.

At the app level, we evaluate the overall coordination of the repaired app through user research (*Task 3*). The goal of *Task 3* is to verify whether Iris performs well in maintaining overall color harmony within the app. We asked the participants to circle the UI pages and uncoordinated components in each app and counted the proportion of repaired components among the circled components. It is worth noting that we set the same resolution and the same page size for each UI screenshot, ensure that each participant can see the same screenshot, and eliminate the impact of other factors on user selection.

User study result. The results of the user study are shown in Figure 10 and Figure 11. Figure 10 shows the preference of participants with different vision levels for the UIs before and after repair, in which the blue bar chart represents the preference of participants with low vision and the orange bar chart represents the choice of participants with normal vision. On the whole, our repair is more attractive to people with low vision. Under the condition of low vision (without glasses), almost all participants chose the repaired page (94.80%). Meanwhile, under normal vision, only one-third of the results of page preferences are the original pages, and more than half (56.3%) of the results are the repaired pages. This also shows that the effect of our repair is also accepted and liked by most people, which can effectively solve the impact of low contrast on the viewing effect of people with weak vision. We also investigated the reasons why some participants chose the original page. For example, they said that they preferred the color matching of the original UI on the premise that they could see the text content. There is no doubt that this color preference has the participants' personal aesthetic standards.

Figure 11 shows the distribution of participants' score results on the repair effect of Iris under the condition of known issues (both participants' scores are greater than 3). More than one-third of the results (36.6%) are full scores (i.e., 5), and the results higher than 4 accounts for 81.3% of all scores. The final calculated average score is 4.218. Most of the participants said that the color contrast of the repaired UI was improved and easy to read, the replacement color selected during repair is also more coordinated with the style of the original UI, and the feeling of the page is better than the original UIs. In addition, some participants said that the repair of Iris is

Table 3: Results of *Task 3*.

App Package	Component			UI Page		
	C^{Re}_{All}	C^{Un}_{All}	C^{Un}_{Re}	P^{Re}_{All}	P^{Un}_{All}	P^{Un}_{Re}
jp.co.hateblo.bomberhead	11	3.88	1.12	4	0.69	0.13
com.guidoo.lulo.booxx	17	5.69	1.69	7	1.56	0.38
ch.rmy.android.http	10	2.50	0.88	8	1.81	0.37

Table 4: Feedback from app developers.

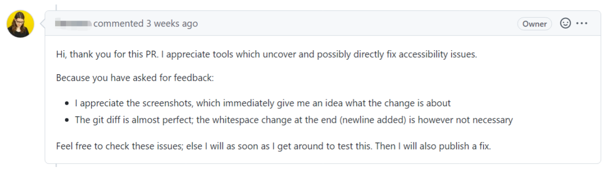
App	# Star	Version	IssueID	Issue State
OpenPods	613	v1.7 (18)	#142	**Merged**
motioneye-client	27	v1.0.0-alp9 (10000008)	#22	**Merged**
TapUnlock	28	v2.1.0 beta (13)	#5	**Merged**
ItsDicey	4	v1.0.1 (2)	#3	**Merged**
greentooth	20	v1.12 (5)	#4	**Merged**
knightsofalentejo	8	v5.0.0-RC-1 (5021)	#6	**Merged**
mundraub-android	32	v1.236 (237)	#326	**Merged**
DriSMo	23	v1.0.3 (17)	#4	**Merged**
Icicle for Freenet	10	v0.4 (4)	#5	**Merged**
Anki-Android	5.4k	v2.15.6 (21506300)	#10472	Positive Response
ActivityManager	69	v4.2.0 (415)	#6	Positive Response
Mensa	9	v1.7.0 (38)	#5	Positive Response
mundraub-android	32	v1.236 (237)	#326	Positive Response

more effective for people with low vision, and the color matching of the original UI also has practical advantages under the condition of normal vision.

Table 3 shows the feedback results of users on the overall coordination of the repaired apps at the app level. C_{All}^{Re} and P_{All}^{Re} respectively represent the number of components and UI pages repaired by Iris, while C_{All}^{Un} and P_{All}^{Un} respectively represent the average number of components and UI pages that are considered uncoordinated by the user. While C_{Re}^{Un} and P_{Re}^{Un} are the number of components repaired by Iris and contained in the components or UI pages circled by users (considered uncoordinated with the design style of the original app). It can be seen that the components and UI pages repaired by Iris are rarely considered to be uncoordinated with the original app, which also indicates that when Iris is applied to an app with color-related issues including multiple UI interfaces, the color selected by Iris is consistent with the original app design.

