GUI-Guided Test Script Repair for Mobile Apps

Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, Xuandong Li

Abstract—Graphical User Interface (GUI) testing is widely used to test mobile apps. As mobile apps are frequently updated and need repeated testing, to reduce the test cost, their test cases are often coded as scripts to enable automated execution using test harnesses/tools. When those mobile apps evolve, many of the test scripts, however, may become broken due to changes made to the app GUIs. While it is desirable that the broken scripts get repaired, doing it manually can be preventively expensive if the number of tests need repairing is large.

We propose in this paper a novel approach named METER to repairing broken GUI test scripts automatically when mobile apps evolve. METER leverages computer vision techniques to infer GUI changes between two versions of a mobile app and uses the inferred changes to guide the repair of GUI test scripts. Since METER only relies on screenshots to repair GUI tests, it is applicable to apps targeting open or closed source mobile platforms. In experiments conducted on 22 Android apps and 6 iOS apps, repairs produced by METER helped preserve 63.7% and 38.8% of all the test actions broken by the GUI changes, respectively.

1 Introduction

Mobile apps—programs that run on mobile devices—are becoming increasingly prevalent and transforming the world [1], and the competition in mobile industry is also getting continuously more fierce than before. Since most users prefer, if everything else being similar, apps with more frequent updates [2], mobile developers tend to release updates more frequently in order to keep existing users and attract new users. For example, major companies like Facebook and Netflix release their mobile apps once every two weeks. Unfortunately, more frequent updates and shorter developing time for each update make it harder to guarantee the quality of apps. In fact, many users encountered problems after updating apps [2]. Quality control has become a pressing issue for mobile app development.

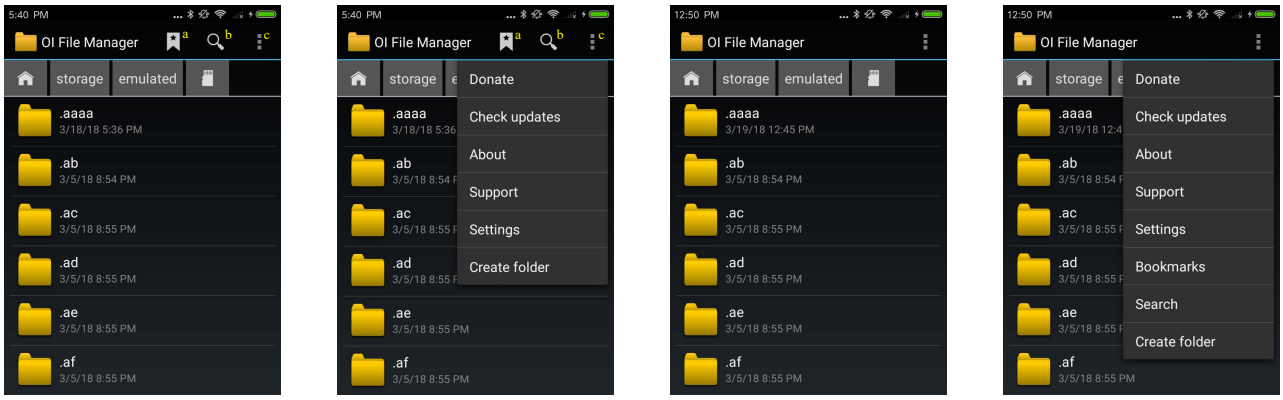
Meanwhile, the event-driven nature and gesture-based interactions of mobile apps [3] make the testing of such apps highly dependent on their Graphical User Interfaces (GUIs), and GUI testing has become one of the most widely used methodologies for testing mobile apps [4]. By feeding test inputs (e.g., touching a button) to the GUI of an app, GUI testing examines the behaviors of the app and checks whether they are correct [5], [6]. Since pure manual GUI testing is costly and time-consuming, most GUI tests used in industry are programmed or recorded as scripts to enable automated execution by test harnesses/tools [7] such as Appium [8] and Robotium [9]. For such scripts to comprehensively cover the business logic of apps, human testers have to invest valuable time to transcribe their domain knowledge, which makes the scripts valuable artifacts of mobile app development [7]. However, test scripts prepared for an app may become broken after the app is updated and its GUI changed. It is desirable that these broken test scripts get fixed and the testers' knowledge gets preserved, but the benefits of fixing the test scripts can be quickly dwarfed by the entailed high costs if the fixing is to be done manually and the number of test scripts that need fixing is large [10], [11]. Considering that a typical mobile app company often consists of just a small number of developers [12], the demand for automated test script repair for mobile apps is enormous.

Although research on test script repair for desktop and web applications has attracted growing interest in the past few years and produced promising results [7], [10], [13]–[18], characteristics of mobile app testing present new challenges. Particularly, most existing test script repair techniques that do not require human assistance need extensive static information from application source code or structure to effectively repair test scripts. For example, given a web application, detailed information about the composition of its GUI can be reliably obtained by analyzing the DOM of the web page, and such information can be used to facilitate both the extraction of GUI changes between different versions of the application and the maintenance of broken tests due to those changes [16]–[18]. To statically acquire similar information about the GUI of a mobile app, however, is not always feasible. On the one hand, the source code of mobile apps is often not accessible to testers, as testing of mobile apps is increasingly more often outsourced or offered as cloud-based services [19], rather than conducted by testers from the apps' development teams. On the other hand, information directly extracted from an app's package is seldom sufficient for comprehending the composition of the GUI: Although it is straightforward to obtain the IDs and string literals of an app's GUI elements, e.g., by parsing the XML-based files on the Android platform or the iOS platform, many GUI elements do not have IDs; Besides, mobile apps often have more GUI elements in image than

- *Minxue Pan is with the State Key Laboratory for Novel Software Technology and the Software Institute of Nanjing University, China. E-mail: mxp@nju.edu.cn.*
- *Tongtong Xu, Zhong Li, Tian Zhang and Xuandong Li are with the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology of Nanjing University, China. E-mail: {dz1633014,mg1733033}@smail.nju.edu.cn, {ztluck,lxd}@nju.edu.cn.*
- *Yu Pei is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. E-mail: csypei@comp.polyu.edu.hk.*

Received: revised:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007664, IEEE Transactions on Software Engineering



(a) Initial screen of the base version.

(b) Menu "More options" in the base version.

(c) Initial screen of the updated version.

(d) Menu "More options" in the updated version.

Figure 1: GUIs of OI File Manager in the base and updated versions.

in text, compared with most desktop or web applications, to make their GUIs appear more attractive. Little information about the images used on the GUIs, however, can be derived from those files for apps targeting the iOS platform or for obfuscated Android apps.

In response to the challenges, we develop an approach called METER (**MOBILE TEST REPAIR**) to automatically repairing broken test scripts when mobile apps are updated. A key observation METER exploits is that, when fixing a broken test script, instead of scrutinizing the code, a developer usually looks at the app's GUI, tries to identify the parts that differ from what the test script expects, and then uses common sense to deduce likely fixes based on the GUI changes. Inspired by this observation, METER determines if a GUI has changed by analyzing its screenshots via computer vision (CV) techniques, and retains or repairs a test action based on the analysis result. Since METER only relies on screenshots to repair test scripts, it is applicable to apps targeting open or closed source mobile platforms including Android and iOS.

METER uses the behavior a test script triggers on the base version app as the reference, or oracle, to decide whether the script's execution on the updated version app is as expected. What METER implements is essentially a form of *reference testing* [20]. Compared with manually preparing the oracles, e.g., in the form of assertions, this approach is reasonably effective and much less expensive. Therefore, it has been adopted in many previous studies [21], [22] and tools [23], [24].

We implemented our approach in a tool also called METER. To evaluate METER's effectiveness and efficiency, we carried out experiments on 22 open-source Android apps used in existing literature and 6 iOS apps that are available counterparts of the chosen Android apps. In the experiments, repairs produced by METER helped preserve 63.7% and 38.8% of all the test actions broken by the GUI changes, respectively.

The basic idea and some preliminary results of METER were briefly reported in [25]. This work significantly extends the previous one in the following important aspects. First, we explain in detail the design of various components of the technique and how they fit together to achieve good repairing results. Second, we have devoted considerable effort to re-engineer the prototype of METER and make the tool more robust and more efficient. Third, we conduct a more comprehensive experimental evaluation to assess the effectiveness and efficiency of METER on more subject apps from various mobile platforms, in different settings, and in comparison to other GUI test repair tools.

The contributions this paper makes are as the following:

- *Technique*: To the best of our knowledge, METER is the first approach to repairing GUI test scripts for mobile apps that leverages computer vision techniques. Treating mobile apps as black-box systems enables METER to be applied to apps on not only open source platforms but also closed source ones where static analysis or reverse engineering tools are not available.
- *Tool*: We implement a supporting tool with the same name for METER.
- *Experiments*: We conduct an extensive experimental evaluation on METER. Repairing results on 28 real-world apps from both the Android and iOS platforms show that METER is both effective and efficient in repairing GUI test scripts for mobile apps.

The METER tool as well as the complete collection of experimental materials, including our results and the scripts to rerun the experiments, are available for download at:

https://github.com/metter2018/metter2018.

The rest of this paper is organized as follows. Section 2 uses an example to show how METER repairs test scripts from a user's perspective. Section 3 describes in detail the METER's repair mechanism based on computer vision techniques and the GUI matching relation. Section 4 discusses the experiments we conducted to evaluate METER and the results. Section 5 reviews existing works related to METER. Section 6 concludes the paper.

2 METER IN ACTION

In this section, we use a file management app for the Android platform, named *OI File Manager*, to illustrate from a user's perspective how METER repairs GUI test scripts for mobile apps. Figure 1 displays the screenshots of the app in version 2.0.5 (base version) and version 2.2.2 (updated version), respectively.

Listing 1 shows excerpts from two test scripts TS1 and TS2 for the base version of the app. Both scripts are written in Python for the Appium testing tool. TS1 tests the functionality of folder creation in the file system, and it starts with first clicking on the "More options" button (labeled with letter "c" in Figure 1a) and then clicking on the "Create folder" menu-item (Figure 1b). Particularly, the first test action locates the button by using the accessibility id of the button as the argument to invoke function `find_element_by_accessibility_id` provided by the testing engine, while the second test action obtains a handle of the menu item by first searching for the corresponding menu using its id and then locating the menu item at index 5 (0-based) within the menu. TS2 tests the functionality of file/-folder search, and it first locates the "Search" button (labeled by letter "b" in Figure 1a) using the button's accessibility id and then clicks on the button.

The GUI of *OI File Manager* is changed in the updated version: Buttons "Bookmarks" (labeled by letter "a" in Figure 1a) and "Search" were turned into items under menu "More options", and the order of items in menu "More options" is changed too, as depicted in Figure 1d. The changes break both previous test scripts: the second test action in TS1 will not be able to locate the menu item "Create folder" using the old index 5, and the first test action in TS2 cannot find an element with accessibility id "Search", unless menu "More options" is expanded already. It is especially worth noting that the first two actions of TS1 can still execute on the updated version app, but they will cause menu-item "Bookmarks", instead of "Create folder", to be activated. The test execution will not hang or crash in such a case, but it will silently exercise different behaviors than intended and produce misleading testing results.

Taking both versions of the app and the test scripts in Listing 1 as the input, METER is able to automatically produce the repaired test scripts as shown in Listing 2. In TS1', the way to locate and activate menu-item "Create folder" is modified for the updated version app, since the new menu has 8 items and "Create folder" appears on the bottom with index 7. In TS2', a new test action is inserted to the test script to expand menu "More options" first, and the original test action is changed to access the menu item at index 6.

```
    # TS1: To create a new folder.
1 driver.find_element_by_accessibility_id('More options').click()
2 driver.find_elements_by_id('android:id/title')[5].click()
  ...
    # TS2: To perform a file/folder search.
1 driver.find_element_by_accessibility_id('Search').click()
  ...
```

Listing 1: Test scripts for the base version.

```
    # TS1': To create a new folder.
1 driver.find_element_by_accessibility_id('More options').click()
2 driver.find_elements_by_id('android:id/title')[7].click()
  ...
    # TS2': To perform a file/folder search.
1 driver.find_element_by_accessibility_id('More options').click()
2 driver.find_elements_by_id('android:id/title')[6].click()
  ...
```

Listing 2: Repaired test scripts for the updated version.

3 THE METER APPROACH

Figure 2 illustrates an overview of the METER approach. Given a base version mobile app (*App*), a group of test scripts for it (*TS*), and an updated version of the same app (*App'*), METER 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 METER checks if the action preserves its intended behavior when executed on the updated version app via GUI screen matching. If yes, the test action does not need repairing; Otherwise, the test action is *broken* and METER constructs a sequence of test actions to replace it: The execution of the constructed replacement test actions on the updated version app should produce the same screen transition as triggered by the broken test action 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 METER uses to determine the matching relation between GUI elements and screens (Section 3.1), then explain how METER repairs test scripts based on such matching relation (Section 3.2), and in the end briefly describe the implementation details of a supporting tool for METER (Section 3.3).

3.1 GUI Matching

METER decides whether two test script executions conform to each other based on a matching relationship between their source and destination screens—a *screen* of an app refers to the part of the app's GUI that is visible to users at a particular point in time. The decision process is guided by a group of rules concerning the characteristics of mobile GUIs. The results of our experimental evaluations of METER, as discussed in Section 4, suggest these rules are reasonably effective on real-world mobile apps.

