Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2021-IC-001

2021-TC-001

Guider: GUI Structure and Vision Co-Guided Test Script Repair for Android Apps

Tongtong Xu, Minxue Pan, Yu Pei, Guiyin Li, Xia Zeng, Yuetang Deng, Xuandong Li

Technical Report 2021

Guider: GUI Structure and Vision Co-Guided Test Script Repair for Android Apps

Tongtong Xu, Minxue Pan, Yu Pei, Guiyin Li, Xia Zeng, Yuetang Deng, Xuandong Li

ABSTRACT

GUI testing is an essential part of regression testing for Android apps. For regression GUI testing to remain effective, it is important that obsolete GUI test scripts get repaired after the app has evolved. In this paper, we propose a novel approach named Guider to automated repair of GUI test scripts for Android apps. The key novelty of the approach lies in the utilization of both structural and visual information of widgets on app GUIs to better understand what widgets of the base version app become in the updated version. A supporting tool has been implemented for the approach. Experiments conducted on the popular messaging and social media app Wechat show that Guider is both effective and efficient. Repairs produced by Guider enabled 88.8% and 54.9% more test actions to run correctly than those produced by existing approaches to GUI test repair that rely solely on visual or structural information of app GUIs.

1 INTRODUCTION

The importance of regression testing for ensuring that changes to an app do not break existing functionalities has been widely recognized and is greatly appreciated in mobile app development industry. Since most mobile apps interact with their users through rich graphical user interface (GUI), GUI testing has become an essential part of regression testing for these apps. In GUI testing, user inputs likes clicks and swipes on the screen are fed to the GUI of an app and the behaviors of the app are examined to determine whether they are correct or not [1, 2]. Most GUI tests are crafted or recorded as scripts to enable automated execution using test harnesses/tools like Appium [3] and Robotium [4]. In such scripts, GUI elements, or widgets, to be exercised are selected based on their positions and/or properties, making those test scripts highly sensitive to changes to the app GUI. While these test scripts should be repaired when they become obsolete, doing it manually can be highly tedious, time-consuming, and expensive. The fact that mobile developers tend to release new versions of their apps with new or improved features frequently to retain existing users and attract new users also renders manual repair of the obsolete test scripts undesirable, if not infeasible. On the one hand, new versions often involve changes to the app GUI to make the evolution of the app more visible to users, which implies that extra time is needed to repair the affected test scripts. On the other hand, frequent releases leave relatively shorter time for test script repair and regression testing.

Various approaches have been proposed to automatically repair the obsolete GUI test scripts for mobile apps. Model-based approaches like Atom [5] and Chatem [6] assume the availability of a precise behavioral model of the app under consideration and exploit the model to guide the construction of replacement test actions for the obsolete ones. Although such approaches can often produce high quality results when they have access to the required models, their applicability in practice is limited due to the challenges involved in constructing and maintaining the models for real-world apps. Recently, we proposed a computer-vision-based approach, named Meter [7], to GUI test script repair. Meter establishes the matching relation between elements on app GUIs based on their visual appearance, and it utilizes that relation to better locate the evolved GUI elements and validate the repaired test scripts. While Meter produced overall good results in repairing test scripts for open-source mobile apps across iOS and Android platforms, its effectiveness will become impaired when major changes happen to the appearances of app GUIs. In this paper, we argue that the static, structural information about app GUIs, which is easily accessible on Android, provides valuable guidance on understanding the evolution of the apps and should be combined with the visual information of elements on those GUIs to guide test repair. To the best of our knowledge, structural information has not been utilized in repairing GUI test scripts for Android apps, although similar information obtained from the DOMs of web pages has been successfully leveraged by approaches like Water [8] and Waterfall [9] to repair web application tests.

To obtain a better understanding of the limitations of existing approaches that are solely based on visual or structural information in repairing GUI tests for popular Android apps, we conducted an exploratory study. In the study, we applied Meter and an implementation of Water on Android, which we refer to as Wateroid, to repair GUI test scripts for top-ranked Android apps from the Google Play app store. The repair results show that each tool was actually successful in a significant percentage of cases where the other tool failed, which suggests the two approaches can be complementary in repairing GUI tests.

Based on the findings from the exploratory study, we propose in this paper a novel approach, named Guider (**G**UI str**u**cture and v**i**sion co-gui**d**ed t**e**st **r**epair), that combines the structural and visual information about app GUIs to guide effective and efficient test repair. An important task in GUI test script repair with Guider is to decide which widgets from the base version app are more likely to have changed and identify, for each of those widgets, which other widgets from the updated version are more likely to be the results of the changes. Guider classifies widgets of the base version app into three types by comparing their structural information in the two versions and applies different strategies in repairing test actions on different types of widgets. During the process, visual information of the widgets extracted using computer vision techniques complements the structural information and fine-tunes the priority of different widgets being used to construct repairs. Guider relies on the behaviors of the input test scripts on the base version app, or intentions [7], as the reference to decide the correctness of repairs.

We have implemented the Guider approach into a tool with the same name. To evaluate the effectiveness and efficiency of Guider, we applied the tool to repair GUI test scripts for Wechat. Wechat is a popular messaging and social media app with over 1.2 billion monthly active users as of the third quarter of 2020 [10] and the GUI test scripts used in the experiments were the ones crafted and maintained by the Wechat development team. Guider produced repairs to enable 62.7% and 58.9% more test actions to run successfully and correctly according to manual inspection, respectively. Compared with Meter and Wateroid, Guider enabled 88.8% and 54.9% more test actions to run correctly after repairing, respectively, taking a comparable amount of repairing time.

The contributions this paper makes are as the following:

- We conduct an exploratory study on 32 popular Android apps to understand the limitations of existing GUI test script repair tools that solely rely on structural or visual information about app GUIs;
- We propose a novel approach called Guider to automated GUI test script repair for Android apps; The approach combines structural and visual information of widgets on app GUIs to produce high-quality repairs.
- We implement a tool with the same name to support the easy application of the Guider approach;
- We empirically evaluate Guider's effectiveness and efficiency by applying it to repair GUI test scripts for Wechat. The evaluation results show that Guider is both effective and efficient in repairing obsolete GUI test actions.

The rest of this paper is organized as the following. Section 3 uses an example to demonstrate how Guider works from a user's perspective. Section 4 explains in detail how Guider relates widgets on app GUIs and construct repairs for obsolete test actions. Section 5 reports on the experiments we conducted to evaluate the supporting tool for Guider. Section 6 reviews research studies that are closely related to this work. Section 7 concludes the paper.

2 EXPLORATORY STUDY

To obtain first-hand knowledge about main reasons why existing approaches that are solely based on visual or structural information fail to produce correct repairs to Android GUI tests in practice, we conducted an exploratory study.

2.1 Subject GUI Test Repair Tools

We consider two subject GUI test script repair tools in this study, namely Meter [7] and Wateroid. Meter establishes the matching relation between elements on app GUIs based on their visual appearance, and utilizes that relation to better locate the evolved GUI elements and validate the repaired test scripts. Wateroid is our implementation of the Water technique [8] on Android. Water aims to repair GUI test scripts for web applications. It extracts structural information about GUI elements from the document object models (DOMs) of web pages, naively attempts all web elements that have the same value as the original element for at least one key property in constructing repairs, and accepts a repair as long as it can make the test execute further. Wateroid employs the UI Automator test automation framework [1] to retrieve the structural information about GUI elements of Android apps during run-time,

[1] https://developer.android.com/training/testing/ui-automator

Table 1: Subject apps used in the exploratory study.