4.3.2 ***Feedback from app developers.*** **Setup.** To understand the views of app developers, we randomly selected 40 open-source apps containing color-related accessibility issues from F-Droid [26] and used Iris to repair them, aiming to obtain feedback from them about our issue repair results. Although there are many issues detected and repaired in each app, to investigate how different app developers of specific apps act towards accessibility issues and the repair result, we randomly selected one issue from each app and submit pull requests in the corresponding GitHub repositories, in which we introduced the focus of Iris, implementation functions, and the repair results to developers. We also asked for comments about Iris, the repair result, and suggestions for improvement.

Result. As shown in Table 4, till now, we received 9 merged pull requests and 4 positive comments. Meanwhile, we display the app name, app version, the number of stars, and the id of the pull requests in this table. During the traceability analysis, the developers claimed "*That looks interesting*" [9] and "*Good you made a PR and bring in your knowledge*" [10] in their feedback, indicating the usefulness and practicability of Iris. Figure 12 shows an example of feedback from an app developer. She pays attention to repairing such issues and hopes to have tools that uncover and directly fix accessibility issues. Although we find that some developers even do not know about these contrast issues [24], they have realized the importance of accessibility through Iris and want to use it to automatically repair their apps, which shows that Iris is meaningful. At the same time, other developers think that their limitation


[redacted] commented 3 weeks ago

Hi, thank you for this PR. I appreciate tools which uncover and possibly directly fix accessibility issues.

Because you have asked for feedback:

- I appreciate the screenshots, which immediately give me an idea what the change is about
- The git diff is almost perfect; the whitespace change at the end (newline added) is however not necessary

Feel free to check these issues; else I will as soon as I get around to test this. Then I will also publish a fix.

Figure 12: Feedback from the real app developers.

is the time to implement the fixes and worry about the difficulty of positioning. They also hope to have a tool that can automatically analyze relevant issues, and Iris just implements this function and realizes the positioning of related components and automated recommendation of replacement colors, which shows that Iris has practical significance. More importantly, Iris can clearly make more developers aware of these issues, so they can effectively avoid them during development.

5 DISCUSSION

5.1 Limitations

5.1.1 Scope-to-repair. (1) Iris repairs the apps based on the input test results, so we only repair the problematic components raised in the test report until now. If there are other components with color-related accessibility issues, but they are not detected by the detection tool, Iris will not repair them. In addition, most of the repair failures are due to the limitations of *bounds* shown in our experiments in § 4.1. Although using *bounds* can effectively solve the problem of text contrast, it cannot accurately locate the problem of image contrast. So the repair rate of image contrast is slightly lower. But if the information identifying the components in the detection report is more accurate, the less this restriction will be. In fact, although the repair of the apps highly depends on the detection report, if the other effective detection tools can be integrated in the future, Iris also works for them because it is scalable in practice.

5.1.2 Object-to-repair. (2) Iris adopts the method of static analysis in the localization and repair stage to analyze and extract attributes from the app UI layout file and its resource file. However, the properties of some components are set in the app source code. Currently, we can only handle the basic implementation by Java or Kotlin code such as using official API like *setTextColor*, as shown in Figure 2. We noticed that some third-party libraries provide new API interfaces to restructure the official APIs. Iris cannot resolve the restructured implementations, but they actually do not belong to the research scope of our work. Similarly, Jetpack uses a new view structure to implement GUIs, therefore, Iris has restrictions on apps using Jetpack. Based on our experiments and investigation, most apps use the traditional Android XML layouts (i.e., Android View) to design and implement their UI pages. Therefore, the current version of Iris is applicable to most recent apps.

5.2 Threats to validity

As we mentioned in § 4.2, the proportion of the first strategy directly depends on the size of the color reference DB we constructed in advance. If the size changes, the proportions of the first strategy will change accordingly. We are continuing to expand the reference

DB by running more apps. In fact, under the current situation, the proportion of the first strategy is already the largest, that is, it has the greatest impact on the color selection results.