This section first explains the extraction of GUI elements from snapshots of app screens, or screenshots for short, and then defines the matching relation between both GUI elements and screens.

3.1.1 GUI Element Extraction

As the first step towards deciding if two screens match with each other, METER extracts GUI elements from their snapshots. This is done in two steps. First, METER identifies boundaries of GUI elements through contour detection; Then, METER classifies GUI elements as textual or graphical through optical character recognition.

3.1.1.1 Boundary Detection: Given that GUIs of apps could be arbitrarily complex, determining the boundaries of different GUI elements in a screenshot can be an extremely challenging task. In this work, METER employs a computer vision (CV) technique named contour detection to detect likely boundaries of GUI elements such as buttons and text fields. It filters out contours that less likely define GUI element boundaries based on a set of predefined rules.

Specifically, METER employs the Canny algorithm [26] to detect an initial set of contours. A detected contour, however, does not always delimit a relevant GUI element.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007664, IEEE Transactions on Software Engineering

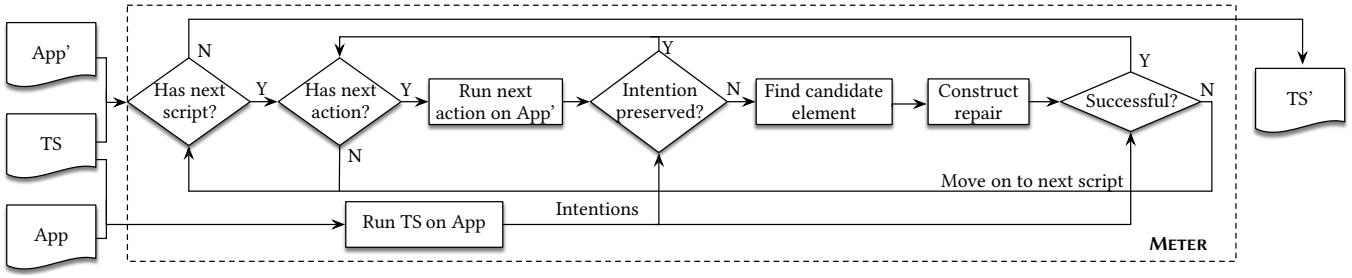


Figure 2: Overview of the METER approach.

TABLE 1: GUI element extraction rules. $\mathcal{C}$ denotes the set of detected contours, c the contour under consideration ($c \in \mathcal{C}$), H the height of the screen, and W the width of the screen. Given $c' \in \mathcal{C}$, $c'.h$, $c'.w$, and $c'.area$ denotes the height, the width, and the area of c', respectively. vDifference(c_1,c_2) calculates the vertical difference between the center of two contours, while hDistance(c_1,c_2) returns the horizontal distance between two contours.

PURPOSE	ID	PREDICATE	RULE[1]
contour identification	R1	isTooSmall(c)	$c.h < 10 \vee c.w < 10$
	R2	isTooLarge(c)	$c.h > 0.75H \vee c.w > 0.75W \vee c.h \times c.w > 0.75H \times W$
	R3	isTooSlim(c)	$c.w/c.h < 0.1$
	R4	isTooFat(c)	$c.h/c.w < 0.1$
	R5	isCovered(c, $\mathcal{C}$)	$\exists c' \in \mathcal{C} : (c \cap c').area/c'.area > 0.8 \wedge c'.area > c.area$
word grouping	R6	areAdjacent(c_1, c_2)	vDifference$(c_1, c_2) < 15 \wedge$ hDistance$(c_1, c_2) < 50)$
	R7	isLikelyTextual(c, $\mathcal{C}$)	$\exists c' \in \mathcal{C} :$ (c'.isTextual() $\wedge$ areAdjacent(c, c'))

[1] The values used in the rules are decided empirically based on the common size of texts, icons, and widgets in mobile apps, and have been tested empirically.

For example, given a piece of text in a screenshot, we are interested in detecting the boundaries of the words, while the Canny algorithm will report one contour for each character, rather than each word, since each character has its own boundary. To solve that problem, we dilate the detected edges by tripling their thickness, so that boundaries of characters in a word would intersect while those of words would not. Afterwards, we use an off-the-shelf implementation of contour detection algorithm[1] [27] on the dilated edges to get the contours.

METER filters out contours that unlikely delimit boundaries of relevant GUI elements based on a set of rules shown in Table 1. Rules R1, R2, R3, R4 are used to identify contours that are too small, too big, too slim, or too fat for GUI elements of interest, respectively. Contours satisfying any of these four rules are considered unrealistic for GUI element boundaries and excluded from further processes. Given that GUI testing is only concerned with triggerable GUI elements, i.e., GUI elements with which test scripts can interact, and that triggerable GUI elements almost never overlap, METER applies rule R5 to identify contours whose most area is covered by another larger contour, and it keeps only the largest contour among the ones that mostly overlap.

3.1.1.2 Classification of Textual and Graphical Elements: After getting contours representing boundaries of GUI elements, METER next classifies the GUI elements into two broad categories, namely textual and graphical GUI elements, with the help of an optical character recognition (OCR) engine. In particular, METER employs the OCR engine provided as part of the Microsoft Cloud-based Computer Vision API [28] (MSCVA) to extract texts from screenshots, and it determines which GUI elements are textual and which ones are graphical by checking whether the position and size of their recognized contours match with those of the extracted texts.

While MSCVA works quite well overall, and especially so on images mostly consisting of texts, some fine tuning to its application is still needed in certain situations. For example, MSCVA handles poorly images where texts are in a lighter color than the background, which unfortunately, is often the case with screenshots of mobile apps: Many apps can run in the night mode where dark backgrounds are used to make the display easier on the eyes. In view of that, before sending an image to MSCVA for OCR, METER first calculates the average RGB values of all pixels. If the sum of the average RGB values is less than a threshold value (empirically set to 150 by default), the image is considered to be mostly dark and the complement of the image, produced by changing each pixel of the image from (r, g, b) to $(255 - r, 255 - g, 255 - b)$, will be used as the input to MSCVA. Otherwise, the original image will be used. For example, the screenshots in Figure 1 have white texts on a black background, hence METER will use the complement of those images for OCR.

With the above processing, METER obtains most of the words in the screenshots. To match elements from different screenshots, the complete text on each element, or *element text*, rather than a group of individual words, is needed. METER reconstructs element texts by concatenating words from *adjacent* textual GUI elements, i.e., elements with their vertical locations being close to each other or the horizontal distance between them being small, and it relies on rule R6 from Table 1 to identify such adjacent GUI elements. While

1. https://opencv.org

useful, rule R6 alone turns out to be not enough. The reason is that, due to technical limitations, MSCVA may fail to recognize some words when it should, effectively cutting the element texts containing those words into pieces. To work around this problem, METER uses rule R7 to search for contours whose contents, if textual, might be used to form larger pieces of element texts. If no text was recognized for such a contour, METER assigns the dummy word "placeholder" to it. Dummy words can help with element text construction, but they do not carry any specific meaning with them and are not taken into account when calculating the similarity between two element texts (Section 3.1.2).

3.1.2 GUI Element Matching

After being extracted and classified into either textual or graphical, GUI elements are then compared to decide if they match with each other. This section illustrates how METER matches different types of GUI elements using different strategies.

3.1.2.1 Textual Element Matching: METER determines whether two textual elements match based on the similarity between their corresponding texts.

Given two textual GUI elements e_1 and e_2, let t_1 and t_2 be the two sets of their non-placeholder words, respectively. The similarity sim_{t_1,t_2} between t_1 and t_2 is calculated as $sim_{t_1,t_2} = |t_1 \cap t_2|/\max(|t_1|,|t_2|)$, and e_1 and e_2 are considered to match if sim_{t_1,t_2} is greater than or equal to a threshold value V_{tm}. V_{tm} is empirically set to 0.4 by default for the following two reasons. On the one hand, many textual elements contain just a few words, therefore even one different word between two elements can already result in a small similarity value. Setting the threshold value too high will prevent these elements from being matched. On the other hand, developers tend to make texts of different elements distinct from each other to reduce users' confusion, which leads to small similarity between texts of different elements. Setting the threshold value too low will risk matching unrelated texts.

3.1.2.2 Graphical Element Matching: Besides textual elements, a great deal of GUI elements like icons and images are graphical in nature. While it is very challenging to match graphical elements in the most general sense, METER implements a technique that turns out to be fairly effective in the context of GUI test repair for mobile apps. The key observation that motivates the technique is that, graphical materials used in mobile apps are typically organized separately as resource files, and they are mostly relocated, resized, or rotated, but seldom changed in other ways, during the evolution of apps. Accordingly, METER employs a computer vision technique called SIFT [29] to extract feature descriptor of the images on screenshots and measures the similarity of two images based on the numbers of feature descriptors they have in common. Since SIFT feature descriptors are invariant to uniform scaling and orientation [29], METER achieves a rather high success rate in identifying graphical elements that truly match. In particular, METER considers two images to match if the percentage of matched feature descriptors is greater than a predetermined threshold value V_{gm}. In view that most icons in mobile apps are simple drawings with only a couple of feature descriptors, METER conservatively sets the threshold percentage V_{gm} to 40% by default.

Given two GUI elements e_1 and e_2 of the same type, i.e, they are both textual or both graphical, we use $e_1 \sim e_2$ to denote that e_1 and e_2 match.

3.1.3 Matching of GUI Element Collection

We extend the definition of matching relation from GUI elements to collections of GUI elements in this section.

Given a collection $\boldsymbol{E}$ of GUI elements, we use $\boldsymbol{E}^t$ and $\boldsymbol{E}^g$ to denote the collections of textual and graphical elements within $\boldsymbol{E}$, respectively. Let $\boldsymbol{E}_1$ and $\boldsymbol{E}_2$ be two collections of GUI elements, the percentage MT of matched textual elements and the percentage MG of matched graphical elements between $\boldsymbol{E}_1$ and $\boldsymbol{E}_2$ can be computed as the following:

$$\mathrm{MT}_{\boldsymbol{E}_1,\boldsymbol{E}_2} = \frac{|\{e \in \boldsymbol{E}_1^t : \exists e' \in \boldsymbol{E}_2^t \rightarrow e \sim e'\}|}{|\boldsymbol{E}_1^t| + |\boldsymbol{E}_2^t|},$$

$$\mathrm{MG}_{\boldsymbol{E}_1,\boldsymbol{E}_2} = \frac{|\{e \in \boldsymbol{E}_1^g : \exists e' \in \boldsymbol{E}_2^g \rightarrow e \sim e'\}|}{|\boldsymbol{E}_1^g| + |\boldsymbol{E}_2^g|}.$$

Intuitively, $\boldsymbol{E}_1$ and $\boldsymbol{E}_2$ are said to match, denoted as $\boldsymbol{E}_1 \sim \boldsymbol{E}_2$, if and only if 1) enough of their textual elements match ($\mathrm{MT}_{\boldsymbol{E}_1,\boldsymbol{E}_2} > V_{cm1}$), 2) enough of their graphical elements match ($\mathrm{MG}_{\boldsymbol{E}_1,\boldsymbol{E}_2} > V_{cm1}$), or 3) a balanced amount of their textual and graphical elements match ($\mathrm{MT}_{\boldsymbol{E}_1,\boldsymbol{E}_2} > V_{cm2} \wedge \mathrm{MG}_{\boldsymbol{E}_1,\boldsymbol{E}_2} > V_{cm2}$), where the two threshold values V_{cm1} and V_{cm2} are empirically set to 0.7 and 0.4 by default, respectively. For collections with only a small number of textual or graphical elements, one or two unmatched elements would result in MT or MG being smaller than V_{cm1} in the first two conditions. The third condition compensates for that and allows such collections to be matched by accepting smaller, but balanced, MT and MG values.

3.1.4 Screen Matching

METER decides whether two screens match based on the matching relation between the GUI elements on them. Since screens of mobile apps are often split into regions with different priorities in receiving user inputs, METER takes a region-based approach to screen matching.

METER differentiates three kinds of regions on screens in this step: blocking regions, background regions, and content regions. *Blocking regions* correspond to cover-up windows like pop-up dialogs and slide-over menus. When invoked, a cover-up window often partially covers an existing screen. Although the screenshots before and after such an invocation may only differ in a small area, the enabled GUI elements on the two screens can be largely different from a user's perspective. Therefore, it is essential that METER is able to correctly identify blocking regions on screenshots. At present, METER recognizes three types of cover-up windows, namely dialogs, left slide-over menus, and right slide-over menus, and it employs a simple strategy to detect the blocking regions they produce. The basic idea is to match detected contours and their grayscale distributions against a group of predefined patterns related with different types of cover-up windows. For example, to detect a pop-up dialog, the strategy looks for a contour that is located near the center of the screen and has a grayscale distribution

Algorithm 1 Region-based Screen Matching

Input: $\boldsymbol{S}_1 = \langle \boldsymbol{S}_1^k, \boldsymbol{S}_1^t, \boldsymbol{S}_1^b, \boldsymbol{S}_1^c \rangle$, $\boldsymbol{S}_2 = \langle \boldsymbol{S}_2^k, \boldsymbol{S}_2^t, \boldsymbol{S}_2^b, \boldsymbol{S}_2^c \rangle$,
Output: **true** if $\boldsymbol{S}_1 \sim \boldsymbol{S}_2$; Otherwise, **false**.