APP	CATEGORY	VERSIONS		#ACTIONS
		BASE	UPDATED	
ABC Kids Tracing	Parenting	1.4.6	1.5.9	1
Dianping	Food & Drink	10.18.4	10.26.32	2
Duolingo	Education	4.89.6	4.91.2	1
ESPN	Sports	2.3.2	2.3.10	1
Firefox	Communication	68.10.1	84.1.1	5
Google Drive	Productivity	2.19	2.2	3
Google Earth	Travel & Local	9.3.19.8	9.121.0.5	1
Google Fit	Health & Fitness	2.45.13	2.46.22	4
Google Kids	Kids	5.34.3	5.36.5	1
Google Pay	Finance	2.117	2.12	2
GoogleTranslate	Tools	6.6.1	6.14	2
Greetings Island	Event	1.1.19	1.1.40	1
HD Camera	Photography	1.0.4	1.1.0	1
ibis Paint X	Art & Design	8.0.1	8.1.1	1
Lark Player	Music & Audio	4.12.7	4.13.3	7
maxim	Auto & Vehicles	3.3.2	3.12.1	9
Microsoft News	News & Magazines	20.105.01	20.275.01	2
Microsoft Team	Business	2020100901	2020121401	3
MyObservatory	Weather	4.17.6	4.17.12	1
Myztherapy	Medical	3.31.1	3.33	2
Pinterest	Lifestyle	8.40.0	8.45.0	2
ReadEra	Book & Reference	20.07.30	20.12.17	2
Reface	Entertainment	1.0.25.2	1.7.3.	1
TroveSkins	Beauty	7.4.4	7.5.0	1
Twitter	Social	8.25.1	8.73.0	5
Universal TV	House & Home	1.0.82	1.1.8	1
V380	Libraries & Demo	1.2.3	1.3.2	7
Wallli	Personalization	2.8.1	2.8.3	1
Waze	Maps & Navigation	4.52.3.4	4.69.0.3	4
Webtoon	Comics	2.5.8	2.6.1	2
Youtube	Video Players & Editors	15.43.32	15.50.35	1
yuu	Shopping	1.0.4	1.2.2	1
Overall	-	-	-	78

and it follows the same logic as that of Water in constructing and validating GUI test repairs.

2.2 Subject Apps and GUI Tests

For the subject apps used in the study to be representative of a wide range of Android apps, we collect one popular app from each category of apps in the Google Play app store (As of November 1, 2020), each app with two visually differentiable versions. All apps in Google Play are organized into 36 categories, including, e.g., `Business`, `Education`, and `Finance`. We exclude apps in categories `Game` and `Entertainment` from our study because the randomness and time-sensitiveness involved in their behaviors and the non-standard widgets they often use make them unsuitable to be tested using regular test scripts. We also exclude apps in categories `Google Cast` and `Wear OS by Google` since they can only be installed on specific devices. For each of the remaining 32 categories, we examine its apps in decreasing order of popularity until we find one app with two visually differentiable versions from the Apkpure website[2]. Apkpure is a third-party app market that provides download for not only the latest but also previous versions of a large amount of Android apps.

Particularly, given an app, we always consider its latest version as the updated version and look for a base version from the earlier versions on Apkpure. To that end, we first use the release notes of the app as a guidance to look for a most recent version of the app whose GUI is different from that of the updated version. If such a version is found, it is used as the base version. If no proper base

[2] https://apkpure.com/

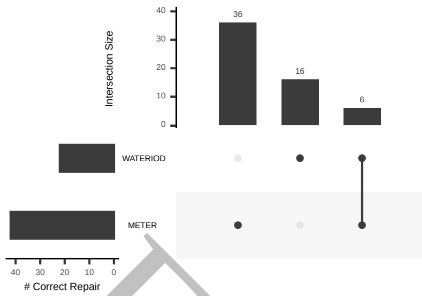


Figure 1: Partition of the obsolete test actions based on whether they can be correctly repaired by METER and WATEROID.

version can be identified based on the release notes, e.g., because the release notes do not provide sufficient information about the differences between versions, as is often the case with large apps like `Whatsapp` and `Facebook`, we manually examine the earlier versions of the app in reverse chronological order to spot a version with a different GUI. If such a version is found in no more than 30 minutes, it is used as the base version. Note that such process is feasible because the number of available versions for each app on Apkpure is typically small. If no desirable base version is found for an app at the end of this process, we move on and examine the next app in the current category. In this way, we gathered in total 32 popular apps, each with two versions that are visually different.

Next, for each subject app, we prepare one automated test script in Appium for its base version and make sure the changed GUI components are exercised at least once by the tests. 78 actions from those test scripts turned out to be obsolete when executed on the updated versions of the apps. In particular, 46 of those test actions caused crashes or became unexecutable, while the other 32 test actions, although still executable, exercised different functionalities than the intended ones.

Table 1 shows basic information about the subject apps used in this study and the tests we prepared for the apps. For each app (APP), the table lists its category (CATEGORY), the base (BASE) and updated (UPDATED) versions used in the study, as well as the numbers of test actions that become obsolete due to the GUI changes (#ACTIONS).

2.3 Study Results

We applied METER and WATEROID to repair the obsolete test actions in those scripts.

WATEROID considered 32 obsolete test actions that are still executable as successful and therefore not needing repairing. For the other 46 obsolete test actions, it correctly repaired 22 of them and failed to repair the remaining 24 test actions. Particularly, WATEROID was not able to find the correct, updated widgets based on their key properties in 17 of the failed cases, the structural information returned by UI Automator was incorrect in 4 of those cases (either because the information was not accessible to UI Automator for security reasons, e.g., on activities handling payments, or because the input focus of apps was not correctly configured, causing UI Automator to return the structural information about a background, rather than the foreground, activity), and the required repairs were too large for WATEROID to construct in the remaining 3 cases.

In comparison, METER attempted to repair all the 78 obsolete test actions, correctly repaired 42 of them, and failed to repair the remaining 36 test actions. Particularly, METER failed to repair 17 obsolete test actions because the GUI changes were too drastic for METER to find the correct, updated widgets based on screenshots, it failed to repair 16 obsolete test actions because the environment-specific contents displayed in the app GUIs prevented the activities from being matched, even when no changes were made to them across versions, and it also failed to produce the repairs for 3 test actions that were too large for METER to construct.

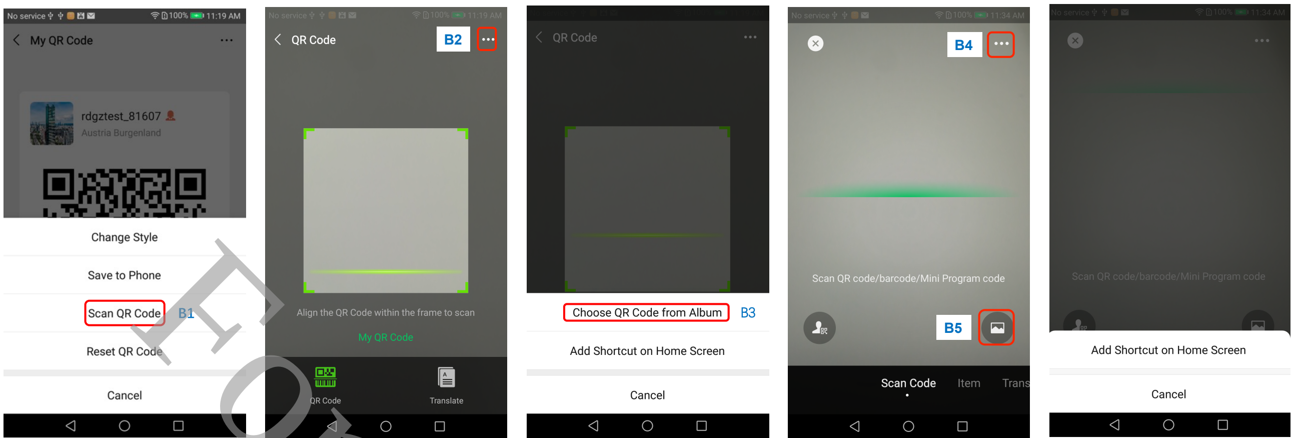
Here, we refer to all contents that are closely related to the testing environment as *environment-specific* contents. For example, messages received during testing and images stored on the testing device are two typical types of environment-specific contents. Unless we make sure tests are always executed in exactly the same environment, computer-vision-based GUI test approaches need to pay extra attention in handling environment-specific contents displayed on app GUIs to prevent such contents from misleading the repairing process. This requirement has been largely overlooked by METER and it was underrepresented in METER's evaluation because the testing environments used to run METER were carefully prepared to guarantee each test is always executed in the same environment. Such preparation, however, may be expensive, undesirable, or even impractical in practice: Always resetting the local testing environment before each test run can be highly expensive, always running a test in the same environment may greatly reduce the number of different behaviors the test may exercise, and controlling, e.g., whether or how many messages the server pushes to an app during test execution may not always be feasible.