6 RELATED WORK

6.1 Accessibility Issue Repair

6.1.1 Mobile platform. For mobile accessibility, automated repair has been a fresh research direction in recent years. Most of the existing studies focused on the issue categories with high proportions such as the issues of *item label and touch target.* As for the *item label* issues, the goal of repair is to augment or predict the missing content labels for UI components. For example, Zhang et al. [83] developed methods for robust annotation of app interface elements by leveraging social annotation techniques that have been used on the web. Chen et al. [15] trained a deep learning model named LabelDroid to predict the missing labels. COALA [56] was a similar work based on using deep learning algorithms. Moreover, crowd-sourcing techniques are also used to recommend the labels of UI components in [12]. In terms of *touch target* issues, named size-based accessibility issues, Alotaibi et al. [1] leveraged a genetic algorithm guided by a fitness function to automatically repair them. All above studies focus on addressing one specific type of issue due to the diverse characteristics of different issues.

Compared with these studies, we focus on the other issue categories including text contrast and image contrast, named color-related accessibility issues. However, this category is of great importance and critical not only for its high proportion but also for the impact of mobile accessibility.

6.1.2 Web platform. Substantial efforts are put into automatically fixing accessibility problems in web settings [50–52, 54, 60], but some of them focused more on Mobile Friendly Problems, which can inspire the repair of size-related issues, however, cannot benefit color-based accessibility issues. In the works related to color-related accessibility issues, the re-coloring tool [30] enhanced the accessibility of the entire page by changing the color matching of the entire web page. Tools [45, 62, 69] that modify the color matching of some web components with accessibility issues also tried to improve the contrast of components by changing the problematic text background color pairs, but they all lack consideration of the overall design style of the original web page. Last but not least, the implementation mechanisms are significantly different for web apps and Android apps, which directly distinguish repair solutions.

6.2 Accessibility Issue Detection and Analysis

Compared with mobile app testing including functional testing [28, 29, 72, 81] and security testing [19–21], mobile app accessibility testing is studied to a lesser extent. Silve et al. [70] surveyed the available approaches for detecting accessibility issues. The existing approaches can be categorized into static and dynamic methods. Android Lint [33] can report missing content labels, but it is ineffective for other issue categories. Accessibility Scanner [32] can detect issues with the help of manual exploration of the app but is limited to the low activity coverage. To mitigate such problems, Eler et al. [25] developed a model, named MATE, by generating testing cases specifically for accessibility testing. Similarly, Salehnamadi et al. proposed Latte [66] and Groundhog [67] by reusing tests written by developers or automatically generated to validate the accessibility of those use cases. Recently, Alshayban et al. [3] detected issues by deploying Monkey [36]. However, in a recent work, Chen et al. [16] identified such a solution is not effective and they further proposed Xbot to automatically detect accessibility issues by leveraging app instrumentation and data-flow analysis. Recently, several works [2, 55] have also focused on the interactive accessibility of apps when users with disabilities are using Assistive Technologies, such as TalkBack [78] and VoiceOver [6].

A large number of empirical studies focused on investigating the characteristics of mobile accessibility. Ross et al. [63] unveiled some common labeling issues. Their following work [64] studied the properties of each accessibility type. Yan et al. [80] investigated if the apps violate the accessibility guidelines and further introduced the degree of violation. Vendome et al. [75] proposed a taxonomy in terms of the aspects of accessibility issues by analyzing the developers' posts on Stack Overflow. Alshaybana et al. [3] proposed a metric named *inaccessibility issue rate* to measure the distribution of such a metric for each app, each issue type, and app categories. Moreover, they also observed this research field from app developers and users. Chen et al. [16] also conducted a large-scale empirical study from the perspective of the issues themselves and revealed many fine-grained findings.

7 CONCLUSION

In this paper, we propose Iris to automatically repair the color-related accessibility issues for Android apps. Our approach builds a large-scale reference DB to help design a context-aware color selection technique as well as well-defined criteria and an effective attribute-to-repair localization method. Based on these key phrases, Iris can identify the optimal color replacement for automated repair and further generate a new repackaged APK for app developers. Our comprehensive experiments including a user study demonstrate the effectiveness, efficiency, and usefulness of our approach from different aspects. We finally highlight that the feedback from several real app developers is quite positive and the merged pull requests on GitHub confirm the practicality of Iris.