1: **if** $(\boldsymbol{S}_1^k \neq \emptyset \vee \boldsymbol{S}_2^k \neq \emptyset)$ **return** $\boldsymbol{S}_1^k \sim \boldsymbol{S}_2^k$; **end if**
2: **if** $((\boldsymbol{S}_1^t \neq \emptyset \vee \boldsymbol{S}_2^t \neq \emptyset) \wedge \boldsymbol{S}_1^t \not\sim \boldsymbol{S}_2^t)$ **return false**; **end if**
3: **if** $((\boldsymbol{S}_1^b \neq \emptyset \vee \boldsymbol{S}_2^b \neq \emptyset) \wedge \boldsymbol{S}_1^b \not\sim \boldsymbol{S}_2^b)$ **return false**; **end if**
4: **return** $\boldsymbol{S}_1^c \sim \boldsymbol{S}_2^c$;

significantly different from the rest of the screen. METER detects at most one blocking region on each screenshot.

Background regions correspond to parts of screens that are often the same across screens. For example, the top area and the bottom area of a screen are often reserved for navigation bars or tab bars consisting of a list of horizontal elements. Given that it is common for such areas to stay unchanged across screens, METER does not regard those areas to be equally important in screen matching. To identify background regions on the top or bottom of a screen, METER first looks for contours that are within the top/bottom 20% of the screen and as wide as the screen, and then considers the minimum bounding box of those contours as delineating background regions. METER detects at most one background region on the top and one on the bottom of each screenshot. Areas not covered by blocking or background regions constitute the *content region* of a screen.

Given a screen $\boldsymbol{S}$, METER always tries to identify four different component regions on $\boldsymbol{S}$: a blocking region $\boldsymbol{S}^k$, a top background region $\boldsymbol{S}^t$, a bottom background region $\boldsymbol{S}^b$, and a content region $\boldsymbol{S}^c$. Each identified region is represented by the collection of textual and graphical GUI elements from that region, while regions not present in the screen are denoted using $\emptyset$. Two regions are said to match if and only if their collections of GUI elements match.

METER determines whether two screens $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ match, denoted as $\boldsymbol{S}_1 \sim \boldsymbol{S}_2$, by comparing their component regions using Algorithm 1: In case one of the screens has a blocking region, the two screens match if and only if the other screen also has a blocking region and the two blocking regions match (Line 1); If none of the two screens has a blocking region, but a background region from one screen has no match on the other, the two screens do not match (Lines 2 and 3); If neither screen has a blocking region and both their background regions match, the two screens match if and only if their content regions match (Line 4).

3.2 Test Script Repair

In this work, we use a pair $\langle loc, evt \rangle$ to denote a test action α, 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 α is executed. Following [30], we define a test script as a sequence $K = \alpha_1, \alpha_2, \ldots, \alpha_n$, where each α_i $(1 \leq i \leq n)$ is a test action.

Executing a test action $\alpha = \langle loc, evt \rangle$ on a screen $\boldsymbol{S}$ involves first applying the locator loc to identify on $\boldsymbol{S}$ a target GUI element to interact with and then triggering the event evt on the element. If the execution terminates successfully, it should transit the app to a (possibly different) destination screen. We denote the screen transition caused by the successful execution of α 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 rest of this section first presents the algorithm METER implements to repair test scripts written for a base version app so that their intention is preserved as much as possible on the updated version app (Section 3.2.1), then explains the model METER builds from the base version app to guide the repair of test scripts (Section 3.2.2).

3.2.1 Repair Test Script Construction

Algorithm 2 shows how METER repairs the test scripts step by step. The algorithm takes a base version app $\mathcal{P}$, its updated version $\mathcal{P}'$, and a list $\mathcal{K}$ of test scripts for $\mathcal{P}$ as the input. The whole repairing process involves two nested loops at a high level: The outer loop iterates through each test script $K \in \mathcal{K}$ and constructs a new test script Q for $\mathcal{P}'$ as the repairing result (Lines 2 through 37); The inner loop iterates through each test action α_i $(1 \leq i \leq n)$ from a particular test script $K = \alpha_1, \alpha_2, \ldots, \alpha_n \in \mathcal{K}$ and tries to derive a sequence q_i of test actions such that 1) test script $[q_1, q_2, \ldots, q_i]$ executes successfully on $\mathcal{P}'$, and 2) the transition caused by q_i on $\mathcal{P}'$ matches the intention of test action α_i on $\mathcal{P}$ (Lines 3 through 36). Here $[q_1, q_2, \ldots, q_i]$ denotes the test script produced by concatenating the sequences q_1 through q_i. Note that, during the repairing process, METER always reuses a test action if its intention is preserved on $\mathcal{P}'$ and only builds a new sequence of test actions when necessary. In case METER fails to build such a sequence q_j for a test action α_j $(1 \leq j \leq n)$ from K, it immediately returns $Q = [q_1, q_2, \ldots, q_{j-1}]$ as the repairing result for K.

More concretely, given a test action α from test script K to repair (Line 4) and α's intention ι (Line 5), the inner iteration always starts from a screen $curS$ of $\mathcal{P}'$ that matches $\iota.src$ (Line 6). Let ε be the GUI element α interacted with on $\iota.src$ (Line 7). METER first attempts α as-is on $curS$. If α is still executable and its intention is preserved, i.e., a GUI element at $\alpha.loc$ can be found on $curS$ and event $\alpha.evt$ triggered on the element will transit $\mathcal{P}'$ to a destination screen that matches $\iota.dest$ (Line 8), α is directly reused in the repair test script (Line 9).

In case α does not preserve the original intention ι on $\mathcal{P}'$, METER builds a sequence of test actions as the repairing result of α. During this process, METER assumes the functionalities of the app, and therefore their corresponding GUI elements, were not removed during the update, so it always tries to find a GUI element e in $\mathcal{P}'$ as the counterpart of ε and build a sequence of test actions around e that, when executed from $curS$, will satisfy the following two conditions: i) It will trigger event $\alpha.evt$ on e; ii) It will transit $\mathcal{P}'$ to a screen matching $\iota.dest$, in hope that the execution of following test actions will be unaffected.

To identify the counterpart e of element ε, METER iterates through a sorted set $\mathcal{E}$ of candidate GUI elements in $\mathcal{P}'$. If there exists an element ε' in $curS$ that matches

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007664, IEEE Transactions on Software Engineering

Algorithm 2 Test script repairing.

Input: $\mathcal{P}$: base version app; $\mathcal{P}'$: updated version app;
$\mathcal{K}$: list of original test scripts, with each test action associated with its intention on $\mathcal{P}$;
Output: $\mathcal{M}$: Map from each test action α in $\mathcal{K}$ to a triple $\langle \tau, src, dest \rangle$, where τ is the sequence of test actions derived from α for $\mathcal{P}'$ and it transits $\mathcal{P}'$ from screen src to screen $dest$.

1: init($\mathcal{M}$)
2: **for** $K \in \mathcal{K}$ **do**
3: **while** K.hasNext() **do**
4: $\alpha \leftarrow K$.next();
5: $\iota \leftarrow \alpha$.intention();
6: $curS \leftarrow \mathcal{M}(\alpha.pre).dest$ ▷ $\alpha.pre$ is the test action before α.
7: $\varepsilon \leftarrow$ ELE($\iota.src, \alpha.loc$)
8: **if** ELE($curS, \alpha.loc$) $\sim \varepsilon \wedge$ DEST($curS, \alpha$) $\sim \iota.dest$ **then**
9: $\mathcal{M}(\alpha) \leftarrow \langle [\alpha], curS,$ DEST($curS, \alpha$)$\rangle$
10: **continue**
11: **end if**
12:
13: $\mathcal{E} \leftarrow \emptyset$
14: **if** $\exists\, \varepsilon' \in curS : \varepsilon' \sim \varepsilon$ **then** $\mathcal{E}.append([\varepsilon'])$ **end if**
15: $\mathcal{E}.append($CANDSORTED($curS, \iota.src, \varepsilon$))
16: $\mathcal{E}.append($CANDSORTED($\mathcal{P}', \mathcal{P}, \varepsilon$))
17: $isFound \leftarrow$ **false**
18: **for** $e : \mathcal{E}$ **do**
19: $preK \leftarrow$ SCRIPTTOEQUAL($\mathcal{P}', curS,$ e.containingS())
20: $midS \leftarrow$ DEST(e.containingS(), $\langle e$.locator(), $\alpha.evt\rangle$)
21: $\langle postK, destS\rangle \leftarrow$ SCRIPTTOMATCHING($\mathcal{P}', midS, \iota.dest$)
22: **if** $preK == null \vee midS == null \vee postK == null$ **then**
23: BACKTRACK($K, \alpha, \mathcal{M}$)
24: **continue**
25: **end if**
26: **if** $preK$.isEmpty() $\vee$ $postK$.isEmpty() **then**
27: $\mathcal{M}(\alpha) \leftarrow$
28: $\langle preK + \langle e$.locator(), $\alpha.evt\rangle + postK, curS, destS\rangle$
29: $isFound \leftarrow$ **true**
30: **break**
31: **end if**
32: **end for**
33: **if not** $isFound$ **then**
34: **break**
35: **end if**
36: **end while**
37: **end for**
38: **return** $\mathcal{M}$

39: ELE($scr, locator$) ▷ Return GUI element selected by $locator$ on scr.
40: DEST(scr, α) ▷ Return destination screen of executing α on scr.
41: CANDSORTED(S, S', ele) ▷ Return GUI elements from S that are ▷ not matched with any element from S'. ▷ Sorted in decreasing order of similarity to ele.
42: SCRIPTTOEQUAL($P, src, dest$) ▷ Return *confirmed* sequence of test ▷ actions in P that transits P from src to $dest$. Return empty ▷ sequence if $src = dest$, or null if no such sequence is found.
43: SCRIPTTOMATCHING($P, src, dest$) ▷ Return $\langle l, dest'\rangle$, where l is a ▷ *confirmed* sequence of test actions, $dest'$ is a screen in P ▷ ($dest' \sim dest$), and l transits P from src to $dest'$. ▷ l is empty if $src \sim dest$, or null if none is found.
44: BACKTRACK($K, \alpha, \mathcal{M}$) ▷ Backtrack by killing the current testing ▷ process and executing the partial repair constructed for K so ▷ far, i.e., the repairs stored in $\mathcal{M}$ for K's test actions until α.

ε, ε' is considered as a candidate with the highest priority (Line 14). GUI elements from $\mathcal{P}'$ that are not matched with any elements in $\mathcal{P}$ yet are also considered as candidates, but with lower priority (Line 15 and 16). Given a screen $\boldsymbol{S}'$ in $\mathcal{P}'$, let $\boldsymbol{S}$ be its matching screen in $\mathcal{P}$, the set $\delta_{\boldsymbol{S}'}$ of elements on $\boldsymbol{S}'$ that do not match any element on $\boldsymbol{S}$ is calculated as $\delta_{\boldsymbol{S}'} = \{e | e \in \boldsymbol{E}' \wedge \nexists e' \in \boldsymbol{E} : e \sim e'\}$, where $\boldsymbol{E}'$ and $\boldsymbol{E}$ are the collections of GUI elements on $\boldsymbol{S}'$ and $\boldsymbol{S}$, respectively. In case screen $\boldsymbol{S}'$ does not match any screen in $\mathcal{P}$, $\delta_{\boldsymbol{S}'}$ is equal to $\boldsymbol{E}'$. Note that, since the modification to a GUI element can be arbitrary, METER does not require candidate GUI elements to be similar to ε in this step. Also note that METER examines all local candidates (i.e., candidates on $curS$) before looking at global ones (i.e., candidates on other screens of $\mathcal{P}'$) so as to favor local matches for ε.

For each $e \in \mathcal{E}$, METER attempts to 1) construct a sequence $preK$ of test actions to navigate $\mathcal{P}'$ from $curS$ to the containing screen of e (Line 19), 2) trigger $\alpha.evt$ on e and transit $\mathcal{P}'$ to another screen $midS$ (Line 20), and 3) construct another sequence $postK$ of test actions to navigate from $midS$ to a screen $destS$ that matches $\iota.dest$ (Line 21). Here, the construction of both $preK$ and $postK$ involves exploring the screen transition relation observed during previous test executions on $\mathcal{P}'$ to find the shortest test action sequences to achieve the desired navigation, executing those test action sequences on $\mathcal{P}'$, and returning the test action sequences or $null$ depending on whether they do achieve the desired screen transitions. The construction using element e fails immediately if any of the three steps is unsuccessful (Line 22). In that case, to undo the possible changes caused by the attempted test action sequences, METER backtracks to $curS$ by killing the current testing process and executing the partial repair constructed for K so far, i.e., the repair action sequences stored in $\mathcal{M}$ for K's test actions until α (Line 23). Otherwise, the concatenation of $preK$, $\langle e.locator, \alpha.evt\rangle$, and $postK$ constitutes a sequence of actions that satisfies the abovementioned conditions i) and ii). To reduce the amount of manual effort possibly required for testers to confirm or maintain the resultant test scripts, METER strives to keep the differences between the test scripts before and after repairing small. Therefore, if the constructed sequence is simple enough, in the sense that either $preK$ or $postK$ is empty (Line 26), METER accepts it as the repair for α (Line 28); Otherwise, the repair of α fails, and METER returns the constructed test action sequences as the repairing result for K before it starts to repair the next test script (Line 34).