More importantly, METER correctly repaired 36, or 64.3%, of the 56 obsolete test actions where WATEROID was ineffective, while WATEROID correctly repaired 16, or 44.4%, of the 36 obsolete test actions where METER was ineffective. Although this study is preliminary and its findings are far from being conclusive, such results provide clear evidence that visual and structural information about app GUIs should be combined to support more effective GUI test repair.

Fig 1 summarizes the repairing results produced with METER and WATEROID by partitioning the obsolete test actions based on whether they can be correctly repaired by each tool. Each vertical bar measures the number of obsolete test actions that a group of tools (indicated by connected dots in the lower part of the diagram) can correctly repair in common while no other tool can. For example, the leftmost column indicates that METER can correctly repair 36 obsolete test actions that WATEROID cannot, while the rightmost column indicates that METER and WATEROID can correctly repair 6 obsolete test actions in common. The horizontal bars on the left report how many obsolete test actions each tool can repair in total.

3 GUIDER IN ACTION

Based on the findings from the exploratory study, we propose a novel approach, named GUIDER, to effective GUI test repair for Android apps. In this section, we demonstrate from a user's perspective how GUIDER automatically repairs GUI test scripts for Android apps. Section 4 describes the approach in detail.



(a) Screen S1 (b) Screen S2 (c) Screen S3 (d) Screen S4 (e) Screen S5

Figure 2: Snapshots of WeChat screens in the base and updated versions.

```
# TS1
...
1 driver.find_element_by_description_content('Scan QR Code').click()
2 driver.find_elements_by_id('openIcon').click()
3 driver.find_elements_by_id('chooseFromAlbum').click()
...
```

Lst. 1: Test script for the base version.

```
# TS1'
...
4 driver.find_element_by_description_content('Scan QR Code').click()
5 driver.find_elements_by_id('moreMenu').click()
6 System.press_back()
7 driver.find_elements_by_id('chooseFromAlbum').click()
...
```

Lst. 2: Repaired test script for the updated version.

WeChat is a popular messaging and social media app with plenty of other functionalities. In particular, the app has a built-in QR code scanner that can be used to scan the QR code from an image stored on the device. Figure 2 shows the screen snapshots of the app in version 7.0.7 (the base version) and version 7.0.14 (the updated version) when invoking the functionality.

To scan the QR code from an image in the base version, a user may 1) tap the button with description content `Scan QR Code` (marked as B1) on screen S1, 2) tap the button with id `openIcon` (marked as B2) on screen S2, and 3) tap the button with id `chooseFromAlbum` (marked as B3) on screen S3. Afterwards, the app will list all the images from the album for the user to select from. Listing 1 shows the three test actions corresponding to these steps from a test script TS1 that exercises this functionality. The test script runs successfully on the base version of WeChat.

Screens S2 and S3, however, evolved into screens S4 and S5 in the updated version of the app. Particularly, the id of button B2 was changed to `moreMenu` (marked as B4), while the text button B3 was changed to an image button (marked as B5) and moved to screen S4, but with its id and functionality unchanged. The revision makes the test actions on lines 2 and 3 of test script TS1 obsolete, as none of the two actions can find any button with the desired ids on their corresponding screens.

Taking both versions of WeChat and the test script in Listing 1 as the input, Guider is able to automatically produce the repaired

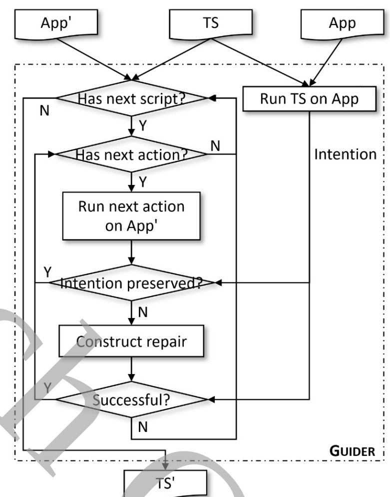


Figure 3: Overview of Guider.

test script TS1' as shown in Listing 2. While the id of button B2 was changed to `moreMenu`, its appearance remains the same as before. Guider is therefore able to identify that button B4 is the updated version of button B2 and revise the test action to tap the right button (Line 5 in Listing 2). Guider also discovers that there is no button with id `chooseFromAlbum` on screen S5, but a new button on screen S4 has the same id. Hence, the tool treats button B3 as being moved to screen S4 and becoming button B5, and produces a repair for the next test action so that first the app is navigated to screen S4 by pressing the `Back` button (Line 6 in Listing 2) and then button B5 is tapped using the right id (Line 7 in Listing 2) to list the images from the album. Note that it would be much less likely for test repair to produce such results if relying on just the structural information or only the visual information of the app GUIs.

4 THE GUIDER APPROACH

Figure 3 illustrates an overview of the Guider approach. Given a base version Android app (*App*), a group of test scripts for it

```
<hierarchy rotation="0">
   <node class="android.widget.FrameLayout" bounds="[0,0][1080,1812]" ...>
      ...
      <node text="Save to Phone" resource-id="com.tencent.mm:id/gam"
         class="android.widget.TextView" ... />
      ...
      <node text="Scan QR Code" resource-id="com.tencent.mm:id/gam"
         class="android.widget.TextView" .../>
      ...
   </node>
</hierarchy>
```

Lst. 3: Layout hierarchy extracted for screen S1 shown in Figure 1.

(TS), and an updated version of the same app (App'), Guider first records the intended behaviors of each input test script by running it on the base version app; Then, for each test action under repair Guider checks if the action preserves its intended behavior when executed on the updated version app. If yes, the test action does not need repairing; Otherwise, the test action is obsolete and Guider constructs a replacement for the next one or two test actions: The execution of the constructed replacement test action(s) on the updated version app should produce screen transitions that match with the ones triggered by the corresponding input test action(s) on the base version app. Without loss of generality, we assume all the input test scripts run successfully on the base version app.

Next, we first introduce the mechanism Guider uses to determine the matching relation between GUI elements and screens (Section 4.1), then explain how Guider repairs test scripts based on such matching relation (Section 4.2), and in the end describe the implementation details of a supporting tool for Guider (Section 4.3).

4.1 Widget and Screen Matching

Guider decides whether two test script executions conform to each other based on a matching relationship between the source and destination screens of their test actions—a screen of an app refers to the app's GUI that is visible to users at a particular point in time, and it determines the matching relationship between screens based on the matching relation between widgets on those screens. To strike a good balance between accuracy and efficiency in establishing the matching relations, Guider exploits both the structural and visual information of widgets.

Guider exploits the UI Automator framework to extract the structural and visual information of widgets and screens at runtime. UI Automator is a UI testing framework released as part of the Android SDK, and it features an API to retrieve not only the layout hierarchy that reflects the relations between widgets but also the properties of widgets on a screen. Widget properties that UI Automator can extract include, e.g., a descriptive `text`, a `bound` reflecting the position and size of a widget, a `content-desc` to help physically challenged users understand the purpose of a widget, and a `resource-id` indicating the resource from which a widget was instantiated. Listing 3 shows part of the layout hierarchy (in XML) that UI Automator extracted from screen S1 shown in Figure 2.

4.1.1 Identity Properties of Widgets. Three properties common to all widgets are especially important for deciding whether two widgets are matching in Guider, namely property `resource-id`, property `content-desc`, and property `text`, since the Android documentation recommends that different widgets should have distinct values for these properties[3] [4] [5]. We refer to these properties as *identity* properties.

Note that property `class` is not considered an identity property for two reasons. First, multiple valid values are often acceptable for the class property of a widget, making its distinguishing power limited. Second, the number of widgets with a particular class value can be large. For instance, there often exists dozens of widgets of class `ImageView` and/or `FrameLayout` on a screen.