8 DATA AVAILABILITY

We have released the code of Iris on GitHub [8], the constructed reference DB based on 9,978 apps and the 100 APK files used in our experiment on Google Drive [7]. To facilitate developers to understand the repair performance, we also have uploaded the data of the user study and some other repair cases to our website [11].

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No. 62102284, 62102197).

REFERENCES

[1] Ali S Alotaibi, Paul T Chiou, and William GJ Halfond. 2021. Automated Repair of Size-Based Inaccessibility Issues in Mobile Applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 730–742.

[2] Ali S Alotaibi, Paul T Chiou, and William GJ Halfond. 2022. Automated Detection of TalkBack Interactive Accessibility Failures in Android Applications. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 232–243.

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

[4] Abdulaziz Alshayban and Sam Malek. 2022. AccessiText: Automated Detection of Text Accessibility Issues in Android Apps. In *Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[5] Apple. 2022. *Accessibility - Apple*. https://www.apple.com/accessibility/

[6] Apple-VoiceOver. 2022. *VoiceOver on iPhone*. https://support.apple.com/en-sg/guide/iphone/iph3e2e415f/ios

[7] Anonymous Author. 2022. *The 100 apks used in our experiment*. https://drive.google.com/drive/folders/1MOEnN1j54HkRvTsigTodIpUo0IEWcOIJ?usp=sharing

[8] Anonymous Author. 2022. *Iris-mobile*. https://github.com/iris-mobile-accessibility-repair/iris-mobile.git

[9] Anonymous Author. 2022. *Seek advice from an app developer*. https://github.com/ankidroid/Anki-Android/issues/10472

[10] Anonymous Author. 2022. *Solve issue of low contrast*. https://github.com/niccokunzmann/mundraub-android/pull/326

[11] Anonymous Author. 2023. *Automated repair of color-related accessibility issues for Android apps*. https://sites.google.com/view/iris-mobile/home

[12] Erin Brady and Jeffrey P Bigham. 2015. Crowdsourcing accessibility: Human-powered access technologies. (2015).

[13] Posted by Amnet. 12 April, 2021. *Ensuring Mobile Accessibility: Color Contrast*. https://amnet-systems.com/ensuring-mobile-accessibility-color-contrast/

[14] Posted by Wiinnova. 2 June, 2020. *The Importance of Accessibility in Mobile App Development*. https://www.wiinnova.com/blog/the-importance-of-accessibility-in-mobile-app-development/

[15] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhut, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 322–334.

[16] Sen Chen, Chunyang Chen, Lingling Fan, Mingming Fan, Xian Zhan, and Yang Liu. 2021. Accessible or Not An Empirical Investigation of Android App Accessibility. *IEEE Transactions on Software Engineering* (2021).

[17] Sen Chen, Lingling Fan, Chunyang Chen, and Yang Liu. 2022. Automatically Distilling Storyboard with Rich Features for Android Apps. *IEEE Transactions on Software Engineering* (2022).

[18] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. Storydroid: Automated generation of storyboard for Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 596–607.

[19] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2020. An empirical assessment of security risks of global Android banking apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1310–1322.

[20] Sen Chen, Ting Su, Lingling Fan, Guozhu Meng, Minhui Xue, Yang Liu, and Lihua Xu. 2018. Are mobile banking apps secure? what can be improved?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 797–802.

[21] Sen Chen, Yuxin Zhang, Lingling Fan, Jiaming Li, and Yang Liu. 2022. Ausera: Automated security vulnerability detection for Android apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[22] Daniel Cohen-Or, Olga Sorkine, Ran Gal, Tommer Leyvand, and Ying-Qing Xu. 2006. Color harmonization. In *ACM SIGGRAPH 2006 Papers*. 624–630.

[23] Henrique Neves da Silva, Silvia Regina Vergilio, and André Takeshi Endo. 2022. Accessibility Mutation Testing of Android Applications. *Journal of Software Engineering Research and Development* 10 (2022), 8–1.

[24] Marianna Di Gregorio, Dario Di Nucci, Fabio Palomba, and Giuliana Vitiello. 2022. The making of accessible android applications: an empirical study on the state of the practice. *Empirical Software Engineering* 27, 6 (2022), 145.

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

[26] F-Droid. 2022. *F-Droid*. https://f-droid.org

[27] Facebook. 2022. *Facebook Accessibility - Home*. https://www.facebook.com/accessibility

[28] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 486–497.