Consider app OI File Manager and its test script TS2 shown in Section 2 for example. The execution of TS2's first test action, denoted as α_0 here, will fail on the updated version app, since button "Search" has been turned into a menu item in that version and there is no GUI element with the desired accessibility ID on the screen when α_0 is executing. To construct a replacement for α_0, METER will look for a counterpart of button "Search", first locally within the current screen and then globally in other screens of the app. Since menu item "Search" is the only GUI element around which a simple enough sequence of test actions could be constructed to achieve the same screen transition as α_0's, the menu item will be picked as the counterpart of button "Search" and the test action sequence constructed will be the repair for α_0.

3.2.2 Intentions as A Means to Model App Behaviors

Given two screens $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ of $\mathcal{P}'$, to effectively construct a sequence of test actions to transit $\mathcal{P}'$ from $\boldsymbol{S}_1$ to $\boldsymbol{S}_2$, as required by functions SCRIPTTOEQUAL and SCRIPTTOMATCHING invoked in Algorithm 2, METER goes through the following three-step process: i) To identify the matching screens $\boldsymbol{S}_a$ and $\boldsymbol{S}_b$ from $\mathcal{P}$ for $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$; ii) To construct a list Γ of test action sequences that may transit

$\mathcal{P}$ from $\boldsymbol{S}_a$ to $\boldsymbol{S}_b$; iii) To dynamically validate each $\gamma \in \Gamma$ to find a sequence that actually realizes the screen transition from $\boldsymbol{S}_1$ to $\boldsymbol{S}_2$ on $\mathcal{P}'$.

Here the construction of test action sequences in step ii) is based on the observation that, given two test actions α_1 and α_2 for $\mathcal{P}$ and their intention ι_1 and ι_2, if $\iota_1.dest \sim \iota_2.src$, executing the sequence α_1, α_2 *may* transit $\mathcal{P}$ from $\iota_1.src$ to $\iota_2.dest$. In particular, based on the intention of all original test actions, METER is able to construct a graph $\mathcal{G}_{\mathcal{P}}$ with screens being the vertices and intentions being the edges that models possible behaviors of $\mathcal{P}$. Next, given two screens $\boldsymbol{S}_a$ and $\boldsymbol{S}_b$ from $\mathcal{P}$ as the target source and destination screens, a (breadth-first) search can be performed to find sequences of intentions connecting $\boldsymbol{S}_a$ and $\boldsymbol{S}_b$ on the graph, and METER will be able to obtain a collection Γ of test action sequences that may transit $\mathcal{P}$ from $\boldsymbol{S}_a$ to $\boldsymbol{S}_b$ by mapping those intentions to their test actions. The dynamic validation in step iii) is necessary, since there is no guarantee that the test action sequences in Γ will indeed realize the desired transitions on $\mathcal{P}'$.

3.3 METER Implementation

We have implemented the approach described above into a tool, also named METER, to automate the repair of GUI test scripts for mobile apps. The current implementation of METER exploits the Appium test automation framework [8] to drive the mobile apps under consideration, and relies on the Appium Python Client library [31] to run test scripts written in Python. It, however, is worth noting that, METER is not restricted to any specific framework or test scripting language and can be easily extended to support other test automation techniques.

For contour detection and optical character recognition, METER uses the OpenCV library (Version 3.1) [32] and the Microsoft Azure OCR API [33], respectively. The architecture of METER has been designed to enable easy switch between libraries, so that future developments in computer vision and optical character recognition techniques could be adopted by METER for better performance with ease.

4 EVALUATION

To evaluate, and put in perspective, the effectiveness and efficiency of METER, we conducted experiments that apply METER to repair GUI test scripts for real-world apps. Based on the experimental results, we address the following research questions:

- **RQ1:** How effective is METER in repairing test scripts? In RQ1, we evaluate from a user's perspective the repairing results produced by METER in terms of high level metrics like the number of test actions repaired and the number of test actions preserved.
- **RQ2:** How efficient is METER in repairing test scripts? In RQ2, we investigate the time cost of the overall test script repairing process with METER as well as its individual steps.
- **RQ3:** How robust is METER in repairing test scripts? In RQ3, we examine to what extent different threshold values adopted in GUI matching (Section 3.1.2) impacts METER's effectiveness.
- **RQ4:** How effective is METER in comparison with other test script repair tools for mobile apps? In RQ4, we compare METER with the CHATEM [34] model-based approach.
- **RQ5:** Is METER applicable to test script repair on other mobile platforms? In RQ5, we gather preliminary evidence to confirm our conjecture that METER performs also well on the iOS platform.

Comparison with CHATEM. Various approaches have been proposed to repairing GUI test scripts in the past, among which model-based approaches like ATOM [35] and CHATEM [34] have produced promising results. While both ATOM and CHATEM require precise models of the app under consideration as the input, there is an important difference between the types of models they use to drive the repairing process and how the models are prepared: ATOM requires as the input an event sequence model (ESM) that abstracts sequences of events on the GUI of the base version app and a delta ESM that abstracts the changes made to the based version app, and the application of ATOM involved manually constructing the models [35]. In comparison, CHATEM semi-automatically constructs ESMs for the two versions of a subject app based on the output of model extraction tools like Gator [36], and it extracts the differences between the two ESMs in an automatic fashion to guide the repairing process. Given that less human effort is required in preparing the input models for CHATEM and the performance delivered by the two approaches in repairing test scripts are alike [34], we compare METER with CHATEM in our experiments. Section 5 reviews more related works in GUI test script repair.

4.1 Subjects

To answer RQ1 through RQ4, we collected in total 22 Android apps as subjects from previous studies on mobile apps. The comparison between METER and CHATEM to address RQ4, however, was based on just 9 of those apps since Gator was not able to extract any models from the other apps within the given time. 6 available counterparts of those apps on the iOS platform were used to address RQ5.

4.1.1 Subjects on the Android platform

4.1.1.1 The apps: In view that previous studies on mobile apps systematically collected their subject apps to reflect the diversity of mobile apps and to reduce the influence of our bias in subject selection, we first gathered a pool of 119 unique apps for the Android platform from six papers on mobile apps published in the past 2 years at major software engineering conferences like ICSE, FSE, and OOPSLA: [37] (68), [38] (15), [39] (8), [40] (14), [41] (25), [42] (10). All the apps were then manually checked one by one. Among the 119 apps on the initial list, 61 were excluded from the experiments because we cannot find two different versions of them on the Internet; 5 were excluded because their names (e.g., "browser", "editor", and "contact") are not specific enough to be exactly matched with any app; 16 were excluded because they have obvious functional defects; 11 games were excluded because the randomness and time-sensitiveness involved in their behaviors make them unsuitable to be tested using scripts; and 1 was excluded

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007664, IEEE Transactions on Software Engineering

because its GUI occupies just a portion of the whole device screen, while METER is not designed to operate based on only parts of the screenshots. In the experiments, we collect coverage information either on the bytecode level using the Soot static program analysis framework, or on the source code level using a home-brewed tool. 3 apps that cannot be correctly handled by either of these techniques were therefore also excluded. In this way, 22 apps were collected as the subjects for the Android platform.

4.1.1.2 The base and updated versions: For each app, the latest version is always used in the experiments as the updated version, and we determine which other version, hopefully with different behaviors than the latest one, is to be used as the base version via testing.

Since none of the 22 Android apps was equipped with GUI tests, we first downloaded the latest version of each app (as of December 2018) and manually constructed test scripts for it by following the practice in [42]. Particularly, we recruited nine postgraduate students in Computer Science, each with at least two-year experience in mobile application development and testing; Each student was randomly assigned with 2 to 3 apps, and he/she needs to write test scripts to achieve around 50% statement coverage for each app. The involvement of the nine postgraduate students in this task is reasonable: Since all these students are experienced in writing GUI tests for mobile apps, they are likely to perform similarly to industry personnel in approaching such tasks [43]. The authors exerted no influence on the test construction process except for setting the target level of statement coverage, which is comparable with or higher than the levels reported in related works [38]–[42].

Next, for each app, we executed the test scripts on its earlier versions in reverse chronological order until a version that produces different test results (i.e., passing or failing) than the latest version was found—that version was then used as the base version of the subject app. We successfully determined in this way the base and updated versions for 17 apps. As for the other 5 apps, the test scripts produced the same results on all the available versions. We therefore used the latest-but-four versions as the base versions for them. Here we chose not to use adjacent versions as the base and updated versions so that the differences between the two versions are likely greater and the repairing task is likely more challenging.

4.1.1.3 Test scripts to repair: The same group of students then manually revised the test scripts originally written for the updated version apps so that they achieve comparable percentage of statement coverage on the base version apps. The revised test scripts are the ones to be repaired in our experiments.

Table 2 lists the basic information of the 22 apps. For each app, the table gives the app ID (ID), the name (APP), the number of test scripts we prepared for its base version (#K), the number of test actions in all those test scripts (#A), and the percentage of statements covered by the test scripts on the base version (SC). For both the base (BASE) and updated (UPDATED) versions of each app, the table also lists the version number (V) and the size in number of lines of code (LOC).

In total, 384 test scripts with 4368 test actions were prepared for the 22 base version apps, covering 53.9% of the statements. Test scripts for apps S5 and S16 did not reach the expected coverage level. App S5 enables a phone to answer calls automatically, while app S16 enables users to connect a desktop computer's keyboard to an Android device and use that keyboard to control the device. Both apps require external events, typically triggered by human, to exercise their core functionalities. As the test scripts we prepared in the experiments simulate no external events, we leave a major part of the two apps untested.

4.1.1.4 The GUI changes: We manually examined the GUIs of the base and updated version apps to identify the changes and compared the execution of the subject test

TABLE 2: Android Apps used as subjects in the experiments.

ID	APP	BASE		UPDATED		#K	#A	SC
		V	LOC	V	LOC			
S1	A Time Track	0.21	3880	0.23	3938	21	97	58.1%
S2	A2DP Volume	2.12.4	8529	2.12.9	8641	8	83	50.1%
S3	AnkiDroid	2.6	66146	2.8.3	66643	29	811	53.3%
S4	AnyMemo	10.8	27899	10.10.1	10497	31	239	53.4%
S5	Auto answer & callback	1.9	4135	2.3	4314	17	163	36.4%
S6	Budget Watch	0.18	4514	0.2	4591	14	106	65.1%
S7	c:geo	20171010	76078	20171217	79481	32	659	58.2%
S8	Dumbphone Assistant	0.4	1076	0.5	1120	6	17	67.2%
S9	K-9 Mail	5.207	87791	5.4	90187	39	672	56.8%
S10	KeePassDroid	2	14927	2.2	17185	29	195	47.4%
S11	Lighting Web Browser	4.4.2	19520	4.5.1	21110	23	345	49.2%
S12	Notepad	1	1198	1.12	1370	10	64	71.5%
S13	OI File Manager	2.0.5	8976	2.2.2	9304	29	212	47.4%
S14	Open Camera	1.40.0	29508	1.42.2	32934	17	159	50.8%
S15	Pedometer	5.16	13631	5.18	13955	5	95	50.6%
S16	Remote Keyboard	1.6	1666	1.7	1672	9	42	28.8%
S17	SMS Scheduler	1.37	5965	1.48	6564	19	82	49.1%
S18	Soundboard	0.9.1	581	0.9.2	665	4	13	67.8%
S19	SuperGenPass	2.2.3	1878	3.0.0	1936	5	31	59.2%
S20	SysLog	2.0.0	2718	2.1.1	2493	5	35	67.6%
S21	Tasks Astrid To Do List	4.9.14	24784	5.0.2	24307	22	202	48.3%
S22	Who Has My Stuff?	1.0.24	1764	1.0.30	1887	10	46	64.8%
Overall			407164		404794	384	4368	53.9%

TABLE 3: GUI changes between the base and updated version apps and the numbers of test scripts failed due to those changes.

ID	#GUI ELEMENT						#BROKEN TEST SCRIPT					
	AFF	DEL	ADD	MOD	NBHV	BHV	AFF	DEL	ADD	MOD	NBHV	BHV
s1	0	0	0	0	0	0	0	0	0	0	0	0
s2	5	0	1	4	4	0	2	0	0	2	2	0
s3	4	1	0	3	1	2	5	1	0	4	2	2
s4	5	0	1	4	4	0	6	0	0	6	6	0
s5	2	0	1	1	1	0	10	0	0	10	10	0
s6	3	0	1	2	2	0	3	0	0	3	3	0
s7	0	0	0	0	0	0	0	0	0	0	0	0
s8	3	0	1	2	2	0	4	0	0	4	4	0
s9	6	1	0	5	4	1	11	0	0	11	9	2
s10	0	0	0	0	0	0	0	0	0	0	0	0
s11	9	0	1	8	7	1	9	0	0	9	8	1
s12	5	0	2	3	2	1	6	0	0	6	5	1
s13	4	1	0	3	3	0	17	4	0	13	13	0
s14	6	0	0	6	6	0	13	0	0	13	13	0
s15	1	0	0	1	1	0	1	0	0	1	1	0
s16	4	0	1	3	3	0	5	0	0	5	5	0
s17	1	0	0	1	0	1	1	0	0	1	0	1
s18	2	1	0	1	1	0	2	1	0	1	1	0
s19	4	0	0	4	0	4	3	0	0	3	0	3
s20	2	1	0	1	1	0	3	1	0	2	2	0
s21	4	0	0	4	4	0	1	0	0	1	1	0
s22	3	2	0	1	1	0	2	1	0	1	1	0
Total	73	7	9	57	47	10	104	8	0	96	86	10

TABLE 4: Measures of ESMs for the Android apps.