4.1.2 Three Types of Widget Matches. Given a screen S_a of the base version app and a screen S'_a of the updated version app, Guider partitions the widgets on S_a into three types, namely α-typed, β-typed, and γ-typed, w.r.t. S'_a, based on how much confidence Guider has in finding the right matches for those widgets. Given a widget w on S_a, w is α-typed if and only if there exists a unique *sure* match for it on S'_a; w is β-typed if and only if it has no sure match, but a group of *close* matches, on S'_a; w is γ-typed if and only if it has only a group of *remote* matches, but no sure or close matches, on S'_a. Given S_a, a widget w on S_a, and S'_a, we use $\alpha(S_a, S'_a, w)$, $\beta(S_a, S'_a, w)$, and $\gamma(S_a, S'_a, w)$ to denote the sure match, the set of close matches, and the set of remote matches, when exists, for w on S'_a, respectively.

Sure Match. We identify sure matches for widgets on S_a in two steps. In the first step, we consider a widget w' on S'_a as a sure match for w if and only if the following two conditions are satisfied: 1) w and w' should have the same value for at least one identity property; 2) Compared with w', all the other widgets on S'_a have the same values as w for strictly fewer identity properties. In other words, w' is only considered a sure match for w if it has the same values for strictly the largest number of identity properties. Satisfying the two conditions also implies that there exists at most one sure match on S'_a for w.

In the second step, we build upon the identified sure match relation from the first step and exploit more structural information to extend the relation so that it also includes other pairs of widgets, using the following two policies. *Policy-A:* If 1) w is a component widget of a list item m on S_a, 2) w' is a component widget of a list item m' on S'_a, and 3) w' is the sure match of w, m' is the sure match for m. *Policy-B:* If 1) a list item m' on S'_a is the sure match for list item m on S_a and 2) a component widget w_1 of m and a component widget w'_1 of m' have the same value for at least one identity property, w'_1 is the sure match for w_1. Intuitively, Policy-A states that one list item should be considered the sure match for another list item if the two list items contain component widgets that surely match, while Policy-B states that, if two list items surely match, their component widgets with the same values for at least one identity property should surely match. The two policies enable us to reasonably extend the sure match relation to cover component widgets of list items that are closely related sure matches. According to experimental results reported in Section 5, the two policies work quite well on screens without nested lists or multiple lists of the same type. We leave the design of more sophisticated policies to identify sure matches for future work.

One widget having another widget as its sure match is a strong indication that the former has evolved to become the latter, and

[3] https://developer.android.com/guide/topics/resources/providing-resources
[4] https://developer.android.com/guide/topics/ui/accessibility/principles
[5] https://developer.android.com/reference/androidx/test/uiautomator/UiSelector

therefore both widgets will be excluded from being considered in potential matching relations with other widgets.

Close Matches. Compared with the sure match, a close match of a widget also has the same values for some identity properties, but it is not more likely to be the right match than the others. Specifically, we consider w as β-typed and regard a widget w' on S'_a as a close match for w when the following conditions are satisfied: 1) w is not α-typed, 2) w' is not the sure match of any widget on S_a; and 3) w and w' have the same value for at least one identity property.

Remote Matches. α-typed and β-typed widgets correspond to widgets on S_a that are not drastically changed, in the sense that at least one of their identity properties remains intact. It, however, may happen that the revision to a widget is so great that none of the widget's identity properties has its original value. Let r_1 be the set of widgets on S_a that has no sure or close match on S'_a, r_2 be the set of widgets on S'_a that share no values for identity properties with any widget on S_a. Each widget in r_1 is γ-typed and it has all widgets in r_2 as its remote matches.

Sorting Close and Remote Matches. There can be many close or remote matches for a widget, making it difficult to find the right match. Fortunately, widgets undertaking the same functionality in one app usually have similar appearance. To better distinguish the close and remote matches, Guider resorts to the visual information of w and w's potential matches. As explained at the beginning of Section 4.1, the layout hierarchy extracted by UI Automator from a screen contains a property named `bounds` for each widget on the screen that reflects the position and size of the widget. Using this information and the screenshot of the screen, the image of each widget on the screen can be easily obtained. Guider therefore retrieves the image of each widget on S_a and S'_a and applies the SIFT technique [11, 12] to extract feature descriptors from the image, as was done in [7]. The visual similarity between two widgets is then computed as the percentage of feature descriptors they have in common, and widgets in $\beta(S_a, w, S'_a)$ and $\gamma(S_a, w, S'_a)$ are sorted in decreasing order of their visual similarities to w.

4.1.3 Screen Matching.

Given screen S_a from the base version app and screen S'_a from the updated version app, Guider calculates the similarity between S_a and S'_a based on numbers of three types of widgets discovered on S_a w.r.t. S'_a. In particular, let c_s, c_c, and c_r be the numbers of α-typed, β-typed, and γ-typed widgets on S_a w.r.t. S'_a, respectively. The similarity between S_a and S'_a is then calculated as $sim(S_a, S'_a) = (c_s+c_c)/(c_s+c_c+c_r)$. Guider considers S_a and S'_a as matching, denoted as $S_a \sim S'_a$, if their similarity is greater than a threshold value θ_1. θ_1 is empirically set to 0.5 by default in Guider.

4.1.4 Falling back on Computer-Vision-Based Matching.

We noticed from the exploratory study that in two situations UI Automator may fail to retrieve the correct structural information about a screen. First, for security reasons, it may fail on activities that handle credential information. Second, it may return incorrect information if the input focus of an app is placed on a background, rather than the foreground, screen. Correspondingly, if Guider cannot retrieve any structural information about a screen or it detects mismatch between the retrieved structural information and the actual screen appearance, it will fall back on pure computer-vision-based widget and screen matching, as implemented in Meter [7].

4.2 Intention-Based Test Repair

In this work, we use a pair $\langle loc, evt\rangle$ to denote a test action a, where loc is an element locator to be used to pinpoint a particular GUI element on a given context screen, and evt is an event to be triggered on that element when a is executed. Following the practice in previous work [7, 13], we define a test script as a sequence $K = a_1, a_2, \ldots, a_n$, where each a_i $(1 \le i \le n)$ is a test action.

Test Action Intention. Successfully executing a test action $a = \langle loc, evt\rangle$ on a screen S involves first applying the locator loc to identify on S a target GUI element to interact with, then triggering the event evt on the element, and in the end transiting the app to a (possibly different) destination screen. We denote the screen transition caused by the successful execution of a as a pair $\langle src, dest\rangle$, where src and $dest$ are the source and destination screens of the transition, respectively. If the successfully terminated execution is also correct, or as expected, the transition characterizes the intended behavior of the test action, and we refer to the transition as the *intention* of the test action. A transition $\tau = \langle src_1, dest_1\rangle$ matches an intention $\iota = \langle src_2, dest_2\rangle$, denoted as $\tau \rightsquigarrow \iota$, if and only if $src_1 \sim src_2 \wedge dest_1 \sim dest_2$, i.e., their source screens and destination screens match respectively.

The Repair Algorithm. Algorithm 1 explains how Guider repairs a test script so that as many intentions of its test actions are preserved as possible. The algorithm takes the base version app $\mathcal{P}$, the updated version app $\mathcal{P}'$, and a test script K to repair as the input, and it produces a map that relates sequences of K's test actions to their repairs, with the intention of each test action sequence being preserved by the corresponding repair. The algorithm repairs the test actions from K in an iterative manner. Given the next test action $a1$ (Line 3) and the current screen $curS$ of $\mathcal{P}'$ (Line 4), Guider first retrieves the original intention $\iota 1$ of $a1$ (Line 5) and the original widget ε that $a1$ operates (Line 6) on $\mathcal{P}$, and then obtains the widget ε' that $a1$ will operate on $curS$ (Line 7) and the screen $curD$ that $a1$ will transit the app to (Line 8) on the updated version. All potential matches for ε on $curS$ are also stored into $matches$ (Line 9). Next, if ε' is the best match for ε and the screen it transits the app to match with $\iota 1.dest$, $a1$ can be retained as is without affecting its intention (Lines 10 through 12). Otherwise, Guider choose different strategy to repair $a1$ based on whether it is α-, β-, or γ-typed.