[29] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 40th International Conference on Software Engineering*. 408–419.

[30] David R Flatla, Katharina Reinecke, Carl Gutwin, and Krzysztof Z Gajos. 2013. SPRWeb: Preserving subjective responses to website colour schemes through automatic recolouring. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 2069–2078.

[31] Google. [n. d.]. *Android Accessibility Help - Change text and display settings*. https://support.google.com/accessibility/android/answer/11183305

[32] Google. 2022. *Accessibility Scanner*. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=en_SG

[33] Google. 2022. *Android Lint*. https://developer.android.com/studio/write/lint.html

[34] Google. 2022. *Documentation | Android Developers*. https://developer.android.com/docs

[35] Google. 2022. *Google accessibility*. https://www.google.com/accessibility/

[36] Google. 2022. *Google Monkey*. https://developer.android.com/studio/test/monkey

[37] Google-Accessibility-Guideline. 2022. *Accessibility Guideline for Android apps*. https://support.google.com/accessibility/android/answer/6376559

[38] Google-Accessibility-Test-Framework. 2022. *Accessibility-Test-Framework-for-Android*. https://github.com/google/Accessibility-Test-Framework-for-Android

[39] Google-Espresso. 2022. *Espresso | Android Developers*. https://developer.android.com/training/testing/espresso

[40] Google-Monkey. 2019. *Google-Monkey*. https://developer.android.com/studio/test/monkey

[41] Google-Robolectric. 2022. *Robolectric*. http://robolectric.org/

[42] GSA. 2018. *European accessibility act - Employment, Social Affairs, Inclusion*. https://www.section508.gov/manage/laws-and-policies

[43] GSA. 2018. *IT Accessibility Laws and Policies*. https://www.section508.gov/manage/laws-and-policies

[44] Shing-Sheng Guan. 2002. A study of color harmony relating with area ratio. In *9th Congress of the International Colour Association*, Vol. 4421. SPIE, 199–202.

[45] Fredrik Hansen, Josef Jan Krivan, and Frode Eika Sandnes. 2019. Still not readable? An interactive tool for recommending color pairs with sufficient contrast based on existing visual designs. In *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*. 636–638.

[46] IBM. 2022. *Accessibility Research | IBM*. https://www.ibm.com/able/

[47] Noor A Ibraheem, Mokhtar M Hasan, Rafiqul Z Khan, and Pramod K Mishra. 2012. Understanding color models: a review. *ARPN Journal of science and technology* 2, 3 (2012), 265–275.

[48] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2018. Multi-objective optimization of energy consumption of guis in Android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 3 (2018), 1–47.

[49] Renan Lopes, Agebson Rocha Façanha, and Windson Viana. 2022. I can't pay! Accessibility analysis of mobile banking apps. In *Proceedings of the Brazilian Symposium on Multimedia and the Web*. 253–257.

[50] Sonai Mahajan, Negarsadat Abolhassani, Phil McMinn, and William GJ Halfond. 2018. Automated repair of mobile friendly problems in web pages. In *Proceedings of the 40th International Conference on Software Engineering*. 140–150.

[51] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. 2017. Automated repair of layout cross browser issues using search-based techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 249–260.

[52] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. 2017. Xfix: an automated tool for the repair of layout cross browser issues. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 368–371.

[53] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. 2018. Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 215–226.

[54] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. 2018. Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 215–226.

[55] Forough Mehralian, Navid Salehnamadi, Syed Fatiul Huq, and Sam Malek. 2022. Too Much Accessibility is Harmful! Automated Detection and Analysis of Overly Accessible Elements in Mobile Apps. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[56] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. 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*.