ID	BASE							UPDATED						
	#	#M	#G	#C	#R	#D	#T(h)	#	#M	#G	#C	#R	#D	#T(h)
S2	73	21	159	29	23	107	6.9	74	21	187	30	23	134	7.1
S5	65	31	64	21	13	30	5.3	66	31	86	22	13	51	5.5
S8	14	8	76	2	4	70	2.0	14	8	112	2	4	106	2.3
S12	27	10	42	11	6	25	2.2	29	12	44	11	6	27	2.4
S16	30	21	31	9	0	22	2.0	31	22	31	9	0	22	2.1
S17	81	13	252	30	38	184	9.5	81	13	272	30	38	204	9.7
S18	14	9	7	2	3	2	1.3	25	8	18	2	15	1	3.3
S20	24	14	12	0	10	2	2.9	22	12	11	0	10	1	2.8
S22	51	36	20	10	5	5	4.0	51	37	15	10	4	1	3.9
Total	379	163	663	114	102	447	36.1	393	164	776	116	113	547	38.9

scripts on both version apps to discover the actual influence the GUI changes have on those tests. Table 3 shows the numbers of GUI elements affected (AFF), i.e., deleted (DEL), added (ADD), or modified (MOD), when the apps evolved, and the breakdown of the modified GUI elements into two categories based on whether the modifications affected only a GUI element's non-behavioral aspects (NBHV)—e.g., the position, text, and/or image—or they affected also a GUI element's behavior (BHV)—e.g., the screen transition it triggers and/or the user action to trigger the transition. The table also lists the number of test scripts that become broken due to each category of GUI changes.

Here, we deem a GUI element as being deleted or added if its corresponding functionality is removed from or added to an app, and we regard a GUI element as being modified if its appearance and/or behavior is different but its main functionality largely remains the same. Consider OI File Manager's two screens shown in Figure 1a and 1b for example. During their evolution to the screens shown in Figure 1c and 1d, no GUI element was deleted or added but the non-behavioral aspects of three GUI elements were modified: Buttons "Bookmark" and "Search" were turned into two menu items, and the index of menu item "Create folder" was increased by 2.

Overall, 73 GUI elements were deleted, added, or modified during the updates, breaking 104 test scripts. More importantly, 86, or 82.7%, of the broken test scripts failed due to changes that do not affect the behaviors of existing GUI elements. For those test scripts, METER should be able to preserve the intentions of their component test actions on the updated version apps in an automatic way.

4.1.2 Subjects and their ESMs for running CHATEM.

Given a subject app, we followed the practice in [34] to produce the ESM for the app. Specifically, we 1) ran the Gator tool on the app for at most 180 minutes to produce an initial model for the app, 2) observed the execution of the available test scripts on the app, 3) revised or removed GUI elements and transitions in the initial model as appropriate, and 4) added GUI elements and transitions that were exercised by the tests but missing from the model. The same process was applied to prepare the ESMs for both the base and updated version apps. In the end, Gator helped produce initial models for 13 apps. CHATEM failed to correctly parse the model files produced by Gator on 4 of those apps and managed to run to completion on the remaining 9 apps.

Table 4 lists, for the base (BASE) and updated (UPDATED) versions of each remaining app (ID), the numbers of GUI elements contained in the result ESMs (#), the numbers of GUI elements in the result ESMs that were manually created (#M), the numbers of GUI elements in the result ESMs that were automatically generated by Gator (#G), the breakdown of #G into three parts, namely the numbers of Gator generated GUI elements that were correct (#C), needed revising (R), and should be deleted (#D), and the amounts of time in hours required to construct the result ESMs (T). Note that a GUI element needed revising if and only if at least one transition on that element was missing, needed revising, or should be removed, and we always have # = #M + #C + #R.

In the end, 18 ESMs containing in total 772 GUI elements were created in 75.0 hours, averaging to 4.2 hours per ESM and 0.1 hours per GUI element. Among all the GUI elements contained in the result ESMs, 327 were added manually, while 230 and 215 were retained and revised from the Gator extracted models, respectively; 994 Gator extracted GUI elements were either duplicates or incorrect and therefore removed. Such results suggest that, even with the help from automated model extraction tools like Gator, significant amount of manual effort is still needed in applying model-based test script repair.

4.1.3 Subjects on the iOS platform.

We collected counterparts of the 22 Android apps on the iOS platform as the subjects for addressing RQ5. Among the 22 Android apps, 16 were excluded since they have no implementation on iOS and another one was excluded because its iOS implementation has only one version. This left us with 6 iOS apps in total.

We adapted the test scripts written for the corresponding Android apps to get the initial sets of test scripts for the iOS apps, and followed the same process as described in Section 4.1.1 to identify the base and updated versions of each app. The initial sets of test scripts were then revised to suit the base version apps. Due to the closed-source nature of iOS apps, no coverage information, however, is available for the test scripts.

Table 8 lists, for each of the 6 iOS apps, the name of the app (APP), the base (V_B) and updated (V_U) versions, the number of test scripts we prepared for its base version (#K), and the number of test actions in those test scripts (#A).

The numbers of subject iOS apps, test scripts, and test actions used in these experiments are smaller than those in the experiments targeting the Android platform, mainly because that, the subject Android apps were mostly open source apps and their iOS versions, when in rare cases do exist, tend to support only a subset of the functionalities provided by their Android counterparts. Such limited set of subject apps poses a major threat to the external validity of our findings in the experiments. We discuss the threat in Section 4.5.

4.2 Measures

A test script repairing tool takes a base version app $\mathcal{P}$, its updated version $\mathcal{P}'$, and a set $\mathcal{K}$ of test scripts written for $\mathcal{P}$

as the input, and produces a set $\mathcal{Q}$ of test scripts for $\mathcal{P}'$ as the repairing results of $\mathcal{K}$.

Given such a tool and an input test script $K = \alpha_1, \alpha_2, \ldots, \alpha_n$ ($K \in \mathcal{K}$) to repair, let q_i be the sequence of test actions produced by the tool for α_i as its repair ($1 \leq i \leq n$) and $Q = q_1, q_2, \ldots q_n$ be the repairing result for K ($Q \in \mathcal{Q}$), we can use the metrics defined as follows to measure the effectiveness of the tool in repairing K:

- NAE: The Number of test Actions Executable counts test actions in Q that can execute successfully on $\mathcal{P}'$;
- NAT: The Number of test Actions reTained counts test actions from K that are copied into Q and execute successfully on $\mathcal{P}'$; The NAT can be calculated as $|\{i : 1 \leq i \leq n \wedge q_i = [\alpha_i]\}|$.
- SC: the percentage of statements in $\mathcal{P}'$ covered by the repairing result Q;
- NAP: The Number of test Actions Preserved counts test actions from K that are *retained with their semantics preserved*; The NAP can be calculated as $|\{i : 1 \leq i \leq n \wedge q_i = [\alpha_i] \wedge \tau_{q_i} \rightsquigarrow \iota_{\alpha_i}\}|$, where τ_{q_i} denotes the screen transition triggered by q_i and ι_{α_i} denotes the intention of α_i.
- NAR: The Number of test Actions Repaired counts test actions from K whose replacements were successfully generated during repairing and can execute successfully on $\mathcal{P}'$. The NAR can be calculated as $|\{i : 1 \leq i \leq n \wedge q_i <> [\,] \wedge q_i <> [\alpha_i]\}|$.

Here, metric NAE calculates the effective size of a repaired test script; Metrics NAT and SC reflect two common goals of test script repair, namely to reuse test actions from the input test script and to avoid decrease in code coverage; Metric NAP ensures the intended semantics of the input test actions is taken into account while evaluating the repairing result, which is important since a test action reused from K may have depreciated value if it exercises different functionalities than before and thus loses the human knowledge encoded in it; Metric NAR measures the effort required to achieve the actual repairing results.

We use all these five metrics to empirically evaluate the effectiveness of METER in this work. Note that, when it comes to test script repairing with METER in particular, the NAT will always equate to the NAP, since METER only retains a test action when its intention is preserved, and the NAR will only equate to zero if no repair is really needed for the input test script. In comparison, if K is repaired with a *null test repair tool*, i.e., a test repair tool that simply returns the input test scripts as the repairing results, the NAT is often greater than the NAP, since some retained test actions may not preserve their original semantics, while the NAR will always be zero.

To evaluate the efficiency of METER in repairing K, besides the commonly used metric T that measures the overall repairing time, we also use the following metrics:

- T_O: The time for recording the intention of each test action in K on $\mathcal{P}$;
- T_C: The time for checking whether each test action in K preserves its intention on $\mathcal{P}'$;
- T_R: The time for constructing repairs for test actions in K that do not preserve the intention on $\mathcal{P}'$.

As explained in Section 3.2.1, given a test action α that interacts with a GUI element ϵ on screen S of $\mathcal{P}$, let S' be the match of S in $\mathcal{P}'$, if METER determines α does not preserve its intention on $\mathcal{P}'$, it searches for a counterpart ϵ' of ϵ within $\mathcal{P}'$ and constructs a sequence of test actions around ϵ' as the repair for α. Since the search here is first conducted *locally* within S' and then *globally* in other screens of $\mathcal{P}'$, we break T_R into the following two parts:

- T_L: The time for repair construction through local search for ϵ';
- T_G: The time for repair construction through global search for ϵ'.

We extend the definitions of all these metrics in a natural way so that they can be used to measure the repairing of the set $\mathcal{K}$ of test scripts, and we measure all the times as wall-clock time in minutes, unless otherwise specified.

4.3 Experimental Protocol

All the experiments were run on a Macbook Air laptop running Mac OS X10 with one Quad-core 1.7GHz CPU and 8 GB memory. Subject Android apps are executed on a Samsung Galaxy S6 phone and iOS apps on an iPhone XX.

4.3.1 *Test script repair with* METER *on Android and iOS*

In each experiment with METER on an Android or iOS app, the tool takes the base version $\mathcal{P}$ and the updated version $\mathcal{P}'$ of the app as well as a set $\mathcal{K}$ of test scripts written for $\mathcal{P}$ as the input, and produces a set $\mathcal{Q}$ of test scripts for $\mathcal{P}'$ as the repairing results of $\mathcal{K}$. More concretely, METER first runs $\mathcal{K}$ on $\mathcal{P}$ and records the intention of each test action from $\mathcal{K}$, and then it produces the repaired set $\mathcal{Q}$ of test scripts in a fully automatic fashion. All metrics defined in Section 4.2 are measured and recorded in the process.

We also ask two of the nine postgraduate students, including the author of $\mathcal{K}$, to manually check the execution of $\mathcal{Q}$ on $\mathcal{P}'$ and conservatively mark a test action α as being *correctly repaired* if and only if both students reach a consensus that the replacement test action sequence q for α indeed realizes α's intended semantics.

To put the repairing results produced by METER into perspective, we also repeated the experiments on a *null* test script repair tool (Section 4.2). Since the null test script repair tool does not change the input test scripts at all, the associated metric values effectively characterize the results of running $\mathcal{K}$ on $\mathcal{P}'$.

4.3.2 *Robustness of* METER

In its current implementation, METER's behavior depends on several parameters adopted in GUI matching (Section 3.1.2): Two textual elements are considered to match if their similarity is greater than $V_{tm} = 0.4$; Two graphical elements are considered to match if the percentage of their feature descriptors that match is greater than $V_{gm} = 40\%$; Two collections of GUI elements match if 1) more than $V_{cm1} = 70\%$ of their textual or graphical elements match or 2) the percentage of their matching textual and graphical elements are both greater than $V_{cm2} = 40\%$. To understand whether and how these parameters' values influence METER's effectiveness, we modify one parameter's value at a time and rerun the experiments on all the subject

Android apps. In Section 4.4.3, we report how changing each parameter affects the repairing results in terms of major effectiveness measures defined in Section 4.2.

4.3.3 Comparison with CHATEM

CHATEM's repairing results naturally depend on the completeness of the input ESMs, which in turn depends on how much manual effort is put into the construction of the ESMs. To better understand how CHATEM compares with METER when ESMs of various completeness levels are used, we conduct two experiments with CHATEM on each Android app and compare both results with that produced by METER: One experiment uses the final ESMs constructed in Section 4.1.2 as the input, and the other uses the same ESMs but without the manually added GUI elements, i.e., the ESMs constructed by just revising and deleting elements from the models extracted by Gator.

In each experiment with CHATEM on an Android app, we first feed the ESMs for the base and updated versions of the app and the test scripts for the app to CHATEM, and then gather the repaired test scripts. Next, we run those repaired test scripts on the updated version app and check whether the intention of each test action is preserved as in experiments with METER. Corresponding metrics are also measured and recorded during the process.