In case ε is α-typed (Line 13), Guider first checks whether $a1$'s sure match a' on $curS$ (Line 14) is a proper repair (Lines 15 and 16). If yes, the repair is registered at $\mathcal{M}$ and the process continues (Lines 17 and 18). Otherwise, if there exists another test action x that could be applied after a' to transit $\mathcal{P}'$ to a screen that matches with $\iota 1.dest$ (Line 19), Guider constructs a repair using a' and x for $a1$ (Lines 20 and 21). Or, if the transition achieved by a' preserves the overall intention of $a1$ and the test action $a2$ that follows it (Lines 24 and 25), Guider uses a' as the repair for test actions $a1$ and $a2$ (Line 26). If all these attempts fail, $a1$ cannot be successfully repaired and Guider proceeds to repair the next test script (Line 29). The rationale behind such design is: Since there is strong evidence that $a1$ has evolved into a', a' should always be part of the repair;

Algorithm 1: Intention-based test script repairing.

Input: Base version app $\mathcal{P}$; Updated version app $\mathcal{P}'$; A test script K to be repaired, with each test action associated with its intention on $\mathcal{P}$;
Output: Map $\mathcal{M}$ from sequences of test actions in K to triples of form $\langle \tau, src, dest \rangle$, where τ is the list of test actions derived from the sequence for $\mathcal{P}'$ and it transits $\mathcal{P}'$ from screen src to screen $dest$.

```
1  init(M) ;
2  for i = 0; i < K.length(); i++:
3      a1 ← K[i];
4      curS ← M(a1.pre).dest ;
5      ι1 ← a1.intention();
6      ε ← ELE(ι1.src, a1.loc);
7      ε′ ← ELE(curS, a1.loc) ;
8      curD ← DEST(curS, a1) ;
9      matches ← GETMATCHES(ι1.src, curS, ε) ;
10     if ε′ == matches.pop() ∧ ι1.dest ∼ curD:
11         M([a1]) ← ⟨[a1], curS, curD⟩ ;
12         continue;
13     if ISALPHATYPED(ι1.src, curS, ε):
14         a′ ← ACT(matches.pop());
15         curD′ ← DEST(curS, a′);
16         if ι1.dest ∼ curD′:
17             M([a1]) ← ⟨[a′], curS, curD′⟩ ;
18             continue;
19         if ∃x : DEST(curD′, x) ∼ ι1.dest:
20             newD ← DEST(curD′, x) ;
21             M([a1]) ← ⟨[a′, x], curS, newD⟩ ;
22             continue;
23         if i < K.length() − 1:
24             a2 ← K[i + 1] ; ι2 ← a2.intention();
25             if ι2.dest ∼ curD:
26                 M([a1, a2]) ← ⟨[a′], curS, curD′⟩ ;
27                 i ← i + 1;
28                 continue;
29             else: break;
30     isSuccessful ← false;
31     for j = 0; j < θ2 ∧ j < matches.length(); j++:
32         w ← matches[j] ;
33         a′ ← ACT(w);
34         curD ← DEST(curS, a′) ;
35         if ι1.dest ∼ curD:
36             M([a1]) ← ⟨[a′], curS, curD⟩ ;
37             isSuccessful ← true ;
38             break;
39     if isSuccessful: continue;
40     if ISGAMMATYPED(ι1.src, curS, ε):
41         foreach y ∈ actions(curS), z ∈ γ(DEST(curS, y)):
42             curD′ ← DEST(DEST(curS, y), ACT(z)) ;
43             if curD′ ∼ ι1.dest:
44                 M([a1]) ← ⟨[y, ACT(z)], curS, curD′⟩ ;
45                 isSuccessful ← true ;
46                 break;
47         if isSuccessful: continue;
48         else: break;
```

Guider therefore explores different possibilities regarding which and how other test actions are involved in the repair.

In case ε is not α-typed, Guider iterates through the first θ_2 widgets from ε's candidate matches on *curS* (Line 31). If there exists one match that can preserve $a1$'s intention (Line 35), the match is used to construct the repair for $a1$ (Line 36) and the repair of the current test script continues (Lines 37 through 39). Recall that all matches for β- and γ-typed widgets are sorted in decreasing order of their similarity to the original widget. Matching widgets with greater similarity values are therefore attempted by Guider earlier during repair. θ_2 is empirically set to 5 by default in Guider. We evaluate the impact of this choice on Guider's effectiveness in Section 5.

When no single action on a matching widget could preserve $a1$'s intention and $a1$ is γ-typed (Line 40), Guider also checks whether ε has evolved into a widget z on another screen that is reachable from *curS* in one action. If two test actions can be constructed to first navigate the app to where z is located and then transit to a screen that matches with $\iota1.dest$ (Lines 41 through 43), the two test actions are used as the repair for $a1$ (Line 44), and the repair of the current test script continues (Lines 45 through 47). Otherwise, Guider cannot repair $a1$ and it proceeds to repair the next test script (Line 48).

4.3 Implementation

We have implemented the approach described above into a tool, also named Guider, to automate the repair of GUI test scripts for Android apps. As explained in Section 4.1, Guider exploits the UI Automator framework to extract the structural and visual information of widgets and screens at runtime. For contour detection and optical character recognition (OCR) used in pure computer-vision-based matching, Guider uses the OpenCV library (Version 3.1) [14] and the Tencent OCR API, respectively. It, however, is worth noting that, Guider has been designed to support easy switch between libraries, and it should be easy for Guider to adopt future developments in computer vision and OCR techniques for better performance.

The tool has been integrated with the Appium testing framework and Tencent's dedicated testing infrastructure for Wechat (more about the infrastructure in Section 5.1), respectively, and the result of the former integration is available for download at

https://github.com/SEG-DENSE/Guider.

Since Guider is only loosely coupled with the underlying testing facilities, we can easily add support for other testing frameworks or infrastructures to the tool in the future.

5 EVALUATION

To evaluate, and put in perspective, the effectiveness and efficiency of Guider, we conducted experiments that apply Guider to repair GUI test scripts for Wechat. We address the following research questions based on the experimental results:

- **RQ1:** How effective and efficient is Guider in repairing GUI tests scripts?
- **RQ2:** How does Guider compare with existing test repair approaches like Meter and Wateroid that rely solely on visual or structural information of app GUIs?
- **RQ3:** How do values of parameters θ_1 and θ_2 affect Guider's effectiveness?

5.1 Experimental Subjects

To understand how Guider works on complex, commercial Android apps in practice, we use Wechat—a popular messaging and social media app—as our subject app. In particular, we choose Wechat 7.0.7, released about 12 months before this writing, as the base version, and versions 7.0.14, 7.0.15, 7.0.16, 7.0.17 and 7.0.18 as the updated versions. The reason for not using adjacent versions as the base and updated versions is that, the task of GUI test repair is likely more challenging in such settings since GUI differences between versions further away are likely greater, and we are interested to find out how GUIDER performs in tackling the challenging tasks.

In total, 277 GUI test scripts were crafted and maintained on the base version app by WECHAT developers, and all these test scripts can run automatically on Tencent's dedicated testing infrastructure for WECHAT[6]. There is one thing that is particularly interesting about these tests and their executions on the testing infrastructure: The execution of each WECHAT test script typically involves running multiple WECHAT instances in parallel and checking the interactions between those instances, and the testing infrastructure will launch those instances on mobile devices that are randomly selected from a pool when starting the test script. Although the randomness introduced by such design allows WECHAT to be exercised in more diverse ways during testing, different running mobile devices and/or leading interactions with other WECHAT instances may cause the internal states of the app in which a test action is triggered to vary across test executions, and such variations will add to the challenges GUIDER faces in repairing the test scripts. We include all these test scripts in our experiments.

Table 2 shows, for each pair of WECHAT versions, the base (V_b) and updated (V_u) version numbers, the number of test scripts *affected* when executed on the updated versions (#$\mathcal{K}$), and the number of test actions contained in those test scripts (#$\mathcal{A}$). In particular, a total number of 171 test scripts with 3322 test actions were affected on the updated versions of WECHAT. Note that the size of WECHAT is omitted from the table for confidentiality reasons.