107–118.
[57] Microsoft. 2022. *Microsoft accessibility.* https://www.microsoft.com/en-us/accessibility
[58] Sergio Naranjo-Puentes, Camilo Escobar-Velásquez, Christopher Vendome, and Mario Linares-Vásquez. 2022. A Preliminary Study on Accessibility of Augmented Reality Features in Mobile Apps. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 454–458.
[59] Prane Mariel B Ong and Eric R Punzalan. 2014. Comparative analysis of RGB and HSV color models in extracting color features of green dye solutions. In *DLSU Research Congress*. 1500–20.
[60] Pavel Panchekha and Emina Torlak. 2016. Automated reasoning for web page layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 181–194.
[61] Python. 2022. *Python PIL | getcolors() Method.* https://www.geeksforgeeks.org/python-pil-getcolors-method/
[62] Rick T Richardson, Tara L Drexler, and Donna M Delparte. 2014. Color and contrast in E-Learning design: A review of the literature and recommendations for instructional designers and web developers. *MERLOT Journal of Online Learning and Teaching* 10, 4 (2014), 657–670.
[63] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O Wobbrock. 2018. 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, 119–130.
[64] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O Wobbrock. 2020. An Epidemiology-inspired Large-scale Analysis of Android App Accessibility. *ACM Transactions on Accessible Computing (TACCESS)* 13, 1 (2020), 1–36.
[65] Miho Saito. 1996. Comparative studies on color preference in Japan and other Asian regions, with special emphasis on the preference for white. *Color Research & Application* 21, 1 (1996), 35–49.
[66] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. 2021. Latte: Use-case and assistive-service driven automated accessibility testing framework for Android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–11.
[67] Navid Salehnamadi, Forough Mehralian, and Sam Malek. 2022. Groundhog: An Automated Accessibility Crawler for Mobile Apps. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[68] Amit Samsukha. 2021. *Why Mobile Application Development Is Important In Today's Scenario.* https://www.emizentech.com/blog/why-is-mobile-app-development-important.html
[69] Frode Eika Sandnes. 2021. Inverse Color Contrast Checker: Automatically Suggesting Color Adjustments that meet Contrast Requirements on the Web. In *The 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. 1–4.
[70] Camila Silva, Marcelo Medeiros Eler, and Gordon Fraser. 2018. A survey on the tool support for the automatic evaluation of mobile accessibility. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*. 286–293.
[71] Alvy Ray Smith. 1978. Color gamut transform pairs. *ACM Siggraph Computer Graphics* 12, 3 (1978), 12–19.
[72] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2020. Why my app crashes? understanding and benchmarking framework-specific exceptions of Android apps. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1115–1137.
[73] Sabine Süsstrunk, Robert Buckley, and Steve Swen. 1999. Standard RGB color spaces. In *Color and imaging conference*, Vol. 1999. Society for Imaging Science and Technology, 127–134.
[74] Philipp Urban, Tejas Madan Tanksale, Alan Brunton, Bui Minh Vu, and Shigeki Nakauchi. 2019. Redefining A in RGBA: Towards a standard for graphical 3D printing. *ACM Transactions on Graphics (TOG)* 38, 3 (2019), 1–14.
[75] Christopher Vendome, Diana Solano, Santiago Liñán, and Mario Linares-Vásquez. 2019. Can Everyone use my app? An Empirical Study on Accessibility in Android Apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 41–52.
[76] W3C. 2021. *Mobile Accessibility at W3C.* https://www.w3.org/WAI/standards-guidelines/mobile/
[77] W3C. 2022. *Text or Image Contrast.* https://www.w3.org/WAI/WCAG21/quickref/?versions=2.0#contrast-minimum
[78] Wiki-TalkBack. 2022. *Google TalkBack.* https://en.wikipedia.org/wiki/Google_TalkBack
[79] Moiz Yamani. 7 September, 2021. *Importance of Mobile Accessibility.* https://www.barrierbreak.com/importance-of-mobile-accessibility/
[80] Shunguo Yan and PG Ramachandran. 2019. The current status of accessibility in mobile apps. *ACM Transactions on Accessible Computing (TACCESS)* 12, 1 (2019), 3.
[81] Sen Yang, Sen Chen, Lingling Fan, Sihan Xu, Zhanwei Hui, and Song Huang. 2023. Compatibility Issue Detection for Android Apps Based on Path-Sensitive Semantic Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 257–269.
[82] Xiangyu Zhang, Lingling Fan, Sen Chen, Yucheng Su, and Boyuan Li. 2023. Scene-Driven Exploration and GUI Modeling for Android Apps. In *2023 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.
[83] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. 2018. Robust annotation of mobile application interfaces in methods for accessibility repair and enhancement. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 609–621.
[84] Yuxin Zhang, Sen Chen, and Lingling Fan. 2023. A Web-Based Tool for Using Storyboard of Android Apps. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 117–121.

Received 2023-02-02; accepted 2023-07-27