4.4 Experimental Results

This section reports on the results from experiments.

4.4.1 RQ1: Effectiveness

Table 5 lists, for each Android app, the numbers of test actions executable (NAE), preserved (NAP), and repaired (NAR) in the result test scripts produced by METER, the statement coverage achieved by those test scripts (SC), and the number of test scripts that are completely repaired by METER ($\#K_{\mathrm{R}}$). Each entry in column NAR is in form $x(y/z)$, where x is the number of test actions repaired, while y and z are the numbers of test actions that were successfully repaired via local and global search for candidate elements (Section 3.2.1), respectively. The table also lists the values of the same metrics for a NULL test script repair tool (Section 4.2), which reflect the results of running the original test scripts on the updated version app.

It is worth mentioning that, since the intended semantics of each test action on the base version app is always clearly defined, both graduate students reached a consensus on whether a test action was correctly repaired or not *in all cases*. For the null repair tool, instead of the number of test scripts completely repaired, which is equal to the number of test scripts not affected by the GUI changes, the number of test scripts broken by the GUI changes ($\#K_{\mathrm{B}}$) is reported. A test script is broken if at least one of its test action is broken (Section 3). Recall that, with METER, the NAT always has the same value as the NAP since METER only retains a test action when its intention is preserved, and that, with the null repair tool, the NAT is always equal to the NAE while the NAR is always equal to zero. These metric values are therefore omitted from the table.

Before being repaired by METER, 1063 (=4368-3305) test actions were no longer executable on the updated version app, affecting 104 of the 384 test scripts for the apps and causing the statement coverage to drop from 54.6% to 46.4%. Among the 3305 test actions that were still executable, only 3211, or 73.5% (=3211/4368) of the total amount, preserved their intention on the updated version apps.

Thanks to METER, 4159 test actions from the repaired test scripts were executable, covering 52.8% of the statements of the updated version apps. Compared with the null repair tool, METER preserved 737 (= 3948 - 3211) more transitions from the original test scripts, which amounts to 63.7% (=737/1157) of all the test actions broken by the GUI changes. METER achieved such high intention preservation rate by completely repairing 82 of the 104 broken test scripts and successfully repairing in total 203 broken test actions, averaging to 9.2 repairs per app or 2.0 repairs per broken script. Among all the repaired test actions, 186 and 17 were successfully repaired via local and global search for the candidate GUI elements, respectively, which suggests that both types of searches are essential for the success of METER. It is also worth noting that, the value of the repairing process is not limited to repairing the broken test actions, the process also helps validate that the retained test actions do preserve their original intention.

Figure 3a shows the distribution of improvement on test action intention preservation achieved by METER across apps, where a bar at x-axis $[a, b)$ with height c indicates that the ratio of the NAP after repairing to that before repairing is greater than or equal to a and smaller than b for c apps. On average, the number of test actions preserved after repairing is 1.40 (median) and 3.28 (mean) times of that before repairing. From the figure, we can see that repaired test scripts preserve more test actions than the original ones for all but two apps S5 and S12. In both cases, METER mistakenly decided to repair a test action that needed no repairing, causing the following test actions to be lost.

Figure 3b shows how the improvement on test action intention preservation relates to the impact of the GUI changes on tests, where the x-axis denotes the test action intention preservation rate of the original test scripts on the updated version app ($\mathrm{NAP}_{\mathrm{NULL}}/\#A$), and the y-axis represents the ratio of the NAP after repairing to that before repairing ($\mathrm{NAP}_{\mathrm{METER}}/\mathrm{NAP}_{\mathrm{NULL}}$). The figure clearly shows that the magnitude of improvement in test action intention preservation brings about by METER is reversely related to the percentage of test actions preserved by the test scripts before repairing: If only a few actions of the test scripts are preserved before repairing, METER can really make a difference in preserving more test actions; If a large number of test actions are already preserved before repairing, applying METER may not help much in increasing that number.

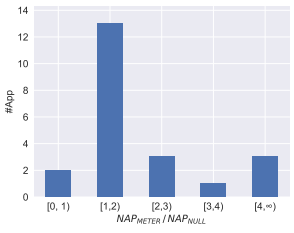
Where METER was ineffective. In the end, 26 test scripts in total (22 reported by METER and 4 discovered through manual check) out of the 104 affected test scripts were only partially repaired.

We examined those test scripts and identified four reasons for the ineffectiveness of METER. First, *major changes to the GUIs* contributed to the incomplete repair of 11 test scripts. With such major changes, both identifying screens that would match the destination screen of an affected transition and constructing test action sequences to accomplish the transition become much more challenging. Although

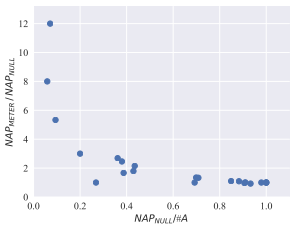
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007664, IEEE Transactions on Software Engineering

TABLE 5: Experimental results of METER on the 22 Android apps.

ID	#K	#A	SC	METER					NULL				T	T_O	T_C	T_R	T_L	T_G
				NAE	NAP	NAR	SC	#K_R	NAE	NAP	SC	#K_B						
S1	21	97	58.1%	97	97	0(0/0)	56.8%	0	97	97	57.0%	0	50.8	9.2	41.6	0.0	0.0	0.0
S2	8	83	50.1%	83	78	5(5/0)	49.2%	2	63	58	48.6%	2	37.3	11.0	23.7	2.7	2.7	0.0
S3	29	811	53.3%	753	747	6(5/0)	50.2%	2	738	738	49.4%	5	502.3	125.3	370.0	7.0	7.0	0.0
S4	31	239	53.4%	239	230	9(9/0)	57.0%	6	211	211	54.0%	6	102.1	22.2	73.9	5.9	5.9	0.0
S5	17	163	36.4%	150	141	9(9/0)	36.0%	9	163	152	35.7%	10	133.2	34.6	89.4	9.2	9.2	0.0
S6	14	106	65.1%	101	99	2(2/0)	61.6%	1	92	90	60.4%	3	125.6	18.2	50.6	56.8	44.2	12.6
S7	32	659	58.2%	659	659	0(0/0)	56.4%	0	659	659	56.4%	0	348.9	32.1	316.8	0.0	0.0	0.0
S8	6	17	67.2%	17	8	9(9/0)	68.0%	4	1	1	35.2%	4	35.4	2.5	29.2	3.7	3.7	0.0
S9	39	672	56.8%	656	633	23(23/0)	54.9%	9	493	476	55.0%	11	413.9	75.9	320.9	17.1	17.1	0.0
S10	29	195	47.4%	195	195	0(0/0)	52.2%	0	195	195	53.1%	0	69.1	20.9	48.2	0.0	0.0	0.0
S11	23	345	49.2%	345	322	23(22/1)	46.6%	9	143	131	38.0%	9	222.3	46.0	100.7	75.6	11.3	64.3
S12	10	64	71.5%	63	56	7(6/1)	73.9%	5	61	58	73.9%	6	24.4	7.3	11.9	5.3	2.2	3.1
S13	29	212	47.4%	191	163	23(17/5)	42.4%	13	110	91	28.2%	17	213.3	35.9	64.7	112.7	12.0	100.7
S14	17	159	50.8%	138	80	58(53/0)	50.1%	11	31	15	47.0%	13	132.1	32.9	69.2	30.1	30.1	0.0
S15	5	95	50.6%	95	93	2(2/0)	50.7%	1	95	93	50.0%	1	44.7	10.8	32.1	1.8	1.8	0.0
S16	9	42	28.8%	42	36	6(6/0)	28.6%	5	4	3	6.2%	5	17.1	4.6	11.1	1.3	1.3	0.0
S17	19	82	49.1%	22	22	0(0/0)	34.0%	0	24	22	34.0%	1	32.4	9.6	20.8	2.0	2.0	0.0
S18	4	13	67.8%	9	9	0(0/0)	62.6%	0	10	9	60.2%	2	11.8	2.3	5.3	4.2	0.4	3.8
S19	5	31	59.2%	27	20	7(3/4)	54.1%	2	14	12	53.2%	3	157.3	4.9	16.8	135.6	2.0	133.6
S20	5	35	67.6%	31	21	7(6/1)	60.1%	1	7	7	49.4%	3	88.4	10.8	17.2	60.4	4.7	55.7
S21	22	202	48.3%	202	196	6(6/0)	47.8%	1	74	73	40.6%	1	74.2	17.7	54.5	2.0	2.0	0.0
S22	10	46	64.8%	44	43	1(1/0)	68.4%	1	20	20	34.8%	2	34.1	5.7	22.6	5.7	0.3	5.4
Overall	384	4368	54.6%	4159	3948	203(186/17)	52.8%	82	3305	3211	46.4%	104	2843.7	532.2	1773.9	537.5	158.3	379.2



(a) Distribution



(b) Relation with updates to apps.

Figure 3: Transition preservation by repaired test scripts.

a more thorough search may be able to find the elements needed to build the repair test scripts in such cases, that is not currently supported by METER as it restricts its search and considers replacement test action sequences in a few limited forms for performance reasons. Second, *functionality deletion* leads to incomplete repair of 8 scripts. Compared with the first reason, functionality deletion renders the preservation of test actions more challenging in some cases and impossible in others. Third, manual check by the post-graduate students discovered that *screenshots as weak oracle* led to 6 test actions from 4 scripts being incorrectly repaired: 1 for app S3, 1 for app S13, 2 for app S14, and 2 for app S20. METER compares destination screens against the recorded intentions to decide whether a repair is correct, and it may get confused and choose the wrong repair when multiple test actions lead to similar screenshots. Fourth, *technical limitations of the CV libraries used* prevented 3 scripts from being completely repaired. For example, MSVC failed to recognize a list of numbers on a screen of app S19, where the size of the numbers is much smaller than that of other characters on the same screen. To alleviate this problem, we plan to experiment with and then integrate other more sophisticated CV and OCR libraries.

In total, METER *repaired 203 test actions and helped preserve 737 more test actions from the input test scripts, which amount to 63.7% of all the test actions broken by the GUI changes.*

4.4.2 RQ2: Efficiency

Table 5 also lists for each app the total repairing time (T), the time required for recording the intention of the test actions (T_O), for checking whether each test action preserves its intention (T_C), and for constructing the repairs (T_R), and the breakdown of T_R into the time for repair construction through local (T_L) and global (T_G) search.

The total time cost for test script repair on all the apps amounted to 2843.7 minutes, with the maximum repairing time being 502.3 minutes and the minimum being 11.8 minutes. App S3 took the longest time to repair, mainly because it is the largest app among the subjects and its 29 test scripts contained 811 test actions. On app S3, METER spent 7.0 minutes on constructing repairs for the tests, while the remaining 495.3 minutes were spent on recording and checking the intention of the test actions before and after repairing. Compared with that, repairing the tests for app S19 was a more challenging task, as it required 133.6 minutes, or 85% of its total repairing time, for repair construction using elements returned from global search.

Figure 4 shows the repairing time for each subject app, where the x-axis represents different apps sorted in decreasing order of their total repairing time, the top half of the figure shows the distribution of the time over various steps in repairing, and the bottom half of the figure shows the average time cost for repairing one test action of that app. Three apps S8, S19, and S20 required on average more than two minutes to repair a test action for different reasons: For

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007664, IEEE Transactions on Software Engineering

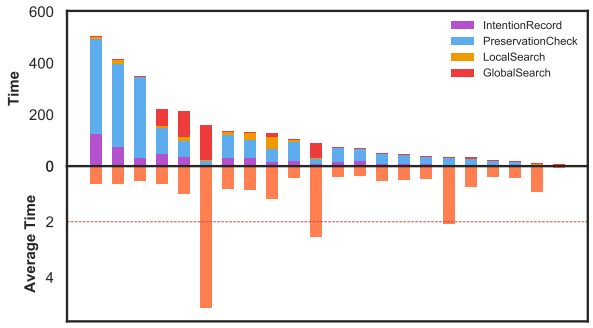


Figure 4: Time cost of individual steps of repairing in minutes.

app S8, a list of local candidate elements were found when repairing a test action and the correct element happened to be ranked towards the end of the list; Apps S19 and S20 required relatively long time because repair construction via the more expensive global search was needed. For all the other apps, the time cost for repairing a test action is less than one minute.

While the time cost of test script repair with METER does not suggest the tool to be applied in an interactive manner, we believe the overall efficiency of METER is acceptable for two reasons. First, the application of METER is fully automatic and requires no human intervention. Second, the total time cost here also includes the time spent on *validating* that 3948 test actions actually do not need repairing and can be reused directly.

We see a few opportunities to make METER more efficient. For example, the original intention of the test actions on the base version app could be gathered offline before a repair session, results from CV-based analyses could be re-used for screenshots if their associated screens are not changed, and test scripts could be repaired on multiple machines in parallel. We leave the implementation of these optimizations for future work.

On average, it took METER *14.0 (=2843.7/203) minutes to repair a test action and 0.7 (=2843.7/3948) minutes to preserve a test action.*

4.4.3 RQ3: Robustness of METER

Table 6 lists, for each parameter value, the overall effectiveness of METER on all subject apps in terms of metrics defined in Section 4.2. The default value for each parameter is marked with an asterisk (*).