5.2 Experimental Protocol

To answer RQ1, we apply GUIDER to repair GUI test scripts for all updated versions of WECHAT. Each experiment targets a particular pair of base and updated versions of WECHAT, and the inputs to GUIDER include the base and updated versions of WECHAT, denoted as $\mathcal{P}$ and $\mathcal{P}'$, respectively, and the set $\mathcal{K}$ of test scripts written for $\mathcal{P}$. Particularly, we first run $\mathcal{K}$ on $\mathcal{P}$ and record the structural and visual information about the screens before and after the execution of each test action from $\mathcal{K}$, then apply GUIDER to get the repaired set $\mathcal{K}'$ of test scripts as a derivation of $\mathcal{K}$, and finally ask five test engineers in Tencent to manually review and check the repair results to determine the numbers of transitions and test actions that fully preserve the test intentions. All the test engineers have more than five-year experience in mobile testing. A repair is considered to be intention-preserving only when all the five test engineers have a consensus on that.

To answer RQ2, on the one hand, we apply METER and WATEROID to repair the same test scripts for WECHAT, respectively, and compare their repairing results with that produced by GUIDER; On the other hand, we modified GUIDER to produce GUIDER-, which works the same as GUIDER except that it does not make any use of visual information about app GUIs. Recall that, GUIDER falls back on computer-vision-based widget and screen matching when structural information about app GUIs is inaccessible or incorrect. We repeat the same experiments using GUIDER- and compare the effectiveness of WATEROID and GUIDER-. We hope such comparison will help us understand better the differences between the two structural-information-based GUI test repair approaches as implemented in WATEROID and GUIDER.

In GUIDER's current implementation, two screens are considered matching if their similarity is greater than a threshold value $\theta_1 = 0.5$, and GUIDER at most examines the first $\theta_2 = 5$ elements from a widget's potential matches. To find out whether and how these parameters' values affect GUIDER's effectiveness and answer RQ3, we modify one parameter's value at a time, rerun the experiments on WECHAT, and study how changing each parameter influences the repairing results.

During each experiment, we record the following information:

- #$\mathcal{K}_s$: The number of test scripts that can execute *successfully*, i.e., without failures, to their completion after being repaired.
- #$\mathcal{A}_s$: The number of test actions that can execute *successfully* after repairing. This number includes test actions that are not affected by the changes and therefore need no repairing, test actions that are affected by the changes and successfully repaired, and test actions that can execute successfully after others being repaired.
- #$\mathcal{K}_c$: The number of test scripts that can execute *correctly* to their completions after being repaired, as manually confirmed by programmers.
- #$\mathcal{A}_c$: The number of test actions that execute *correctly* after being repaired, as manually confirmed by programmers.
- T: The overall wall-clock repairing time in minutes.

5.3 Experimental Results

This section reports on the results from experiments.

5.3.1 RQ1: Effectiveness and Efficiency. Table 2 reports, for each experiment conducted with GUIDER on a pair of WECHAT versions, the recorded measures.

To put the numbers in perspective, the table also lists, for each experiment, the same measures produced by a null test repair tool (NULL). A null test repair tool returns the same test action for each input test action. Therefore, the repairing results produced by a null test repair tool reflects how the test scripts execute on the updated version apps as they are. Measure T reported for the null test repair tool reflects the execution time of the test scripts.

Before being repaired, while 1790 of the 3322 test actions from 171 affected test scripts can still execute without causing any failures, only 1745 of them actually execute correctly. GUIDER was able to help make 101 test scripts and 2844 test actions execute successfully, and it made 100 test scripts and 2839 test actions execute correctly. In other words, GUIDER managed to increase the numbers of test actions that can execute successfully and correctly by 58.9% (=1054/1790) and 62.7% (=1094/1745). We attribute the high precision of GUIDER's repair results to both the adoption of intentions as the oracle for test action correctness and the combination of structural and visual information of widgets in repair construction and validation.

Five test actions were incorrectly repaired by GUIDER, all for the same reason. Specifically, the expected behaviors of those five test actions were to select specific elements from lists of environment-specific contents based on indexes. Since GUIDER always makes the selections based on the appearances of the list items, it tends to produce incorrect repairs in such cases, and because intention

[6] WeTest (https://wetest.qq.com).

Table 2: Experimental subjects and results.

WECHAT		$\#\mathcal{K}$	$\#\mathcal{A}$	NULL					WATEROID					METER					GUIDER-					GUIDER				
V_B	V_U			$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	T (m)	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	T (m)	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	T (m)	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	T (m)	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	T (m)
7.0.7	7.0.14	20	392	0	259	0	254	56.4	15	334	3	239	47.9	8	232	4	35	47.2	7	277	6	268	53.2	9	321	8	320	55.7
7.0.7	7.0.15	30	550	0	291	0	283	55.9	21	504	2	292	50.2	12	303	7	285	48.5	14	438	14	428	87.5	17	479	17	478	89
7.0.7	7.0.16	30	550	0	296	0	288	49	22	505	3	294	49.6	10	280	6	255	47.1	14	433	14	424	90.6	17	444	17	443	106.3
7.0.7	7.0.17	40	798	0	412	0	404	107.1	29	714	10	450	56.9	14	497	8	446	95.9	23	649	23	639	91.8	25	697	25	696	101.7
7.0.7	7.0.18	51	1032	0	532	0	516	69.1	37	888	8	558	63.1	13	536	7	483	58.4	29	837	29	823	120.8	33	903	33	902	135.1
Overall		171	3322	0	1790	0	1745	337.5	124	2945	26	1833	267.7	57	1848	32	1504	297.1	87	2634	86	2582	443.9	101	2844	100	2839	487.8
Average		34.2	664.4	0	358	0	349	67.5	24.8	589	5.2	366.6	53.54	11.4	369.6	6.4	300.8	59.42	17.4	526.8	17.2	516.4	88.78	20.2	568.8	20	567.8	97.56

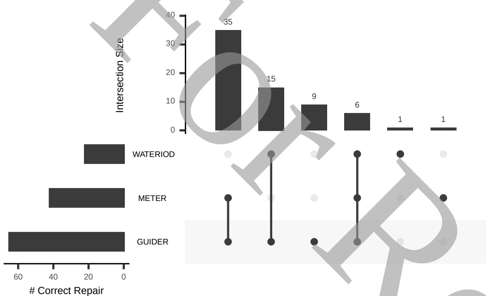


Figure 4: Partition of the obsolete test actions in the exploratory study based on whether they can be correctly repaired by METER, WATEROID, and GUIDER.

is just weak oracle for the correctness of test actions, GUIDER can seldom detect the problems with those repairs.

In total, GUIDER failed to produce any repair for 70 test actions, leaving 478 test actions from the input test scripts no longer executable. We discovered two reasons for the failures. First, GUIDER was not able to locate the widgets required by the correct repairs for 35 test actions. Particularly, the reason why GUIDER produced the 5 incorrect repairs also caused the tool to miss the right widgets in repairing 14 test actions, and GUIDER was unable to find the right widgets in repairing 21 test actions because the changes to the widgets, w.r.t. their property values and appearances, were too large. Second, although GUIDER managed to identify the right widgets in repairing the other 35 test actions, the constructed repairs all failed to satisfy GUIDER’s oracle for repair correctness because of the drastic changes occurred to the app GUIs. To overcome such limitations, we need mechanisms to enable us to communicate the actual intention of test actions to repair tools and novel techniques to help us better understand GUI changes and their impacts. We leave the design and implementation of such mechanisms and techniques for future work.

We have also applied GUIDER to the apps and GUI tests investigated in the exploratory study (Section 2). GUIDER correctly repaired 65, or 83.3%, of the 78 obsolete test actions, significantly outperforming METER and WATEROID, which correctly repaired 42 and 22 test actions, respectively. Such results provide initial evidence that GUIDER is also effective in repairing GUI tests for other Android apps. Fig 4 shows the updated partition of the obsolete test actions investigated in the exploratory study after incorporating GUIDER’s repairing results into Fig 1.

The overall repairing time with GUIDER is less than twice of the execution time of those test scripts. We therefore consider that GUIDER is efficient in producing the repairs. While computer vision techniques are often considered expensive to apply, GUIDER does not suffer from a prolonged repairing process mainly because the Tencent OCR API can often return results instantly.

GUIDER produced repairs to help 58.9% and 62.7% more test actions execute successfully and correctly, respectively. Test repairing time with GUIDER and the test execution time have the same order of magnitude.

5.3.2 *RQ2: Comparison.* Table 2 also lists the same measures achieved by METER, WATEROID, and GUIDER- in the experiments.

METER was able to help make 57 test scripts and 1848 test actions execute successfully, and it made 32 test scripts and 1504 test actions execute correctly. In comparison, GUIDER was able to make 44, or 77.2%, more test scripts and 996, or 53.9%, more test actions run successfully, and it made 68, or 213%, more test scripts and 1335, or 88.8%, more test actions run correctly. We manually examined the repairing results produced by METER and discovered that METER’s limited capability to handle environment-specific contents is the primary reason for its ineffectiveness in repairing tests for WECHAT. As explained in Section 2, both the running mobile device and the leading interactions with other WECHAT instances may vary for a test action across executions in our experiments, but METER was unprepared for handling the discrepancies in GUIs caused by environmental factors at all and was therefore often ineffective. Given the challenges involved in always preparing the identical testing environment in practice, the comparison result between GUIDER and METER highlights that structural information about app GUIs is an essential supplement to visual information in achieving practical, effective GUI test repair.

WATEROID enabled 124 test scripts and 2945 test actions to run successfully, and it enabled 26 test scripts and 1833 test actions to run correctly. In comparison, GUIDER was able to make 23, or 22.8%, fewer test scripts and 101, or 3.6%, fewer test actions run successfully. Meanwhile, GUIDER made 74, or 285%, more test scripts and 1006, or 54.9%, more test actions run correctly than WATEROID. WATEROID produced a large number of incorrect repairs because the oracle it adopts for repair correctness is much weaker than GUIDER’s intention-based oracle: WATEROID considers a repair correct if it does not trigger any error at run-time. WATEROID’s primitive way of finding widget matches by simply comparing their property values also contributed in part to the high number of incorrect repairs.

Compared with WATEROID, GUIDER- was able to make 37, or 42.5%, fewer test scripts and 311, or 11.8%, fewer test actions run successfully, but it made 60, or 231%, more test scripts and 749, or 40.9%, more test actions run correctly. Such results suggest the utilization of structural GUI information in GUIDER is more effective

Table 3: GUIDER effectiveness with different values for θ_1.

VER$_{UPD}$	$\theta_1 = 0.3$				$\theta_1 = 0.4$				$\theta_1 = 0.5^*$				$\theta_1 = 0.6$			
	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$
7.0.14	11	309	9	304	10	314	8	309	9	321	8	320	10	296	8	291
7.0.15	17	463	17	459	18	459	18	458	17	479	17	478	15	462	15	461
7.0.16	16	476	16	475	17	468	17	467	17	444	17	443	12	396	12	395
7.0.17	23	691	23	690	25	715	25	714	25	697	25	696	21	681	21	680
7.0.18	31	899	31	898	35	926	35	925	33	903	33	902	22	736	22	735
Overall	98	2838	96	2826	105	2882	103	2873	101	2844	100	2839	80	2571	78	2562

Table 4: GUIDER effectiveness with different values for θ_2.

VERSION	$\theta_2 = 1$				$\theta_2 = 3$				$\theta_2 = 5^*$				$\theta_2 = 7$			
	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$	$\#\mathcal{K}_S$	$\#\mathcal{A}_S$	$\#\mathcal{K}_C$	$\#\mathcal{A}_C$
7.0.14	7	311	6	310	9	321	8	320	9	321	8	320	9	321	8	320
7.0.15	15	469	14	468	17	479	16	478	17	479	16	478	17	479	16	478
7.0.16	15	434	14	433	17	444	16	443	17	444	16	443	17	444	16	443
7.0.17	25	696	24	695	25	697	24	696	25	697	24	696	25	697	24	696
7.0.18	30	886	29	885	33	903	32	902	33	903	32	902	33	903	32	902
Overall	92	2796	87	2791	101	2844	96	2839	101	2844	96	2839	101	2844	96	2839

and reliable than that in WATEROID. Although GUIDER- outperformed both WATEROID and METER in our experiments, it failed on a significant number of test actions that GUIDER correctly repaired: GUIDER was able to make 14, or 16.3%, more test scripts and 257, or 9.9%, more test actions run correctly. In particular, no structural information was available when repairing 17 test actions, so GUIDER fell back on computer-vision-based widget and screen matching, and visual information was needed to better rank the candidate matching widgets in repairing 26 test actions. These cases clearly demonstrate that visual information is a necessary complement to structural information in effective GUI test repair.

The overall repairing times with all the tools had the same order of magnitude and were less than twice the test execution time. Therefore, we consider all these tools are comparable in efficiency.

Compared with METER and WATEROID, GUIDER enabled 88.8% and 54.9% more test actions to run correctly, respectively. All the studied test repair tools were comparable in efficiency.

5.3.3 RQ3: Parameters. Table 3 lists the measures achieved by GUIDER on each updated version of WECHAT using various values for θ_1. The default value for θ_1 is marked with an asterisk (*). It is interesting to note from the table that, extreme θ_1 values tend to produce worse repair results, in terms of numbers of test scripts and test actions repaired (correctly or not). This is understandable, since both too large and too small θ_1 values increase the chance for GUIDER to miss a correct repair for an affected test action, which also makes it more likely for GUIDER to fail repairing the whole test script. Such results suggest that 0.5 is an appropriate default value for the parameter. Depending on whether the differences between the two versions of the Android app under repairing is large or not, a larger or smaller value may be adopted to suit the repairing task.

Table 4 lists the measures achieved by GUIDER on each updated version of WECHAT using different values for θ_2. The default value for θ_2 is also marked with an asterisk (*). We can observe from the table that, even with a very small value for θ_2, GUIDER is able to produce over 90% of the repairs that it can produce with larger θ_2 values. This suggests GUIDER is in general highly effective in identifying the right matches for widgets. We can also observe that, repair results reach a plateau quickly with larger θ_2 values. On the one hand, this suggests the effectiveness of GUIDER is largely insensitive to θ_2; On the other hand, it also means certain repairs cannot be produced by GUIDER even with larger θ_2 values. In the future, we will investigate further why this is the case and how to overcome this limitation.

A moderate θ_1 value produced the best repair results; A small θ_2 value was enough to produce good repair results already. The effectiveness of GUIDER was insensitive to increase in θ_2 values.

5.4 Threats to Validity

In this section, we discuss possible threats to the validity of our study and show how we mitigate them.

Construct validity. In this work, we asked programmers to manually inspect the repair results and label the correct repairs. Programmers, however, may have different opinions regarding the correctness of repairs. To mitigate this risk, we conservatively mark a repair as correct only when all the five programmers reach a consensus on that.

Internal validity. In our experiments, a major threat to internal validity is the possible faults in the implementation of our approach and the integration of external libraries. To address the threat, we review our code and experimental scripts to ensure their correctness before conducting the experiments.

External validity. A major threat to external validity is that, the apps and test scripts used in our experiments may not be good representatives of Android apps and test scripts people write in industry. To mitigate this threat, we used WECHAT, a popular app with a huge number of monthly active users, and its tests as subjects in our experiments. In the future, we plan to conduct larger scale experiments to evaluate GUIDER more thoroughly.

6 RELATED WORK

In this section, we review works closely related to GUIDER in general purpose test repair and GUI test repair.

6.1 General Purpose Test Repair

Changes made to a software system during its evolution may render some existing tests for the system obsolete. That is, those tests will fail on the evolved system not because the system is buggy, but because the tests do not embody the changes. To reduce the burden of updating those obsolete tests for programmers, various techniques have been developed in the past years. Daniel et al. [15] propose the REASSERT technique to repair obsolete unit tests automatically. REASSERT monitors the execution of a unit test on a presumably correct program and uses the information gathered during the execution to update the assertion methods, assertions or literal values in the test. To overcome some limitations in REASSERT, Daniel et al. [16] propose symbolic test repair. Symbolic test repair creates symbolic values for literals used in the tests and executes the tests in a symbolic way. The assertions and path conditions gathered during the execution are then solved by the Z3 constraint solver [17] and the solutions are used to replace the literals. Deursen et al. [18] propose techniques to fix compilation errors in tests caused by refactorings to the program code. Yang et al. [19] propose the SPECTR technique that repairs tests based on changes to program specifications rather than implementations.