With greater values for V_{tm}, METER was slightly less effective in terms of almost every metric listed. This is understandable: When two textual GUI elements need to have more words in common for them to be considered as matching, it becomes more likely for test actions to be classified as not intention-preserving and less likely for METER to successfully construct the proper test action sequences as replacements. With smaller values for V_{tm}, METER's effectiveness, however, did not change much, suggesting most matching textual GUI elements do share a significant amount of common words. Similar trend was also observed with V_{cm1}.

TABLE 6: How parameters in GUI matching affect METER's effectiveness.

PARAMETER	VALUE	NAE	NAP	NAR	SC	$\#K_R$
V_{tm}	0.3	4159	3948	203	52.7%	82
	0.4*	4159	3948	203	52.7%	82
	0.5	3989	3784	197	52.1%	77
	0.6	3989	3784	197	52.1%	77
V_{gm}	20%	4159	3948	203	52.7%	82
	30%	4159	3948	203	52.7%	82
	40%*	4159	3948	203	52.7%	82
	50%	4159	3948	203	52.7%	82
	60%	4158	3947	203	52.7%	82
V_{cm1}	0.5	4159	3948	203	52.7%	82
	0.6	4159	3948	203	52.7%	82
	0.7*	4159	3948	203	52.7%	82
	0.8	4131	3920	203	52.7%	82
	0.9	3881	3677	196	51.9%	76
V_{cm2}	0.2	4159	3948	203	52.7%	82
	0.3	4159	3948	203	52.7%	82
	0.4*	4159	3948	203	52.7%	82
	0.5	4159	3948	203	52.7%	82
	0.6	4158	3947	203	52.7%	82

Different values for V_{gm} had very limited influence on the repairing results, which suggests that most graphical GUI elements in mobile apps either share many feature descriptors and match or share very few feature descriptors and do not match. The effectiveness of METER was also rather stable with respect to V_{cm2}, and a possible reason is that few collections of GUI elements were considered matching because they had a balanced amount of textual and graphical elements that match—in consequence, the corresponding case may be less important in deciding matching GUI element collections.

Overall, such results align well with our observation that different GUI elements in mobile apps tent to be distinct from each other and share few words or feature descriptors.

METER*'s effectiveness remained quite stable when there were small changes to the parameter values used in GUI matching.*

4.4.4 RQ4: Comparison with CHATEM

Table 7 shows, for each of the 9 subject Android apps, the effectiveness metrics and the total repairing time in seconds (T) achieved by CHATEM using ESMs with (ESM+ADDITION) and without (ESM-ADDITION) manually added GUI elements, respectively. For easy reference, the table also lists the repairing results produced by METER on the same apps.

When the input ESMs do not contain manually added GUI elements, CHATEM's effectiveness was considerably worse than METER's on every app and in terms of every listed metric. Overall, CHATEM was only able to repair 2 test actions and preserve 59 test actions, resulting in repaired tests that contain only 61 executable test actions and cover merely 24.4% of the statements.

When manually added GUI elements are included in the input ESMs, CHATEM's effectiveness improved significantly: It became considerably better on every app and in terms of every metric than in the previous case and comparable with METER's. Overall, CHATEM was able to repair

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007664, IEEE Transactions on Software Engineering

TABLE 7: Comparison between METER and CHATEM.

ID	#K	#A	SC	#K_A	METER						CHATEM ESM-ADDITION						CHATEM ESM+ADDITION					
					NAE	NAP	NAR	SC	#K_R	T(m)	NAE	NAP	NAR	SC	#K_R	T(s)	NAE	NAP	NAR	SC	#K_R	T(s)
s2	8	83	50.1%	2	83	78	5	49.2%	2	37.3	4	4	0	19.9%	0	1.2	73	73	0	37.9%	0	0.9
s5	17	163	36.4%	10	150	141	9	36.0%	9	133.2	3	3	0	15.2%	0	0.4	161	152	9	33.2%	8	1.1
s8	6	17	67.2%	4	17	8	9	68.0%	4	35.4	0	0	0	30.2%	0	0.3	17	8	9	68.0%	0	0.2
s12	10	64	71.5%	6	63	56	7	73.9%	5	24.4	40	38	2	49.9%	3	0.7	41	39	2	51.8%	3	0.9
s16	9	42	28.8%	5	42	36	6	28.6%	5	17.1	1	1	0	5.3%	0	0.4	36	30	6	28.6%	5	0.3
s17	19	82	49.1%	2	22	22	0	34.0%	0	5.3	1	1	0	5.3%	0	1.1	82	78	4	45.6%	2	0.8
s18	4	13	67.8%	2	9	9	0	62.6%	0	11.8	3	3	0	32.9%	0	0.3	7	7	0	37.1%	0	0.3
s20	5	35	67.6%	3	31	21	7	60.1%	1	88.4	2	2	0	23.7%	0	0.3	30	24	6	61.0%	2	0.4
s22	10	46	64.8%	2	44	43	1	68.4%	1	34.1	7	7	0	37.3%	0	0.4	42	42	0	65.8%	1	0.4
Overall	88	545	55.9%	36	461	414	44	53.4%	27	387.1	61	59	2	24.4%	3	5.1	489	453	36	47.7%	21	5.3

TABLE 8: Experimental results of METER on iOS apps.

ID	APP	V_B	V_U	#K	#A	METER				NULL			T	T_O	T_C	T_R	T_L	T_G
						NAE	NAP	NAR	#K_R	NAE	NAP	#K_B						
s23	Camera Ultimate	1	1.2	5	61	61	61	0(0/0)	0	61	61	0	48.9	7.2	41.7	0.0	0.0	0.0
s24	KeePassTouch	1.4.1	1.5	15	228	228	226	2(2/0)	2	212	210	2	254.1	27.5	224.3	2.3	2.3	0.0
s25	Pass2word	1.3.1	1.6.0	9	104	104	103	1(1/0)	1	104	102	1	115.6	13.0	101.5	1.0	1.0	0.0
s26	Pedometer	1.1.1	1.1.9	12	145	73	61	12(12/0)	6	55	35	11	123.8	19.8	78.7	25.3	25.3	0.0
s27	SMS Scheduler	2	7	2	17	19	14	5(5/0)	1	0	0	2	16.8	3.0	9.6	4.2	4.2	0.0
s28	Sound Board	2.1	2.5	1	20	20	20	0(0/0)	0	20	20	0	24.9	3.3	21.6	0.0	0.0	0.0
Overall				44	575	505	485	20(20/0)	10	452	428	16	584.1	73.8	477.4	32.9	32.9	0.0

36 test actions and preserve 453 test actions, producing repaired tests that contain 489 executable test actions and cover 47.7% of the statements. Particularly, the number of test actions preserved by CHATEM is slightly larger than that of METER's and a closer look at the repairing results on each app reveals that, CHATEM only preserved more test actions than METER on 2 of the 9 apps, namely apps s5 and s17, but it preserved much more test actions than METER on app s17 (78 vs. 22), which helped CHATEM achieve a better overall NAP value.

Despite the differences in the completeness of the input ESMs, test script repair with CHATEM almost always completed within 2 seconds, since CHATEM does not need to actually execute the tests. In comparison, it took METER much longer to repair the same sets of tests. However, if we take into account also the dozens of hours spent on constructing the input ESMs for CHATEM (Section 4.1.2), the total amounts of time required by the two tools for repairing the test scripts are of similar order of magnitude.

We make three observations about such results. First, METER is able to produce repairing results comparable with that of CHATEM, even when high quality input ESMs are provided. Second, models extracted by Gator need to be complemented with missing GUI elements so that the resultant ESMs can drive CHATEM to effectively repair test scripts. Third, given that METER runs in a fully automatic way, while a significant amount of manual effort is needed to prepare the high quality input ESMs for CHATEM, METER strikes a better balance between effectiveness and efficiency overall.

> METER *runs automatically and can achieve repairing results that are comparable with what* CHATEM *produces when provided with high quality input ESMs that require considerable manual effort to prepare.*

4.4.5 RQ5: Applicability of METER on iOS apps

Table 8 lists, for each of the six iOS apps, the metrics of the repairing results produced by METER and the null repair tool.

Before being repaired by METER, 123 (=575-452) test actions were not executable on the updated version apps, breaking 16 of the 44 test scripts. Among the 452 executable test actions, 428 preserved their intention on the updated version apps. Meanwhile, test scripts of two apps s23 and s28 were not affected by the GUI changes at all.

METER repaired 20 test actions, all via local search for the candidate GUI elements, and 10 test scripts completely, raising the number of executable test actions to 505. More importantly, 57 (=485-428) more test actions are preserved after the repairing, which amounts to 38.8% (=57/147) of all the test actions broken by the GUI changes. The manual check of the repairing results confirmed that all the repairs were correct, i.e., they all have the intended semantics.

The experiments took in total 584.1 minutes to finish. On average, it took METER 29.2 (=584.1/20) minutes to repair a test action, and 1.2 (=584.1/485) minutes to preserve a test action.

> *In total,* METER *repaired 20 test actions and helped preserve 57 more test actions from the input test scripts, which amount to 38.8% of all the test actions broken by the GUI changes. On average,* METER *spent 29.2 and 1.2 minutes to repair and preserve one test action, respectively.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2020.3007664, IEEE Transactions on Software Engineering

4.5 Threats to Validity

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

4.5.1 Construct validity

Threats to construct validity are mainly concerned with whether the measurements used in the experiment reflect real-world situations.

In this work, we measure the effectiveness of test script repair in terms of to what extent the repairing process preserves the semantics of the test actions, and we consider the semantics of a test action to be preserved if the transition it produces is achieved in the repaired test scripts. Given that realizing a test action's transition does not always mean the semantics of the test action is preserved, the oracle METER employs here is weak and may lead to incorrect repairing results. To mitigate the threats introduced, we asked two postgraduate students to manually examine whether the test actions really have their intended semantics after repairing. The repair for 4 test actions turned out to be incorrect. In the future, we will add the support for stronger oracles, e.g., in the form of assertions, into METER.

4.5.2 Internal validity

Threats to internal validity are mainly concerned with the uncontrolled factors that may have also contributed to the experimental results.

In our experiments, the main 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.

Another threat to internal validity has to do with the amount of manual effort invested in preparing the input ESMs for CHATEM. To mitigate that, we conducted two experiments with CHATEM on each Android app, using input ESMs of different completeness levels, and compared both results with that produced by METER.

4.5.3 External validity

Threats to external validity are mainly concerned with whether the findings in our experiment are generalizable to other situations.

One major threat to external validity concerns the subject apps used in the experiments. To collect a diversified set of Android apps, we resort to six papers on mobile applications from top software engineering conferences and selected in total 22 apps as the subjects. Most apps selected in this way, however, turned out to be open source in nature, which poses a major threat to the external validity of our findings, as these apps may not be good representatives of the other Android apps. The relatively small number of iOS apps used in the experiments poses a similar threat. While we do not see any restrictions that would prevent METER from being as successful on many other apps, we plan to conduct more extensive experiments to further evaluate the effectiveness and efficiency of METER in the future.

Another threat to external validity lies in the test scripts that METER repaired in the experiments. To make sure the preparation of test scripts is not biased in favor of METER, we asked postgraduate students with experience in mobile development to craft the test scripts. The scripts written by other test engineers or generated by automatic tools, however, may have different characteristics that influence the behavior of METER. For future work, we plan to invite our industry partners to use METER to repair test scripts they write for their production apps. Results from such real-world applications will help us understand better the strengths and weaknesses of METER.

4.6 Limitations

We summarize in this section the main limitations of METER that we identified. First, METER expects that the executions of the test scripts to repair are deterministic and time-insensitive, and it also assumes that each app under consideration would occupy the whole screen while running. Apps and test scripts not satisfying such properties are not suitable inputs for METER, and therefore 12 apps from the subject pool were excluded from the experiments (see Section 4.1.1). Second, METER undoes possible changes caused by the attempted but undesirable test actions during repair construction by terminating the current testing process and executing the constructed partial test scripts from the beginning (Line 23 in Algorithm 2). While such design was able to eliminate the effects caused by those undesirable test actions most of the time in the experiments (partly because many screens and test actions are not directly affected by those effects, and partly because, since the matching relation in METER does not demand 100% equivalence, many of those GUI differences were too small to affect repair results), it may not be enough to completely restore the app states in some cases. Take app OI File Manager in Section 2 as an example. If an attempted test action deletes a file, unless the test contains an action to add the file back, the app GUI most likely would show some difference after the backtrack. If the difference is big or it directly influences the execution of the following test actions, the remaining repair process would be affected and the repair results might be incorrect. We see two possible ways to address this limitation. On the one hand, we may increase METER's awareness of app states so that the repair process handles the differences better. On the other hand, we may also stipulate that every test always starts with resetting the relevant parts of the testing environment, so that all tests would be truly independent on each other and on test actions attempted in repair construction. Third, GUI matching in METER is based on a group of heuristics and the current values of the parameters were determined based on our personal experiences with mobile apps. Major factors influencing such experiences include, e.g., screen size and resolution of popular mobile devices and common font size used in most apps. Therefore, incorrect repairs become virtually inevitable. As we have seen in Section 4.4.1, test scripts for two apps preserved fewer test actions after being repaired by METER. While the numbers of these cases and the affected test actions were both small in our experiments, such outliers do increase the costs for using METER and the quality of repair results may vary greatly on specific devices or apps. To address this limitation, on the one hand, we may utilize more static and dynamic information about the environment and apps

so as to prune out as many incorrect matching relations as possible; On the other hand, we may also provide ways for users to guide test repair with ease. For example, an additional process could be installed to allow users to, manually or interactively, tailor the parameter values for the environments and apps; A tool may also be developed to visually display how GUI elements are matched between versions and how repairs are constructed based on the matching relations, so that it becomes more convenient for users to spot problems in the repair process.