6.2 GUI Test Repair

Compared with general purpose test repair, the problem of GUI test repair has attracted more attention from researchers. On the one hand, most software programs interact with their users via GUI for better user experience, and GUI testing is a popular way to detect faults in these programs at the system level. On the other hand, it is common for developers to create GUI test scripts using record-and-replay testing tools. GUI test scripts, however, are often more fragile, e.g., than unit tests.

Targeting traditional desktop applications, Memon and Soffa [20] first propose the idea of GUI test script repair and develop a model-based approach called GUI Ripper. GUI Ripper assumes that the application model and user modifications are completely known, and repairs scripts base on four user-defined transformations. A few years later, through reverse engineering, Memon [21] extends GUI Ripper by adding a mechanism to obtain the application model. Considering that the model built by GUI Ripper is just an approximation of the actual application and may cause incorrect repairs, Huang et al. [22] propose to use a genetic algorithm to generate new, feasible test cases as repairs to GUI test suites. Besides model-based approaches, several white box approaches have also been studied for GUI test script repair. Daniel et al. [23] propose to record GUI code refactorings as they are conducted in an IDE and leverage them to repair test scripts. Grechanik et al. [24] propose a tool to extract information about GUI changes by analyzing the source code and test scripts, and generate repair candidates for GUI test scripts to be selected by testers. Based on static analysis, Fu et al. [25] develop a type-inference technique for GUI test scripts, which can assist testers to locate type errors in GUI test scripts. Dynamic and static analyses have also been combined in GUI test script repair for desktop applications. To repair changed GUI workflows, Zhang et al. [26] combine the information extracted from dynamic execution of the applications and static analysis of matching methods to generate recommendations for replacement actions. Gao et al. [27] study the limitations of existing approaches and the importance of human knowledge, and propose a semi-automated approach called SITAR that takes human input to improve the completeness of extracted models and further repairs test scripts for desktop applications.

Compared with desktop applications, research on GUI testing for web and mobile applications has gained better results. On the one hand, web or mobile applications tend to have less complex GUIs than desktop applications. On the other hand, the DOM tree of a web application's web page and the layout hierarchy of a mobile application record detailed information of the widgets on the GUIs, which, when available, provides extra guidance on how the tests should be repaired. Raina and Agarwal [21] propose to reduce the cost of regression testing for web applications by executing only the tests that cover the modified parts of the applications, thus, developers are required to maintain only a subset of all test scripts. In their approach, the modified part of an application are automatically identified by comparing the DOM trees generated for the corresponding web pages. Choudhary et al. [8] propose the Water technique to repair GUI test scripts for a web application so that the scripts can run successfully on the updated version of the same application. Water only repairs a test action after a failure, naively attempts all web elements that share at least one key property with the original element, and accepts an element as the repair as long as it can make the test execute further. Therefore, Water tends to produce a large amount of overfitting repairs. Stocco et al. [28] propose the Vista technique to repair locator-related test breakages for web applications. Vista relies on visual information to decide the correctness of web element locators utilized in tests and, when a locator is incorrect, to select the right web element to access. The XPath information of the selected element is then extracted from the application DOM to construct the repair locator. Meter [7] leverages computer vision techniques to capture the intended behaviors test scripts, to detect deviations from those intentions, and to construct repairs to reduce the deviations as much as possible. While not requiring any structural information about the apps under consideration makes Meter widely applicable, failing to make good use of the more precise information about the apps even when it is available adversely impacts the precision of the repairing results Meter is able to produce. Compared with these techniques, Guider combines structural and visual information of Android apps to deliver more precise repairs to GUI tests in a more efficient way.

7 CONCLUSION

In this paper, we propose Guider—a novel approach that combines structural and visual GUI information to automatically repairing GUI test scripts for Android apps. Experimental evaluation of Guider on Wechat shows that Guider is both effective and efficient.

REFERENCES

[1] G. Bae, G. Rothermel, and D.-H. Bae, "Comparing model-based and dynamic event-extraction based gui testing techniques: An empirical study," *Journal of Systems and Software*, vol. 97, pp. 15 – 46, 2014.

[2] A. M. Memon, "An event-flow model of gui-based applications for testing: Research articles," *Softw. Test. Verif. Reliab.*, vol. 17, no. 3, pp. 137–157, Sep. 2007.

[3] "Appium: Mobile App Automation Made Awesome," http://appium.io/, 2018, [Online; accessed 20-March-2018].

[4] "Android UI Testing," http://www.robotium.org, 2018, [Online; accessed 20-March-2018].

[5] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "ATOM: automatic maintenance of GUI test scripts for evolving mobile applications," in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 2017, pp. 161–171.

[6] N. Chang, L. Wang, Y. Pei, S. K. Mondal, and X. Li, "Change-based test script maintenance for android apps," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2018, pp. 215–225.

[7] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li, "GUI-guided test script repair for mobile apps," *IEEE Transactions on Software Engineering*, 2020.

[8] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "Water:web application test repair," in *International Workshop on End-To-End Test Script Engineering*, 2011, pp. 24–29.

[9] M. Hammoudi, G. Rothermel, and A. Stocco, "WATERFALL: An incremental approach for repairing record-replay tests of web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 751–762.

[10] "Number of monthly active WeChat users from 2nd quarter 2011 to 3rd quarter 2020," https://www.statista.com/statistics/255778/number-of-active-wechat-messenger-accounts/, 2020.

[11] D. G. Lowe, "Object recognition from local scale-invariant features," in *ICCV*, 1999, pp. 1150–1157.

[12] ——, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[13] M. Leotta, D. Clerissi, C. Spadaro, and C. Spadaro, "Comparing the maintainability of selenium webdriver test suites employing different locators: a case study," in *International Workshop on Joining Academia and Industry Contributions To Testing Automation*, 2013, pp. 53–58.

[14] "OpenCV library," https://opencv.org/, 2018, [Online; accessed 20-March-2018].

[15] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "Reassert: Suggesting repairs for broken unit tests," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, Nov 2009, pp. 433–444.

[16] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, 2010, pp. 207–218.

[17] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 2008, pp. 337–340.

[18] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok, "Refactoring test code," Amsterdam, Netherlands, Tech. Rep., 2001.

[19] G. Yang, S. Khurshid, and M. Kim, "Specification-based test repair using a lightweight formal method," in *FM 2012: Formal Methods*, D. Giannakopoulou and D. Méry, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 455–470.

[20] A. M. Memon and M. L. Soffa, "Regression testing of guis," in *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*, J. Paakki and P. Inverardi, Eds. ACM, 2003, pp. 118–127.

[21] S. Raina and A. P. Agarwal, "An automated tool for regression testing in web applications," *SIGSOFT Softw. Eng. Notes*, vol. 38, no. 4, pp. 1–4, Jul. 2013.

[22] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing GUI test suites using a genetic algorithm," in *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. IEEE Computer Society, 2010, pp. 245–254.

[23] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, and D. Marinov, "Automated gui refactoring and test script repair," in *International Workshop on End-To-End Test Script Engineering*, 2011, pp. 38–41.

[24] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving gui-directed test scripts," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, pp. 408–418.

[25] C. Fu, M. Grechanik, and Q. Xie, "Inferring types of references to GUI objects in test scripts," in *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*. IEEE Computer Society, 2009, pp. 1–10.

[26] S. Zhang, H. Lü, and M. D. Ernst, "Automatically repairing broken workflows for evolving GUI applications," in *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, M. Pezzè and M. Harman, Eds. ACM, 2013, pp. 45–55.

[27] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, "SITAR: GUI test script repair," *IEEE Trans. Software Eng.*, vol. 42, no. 2, pp. 170–186, 2016.

[28] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 503–514.