5 Related Work

In this section, we review works closely related to METER, which fall into three categories: general purpose test repair, GUI test repair, and computer vision in software engineering.

Note that there is a clear distinction between test repair and another popular research area automated program repair: The former aims to *modify tests* so that they run successfully on programs, while the latter focuses on *changing programs* to make existing tests pass [44]–[49].

5.1 General Purpose Test Repair

Changes made to a software system during its evolution may render some existing tests for the system broken. That is, those tests will fail on the evolved system not because the system is buggy, but because the tests do not reflect the changes. To reduce the burden of updating those broken tests for programmers, various techniques have been developed in the past years. Deursen et al. [50] propose techniques to fix compilation errors in tests caused by refactorings to the program code. Daniel et al. [51] propose the REASSERT technique to automatically repair broken unit tests. REASSERT monitors the execution of a unit test on a presumably correct program and uses the information gathered during the execution to update the literal values, assertion methods, or assertions in the test. To overcome some of REASSERT's limitations, Daniel et al. [52] 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 path conditions and assertions gathered during the execution are then solved by the Z3 constraint solver [53] and the solutions are used to replace the literals. Yang et al. [54] propose the SPECTR technique that repairs tests based on changes to program specifications rather than implementations.

5.2 GUI Test Repair

Compared with general purpose test repair, the problem of GUI test repair has attracted more attention from researchers, partly because it is common for developers to create GUI test scripts using record-and-replay testing tools, while those test scripts are more fragile.

Memon and Soffa [13] first propose the idea of GUI test script repair and develop a model-based approach called GUI Ripper targeting desktop applications. 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, Memon [10] extends GUI Ripper by adding a mechanism to obtain the application model through reverse engineering. In view that the model built by GUI Ripper is just an approximation of the actual application and may cause incorrect repairs, Huang et al. [55] propose to use a genetic algorithm to generate new, feasible test cases as repairs to GUI test suites.

Besides model-based approaches, white box approaches have also been studied for GUI test script repair. Daniel et al. [15] propose to record GUI code refactorings as they are conducted in an IDE and use them to repair test scripts. Grechanik et al. [14] 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. Fu et al. [56] develop a type-inference technique for GUI test scripts based on static analysis, which can assist testers to locate type errors in GUI test scripts.

Dynamic and static analyses have also been combined in test script repair for desktop applications. To repair changed GUI workflows, Zhang et al. [21] 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. [7] recognize 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 extracted models and repairs test scripts for desktop applications.

Research on GUI testing for web applications has gained better results, because web applications tend to have less complex GUIs than desktop applications, and because the DOM trees of a web application's web pages can be easily retrieved and utilized to facilitate testing related activities. Raina and Agarwal [16] propose to reduce the cost of regression testing for web applications by executing only the tests that cover the modified parts of the applications. 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. [17] propose the WATER technique to repair GUI test scripts for a web application so that the scripts can run successfully on the successive version of the same application. WATER gathers the DOM trees of the web pages after the execution of each test action on both versions of the application, and it suggests repairs based on the differences between the properties of the DOM nodes. Harmen and Alshahwan [57] develop a technique to repair user session data, instead of test scripts, of web applications. The technique involves a white-box analysis of the structure of the web application. Stocco et al. [18] propose the VISTA technique to repair test scripts for web applications. VISTA captures the visual information associated with each test action and augments the DOMs of a web application with visual information about the GUI elements. Such visual information is then utilized to help decide whether a test action executes as expected and which other GUI element a repair test action should interact with.

Compared with such works, METER mimics the manual test script repairing process of human testers [58] and requires no information about the source code or structure of the application under consideration at all, which makes the approach particularly valuable for mobile app test script

repair, since most mobile apps are closed-source and testers have to treat the apps as black-box systems in testing. To effectively construct new test action sequences as ingredients for the repaired test scripts, METER builds a behavioral model for the app based on the successful execution of test scripts on the base version app and the similarity relation between the app's screens.

Studies targeting the mobile domain are just emerging and quite limited. In two recent studies [34], [35], model-based approaches for Android GUI test repair are proposed. The approaches require a precise model of the app under consideration to guide the test script repair, assuming that the model is either readily available or obtainable at a relatively low cost. The assumption, however, may not hold on large apps like AnyMemo and K-9 Mail used in our experiments. In contrast, METER treats each mobile app as a black-box system and utilizes computer vision techniques to detect undesirable test action executions and construct new actions as repairs by analyzing the snapshots of app screens.

5.3 Computer Vision in Software Engineering

Yeh et al. [59] propose the Sikuli technique targeting desktop applications that allows users to take the screenshot of a GUI element and query a help system using the screenshot instead of the element's name. When used to help crafting GUI test scripts, the technique can significantly reduce the sensitivity of test scripts to GUI changes like element reposition or rotation [60]. Alégroth et al. [61] develop the JAutomate tool that combines image recognition with test script record and replay to reduce the automation cost of visual GUI testing [62]. Chang et al. [63] apply computer vision techniques to enable conventional software testing to observe the states and events in physical world. Nguyen et al. [64] develop the REMAUI approach where computer vision and optical character recognition techniques are applied to reverse engineer mobile application user interfaces. Chen et al. [65] develop the UI X-RAY system that integrates computer vision methods to detect and correct inconsistencies between UI design and implementation.

Different from all these works, METER employs computer vision techniques to identify elements on app GUIs and to establish the matching relation between those elements based on their similarities. The elements and the matching between them are essential for the effective detection of test failures and construction of repair test actions.

6 Conclusion

In this paper, we propose METER—a novel approach to automatically repairing GUI test scripts for mobile apps based on computer vision techniques. Experimental evaluation of METER on 28 real-world mobile apps from both the Android and iOS platforms show that METER is both effective and efficient.

Acknowledgments

The authors thank the anonymous reviewers for their valuable comments and suggestions for improving this article. This work is supported by the National Natural Science Foundation (Nos. 61690204, 61972193, and 61632015) and the Fundamental Research Funds for the Central Universities (Nos. 14380022 and 14380020) of China. This work is also supported in part by the Hong Kong RGC General Research Fund (GRF) PolyU 152703/16E and PolyU 152002/18E and The Hong Kong Polytechnic University internal fund 1-ZVJ1 and G-YBXU.

References

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.

[2] M. Nayebi, B. Adams, and G. Ruhe, "Release practices for mobile apps – what do users and developers think?" in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 552–562.

[3] K. Moran, M. Linares-Vásquez, and D. Poshyvanyk, "Automated gui testing of android apps: From research to practice," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 505–506.

[4] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: ACM, 2011, pp. 77–83.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 4:1–4:36, 2008.

[11] M. Grechanik, Q. Xie, and C. Fu, "Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts," in *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*. IEEE Computer Society, 2009, pp. 9–18.

[12] A. Cravens, "A demographic and business model analysis of today's app developer, 2012," accessed in Aug, 2017.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 503–514.

[19] L. Yu, W. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, and W. Zhao, "Testing as a service over cloud," in *The Fifth IEEE International Symposium on Service-Oriented System Engineering, SOSE 2010, June 4-5, 2010, Nanjing, China*, 2010, pp. 181–188.

[20] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for gui-based software applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, Feb. 2007.

[21] 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.

[22] 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.

[23] "TestWorks CAPBAK," http://www.testworks.com/Products/Regression.msw/capbakmsw.html, 2018, [Online; accessed 20-March-2018].

[24] X. Li, M. d'Amorim, and A. Orso, "Intent-preserving test repair," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, 2019, pp. 217–227.

[25] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li, "Gui-guided repair of mobile test scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, May 2019, pp. 326–327.

[26] J. F. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 6, pp. 679–698, 1986.

[27] G. R. Bradski and A. Kaehler, *Learning OpenCV - computer vision with the OpenCV library: software that sees*. O'Reilly, 2008.

[28] R. Smith, "An overview of the tesseract ocr engine," in *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, vol. 2. IEEE, 2007, pp. 629–633.

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

[30] 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.

[31] "Python language bindings for Appium," https://github.com/appium/python-client, 2018, [Online; accessed 20-March-2018].

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

[33] "Microsoft Cognitive Services," https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/, 2018, [Online; accessed 20-March-2018].

[34] 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.

[35] 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.

[36] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 658–668.

[37] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 429–440.

[38] R. J. Behrouz, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 425–436.

[39] H. Zhang, H. Wu, and A. Rountev, "Automated test generation for detection of leaks in android applications," in *Proceedings of the 11th International Workshop on Automation of Software Test, AST@ICSE 2016, Austin, Texas, USA, May 14-15, 2016*, 2016, pp. 64–70.

[40] D. D. Perez and W. Le, "Generating predicate callback summaries for the android framework," in *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, 2017, pp. 68–78.

[41] W. Song, X. Qian, and J. Huang, "Ehbdroid: beyond GUI testing for android applications," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 27–37.

[42] A. Sadeghi, R. Jabbarvand, and S. Malek, "Patdroid: permission-aware GUI testing of android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 220–232.

[43] P. Runeson, "Using students as experiment subjects - an analysis on graduate and freshmen student data," *Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering*, pp. 95–102, 01 2003.

[44] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the IEEE 31st International Conference on Software Engineering*, 2009, pp. 364–374.

[45] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated Fixing of Programs with Contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[46] L. Chen, Y. Pei, and C. A. Furia, "Contract-based Program Repair Without the Contracts," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 637–647.

[47] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE âĂŹ18. New York, NY, USA: Association for Computing Machinery, 2018, p. 187âĂŞ198.

[48] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts: An extended study," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[49] M. Monperrus, "Automatic software repair: a bibliography," University of Lille, Tech. Rep. hal-01206501, 2015.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] M. Harman and N. Alshahwan, "Automated session data repair for web application regression testing," in *First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008*. IEEE Computer Society, 2008, pp. 298–307.

[58] M. Mirzaaghaei, "Automatic test suite evolution," in *SIGSOFT-/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 396–399.

[59] T. Yeh, T. Chang, and R. C. Miller, "Sikuli: using GUI screenshots for search and automation," in *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology, Victoria, BC,*

Canada, October 4-7, 2009, A. D. Wilson and F. Guimbretière, Eds. ACM, 2009, pp. 183–192.

[60] T. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI 2010, Atlanta, Georgia, USA, April 10-15, 2010*, 2010, pp. 1535–1544.

[61] E. Alégroth, M. Nass, and H. H. Olsson, "Jautomate: A tool for system- and acceptance-test automation," in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 2013, pp. 439–446.

[62] E. Börjesson and R. Feldt, "Automated system testing using visual GUI testing tools: A comparative study in industry," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, 2012, pp. 350–359.

[63] R. Ramler and T. Ziebermayr, "What you see is what you test - augmenting software testing with computer vision," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 2017, pp. 398–400.

[64] T. A. Nguyen and C. Csallner, "Reverse engineering mobile application user interfaces with REMAUI (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 248–259.

[65] C. R. Chen, M. Pistoia, C. Shi, P. Girolami, J. W. Ligman, and Y. Wang, "UI x-ray: Interactive mobile UI testing based on computer vision," in *Proceedings of the 22nd International Conference on Intelligent User Interfaces, IUI 2017, Limassol, Cyprus, March 13-16, 2017*, G. A. Papadopoulos, T. Kuflik, F. Chen, C. Duarte, and W. Fu, Eds. ACM, 2017, pp. 245–255.

[66] M. B. Cohen, L. Grunske, and M. Whalen, Eds., *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 2015.

Minxue Pan is an associate professor with the State Key Laboratory for Novel Software Technology and the Software Institute of Nanjing University. He received his B.S. and Ph.D. degrees in computer science and technology from Nanjing University. His research interests include software modelling and verification, software analysis and testing, cyber-physical systems, mobile computing, and intelligent software engineering.

Tongtong Xu holds a B.S. in Physics from Nanjing University. He is currently a Ph.D. student in Department of Computer Science and Technology, Nanjing University. His main research interests lie in automatic program repair and software testing.

Yu Pei is an assistant professor with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. His main research interests include automated program repair, software fault localization, and automated software testing.

Zhong Li holds a B.S. in Computer Science and Technology from Nanjing University of Posts and Telecommunications. He is currently a Ph.D. student in Department of Computer Science and Technology, Nanjing University. His main research interests lie in software testing.

Tian Zhang is an associate professor with the Nanjing University. He received his Ph.D. degree in Nanjing University. His research interests include model driven aspects of software engineering, with the aim of facilitating the rapid and reliable development and maintenance of both large and small software systems.

Xuandong Li received the B.S., M.S. and Ph.D. degrees from Nanjing University, China, in 1985, 1991 and 1994, respectively. He is a full professor in Department of Computer Science and Technology, Nanjing University. His research interests include formal support for design and analysis of reactive, distributed, real-time, hybrid, and cyber-physical systems, and software testing and verification.