Change-Based Test Script Maintenance for Android Apps

Nana Chang*, Linzhang Wang*, Yu Pei‡, Xuandong Li*
*State Key Laboratory for Novel Software Technology, Nanjing University, China
‡Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China
chang1533002@gmail.com, lzwang@nju.edu.cn, csypei@comp.polyu.edu.hk, lxd@nju.edu.cn

***Abstract*—In regression GUI testing for Android apps, test scripts often fail due to outdated test actions. To avoid such false positives while still retaining the value of the old test scripts as much as possible, programmers need an automatic way to maintain the tests after the corresponding GUI has evolved.**

In this paper, we propose the CHATEM approach to automate GUI test script maintenance for Android apps. Taking as input the models for the GUIs of the base and updated version app and the original test scripts, CHATEM automatically extracts the changes between the two GUIs and generates maintenance actions for affected test scripts. In an experimental evaluation on 16 Android apps, CHATEM was able to automatically maintain the test scripts so that overall more than 95% of the remaining behaviors tested before are still tested, and almost 80% of the reusable test actions are retained in the result tests.

***Index Terms*—GUI testing, Regression testing, Automated test maintenance**

I. INTRODUCTION

In the past few years, mobile devices have become an indispensable part of people's daily life and, along with them, tens of thousands of mobile applications (or apps) have been programmed and put on the market, creating fierce competition among the apps with overlapping functionalities. As users are offered more options now and it is very easy for them to switch from one app to another, developers have to pay more attention to product quality in order to attract the users and win the competition. Mobile apps are typically developed in short iterations, and automated regression testing is often needed at the end of an iteration to efficiently check that the introduced changes do not break existing functionality. While most mobile apps have rich graphical user interfaces (GUIs) and ensuring the correctness of an app at the GUI level is critical for the success of the app, automated regression GUI testing of those apps has been a challenging task.

In GUI testing for mobile applications, tests are often written as scripts for triggering events on the GUI widgets. Since most such test scripts locate widgets with respect to a context window and through querying certain properties of the widgets like IDs, texts, and indexes within a list, they are sensitive to changes that affect those relevant properties of widgets. For example, if a test script accesses a button through its text, assigning a different text to the button may then cause the test script to fail. Such fails in regression GUI testing, however, are not useful in the sense that they do not reveal faults in the application under testing: they occur only because the test scripts become invalid after the changes. To avoid such failures during regression testing while reusing as many as possible existing test scripts, considerable research has been conducted to automatically bring test scripts in sync with the GUI they exercise.

Although there are a number of ways to automatically generate GUI test scripts [1]–[3], completely throwing away old test scripts and generating new ones from scratch is not always affordable or desirable. First, generating new test scripts with acceptable quality, e.g., measured in state coverage, can be enormously expensive; Second, automated test script generation may fail to cover important behaviors due to the lack of domain specific expertise or limitations of the generation techniques, and manually crafted test scripts are needed to complement the generated ones. Last but not least, existing tests often incorporate human effort already, since test engineers often need to manually revise the generated scripts to improve their efficiency, understandability, or documentation.

Researchers have proposed different approaches to repairing GUI test scripts for regression testing. Grechanik et al. [4] developed the REST technique that models GUIs as trees of widgets and activities and compares the trees to identify changed GUI objects, which are then used to guide the analysis and update of test actions involving the changed GUI objects. Gao et al. [5] proposed SITAR to repair test scripts. SITAR maps test scripts to an annotated event-flow graph (EFG) model, and uses four repairing transformations to repair the unusable test script actions. For repairing test actions that are not in the model or changes that can not be repaired by the repairing transformations, human input is needed. In our previous work [6], we developed the ATOM technique for GUI test scripts maintenance. Given an event sequence model (ESM) encoding the behavior of a base version app and a delta ESM (DESM) capturing the changes to the base version, ATOM automatically repairs the test scripts targeting the base version app to run on the changed version app. Although useful, ATOM does require manual construction of the DESM, which can be challenging at times as it requires knowledge about not only the changes, but also how the base version app and the ESM are related.

In this paper we propose a CHAnge-based TEst Maintenance approach to regression GUI testing, named CHATEM. Given two ESMs for the base and updated version app and

a group of test scripts for the base version app, CHATEM automatically extracts changes between the two GUIs, identifies the impact of the changes on the tests, generates and applies maintenance actions to update the affected test scripts. As for the construction of the input ESMs, we propose a semi-automatic approach, which combines the application of state-of-the-art model extraction tools and manual confirmation to strike a balance between efficiency and model quality. Since all the maintenance actions constructed by CHATEM have direct connection with affected actions in test scripts, we hope maintenance results generated in this way are easier for programmers to understand and then more likely to be accepted by programmers.

We have implemented the CHATEM technique into a tool, also named CHATEM, and conducted an experiment to evaluate the maintenance power of CHATEM. In the experiment, we used 16 production Android apps from previous research work as the subjects, and overall CHATEM was able to produce maintained scripts that test more than 95% of the behaviors tested before and retain almost 80% of the reusable test actions. The effectiveness of CHATEM was comparable with ATOM's, suggesting CHATEM could be used as an alternative or a complement to ATOM.

The main contributions of this paper are as the following.

- We propose the CHATEM automated approach to Android GUI test script maintenance. CHATEM extracts changes between different GUIs based on their corresponding ESMs, and generates maintenance actions that can be applied directly to affected test scripts to bring them in sync with the updated version GUI.
- We implement the CHATEM technique into a prototype tool, also named CHATEM, for more efficient test script maintenance and regression testing of mobile applications.
- We conduct an empirical study on 16 real-world Android applications to evaluate the effectiveness of CHATEM. The experimental results suggest that the effectiveness of CHATEM is comparable with our previous approach ATOM, but users are relieved of the burden of manually building a model to encode the changes between different versions of the app.

The rest of this paper is organized as follows. Section 2 takes a mobile application as an example to illustrate CHATEM from a user's perspective. Section 3 introduces the preliminaries to Android GUI testing. Section 4 details the framework and the individual components of CHATEM. Section 5 explains the experiments we conducted to evaluate the effectiveness of CHATEM as well as the experimental results. Section 6 briefly reviews related work in GUI testing and test script maintenance for mobile applications. Section 7 concludes the paper and discusses a few options for possible future work.

II. PRELIMINARY

In this section, we will give an introduction to the basic concepts related to GUIs of Android apps and the scripts developers use for testing the GUIs.

A. Composition of an Android GUI

Android applications are typically developed in the Java programming language and executed on the Dalvik virtual machine. The Android SDK tool compiles the source code, the data and resources of an app into an APK file, which can then be installed directly on Android devices. Most APK files include a manifest file (named `AndroidManifest.xml`), a group of bytecode files for the compiled application classes (to be interpreted by the Dalvik virtual machine), and other necessary resource files. The manifest file contains the global configuration and is an essential part of the app. Besides specifying the file structure of an app, a manifest file also lists the major components of the app like activities, services, broadcast receivers, and content providers, and declares the app's permission to access various APIs and interact with other apps.

Most interactions between apps and their users are through activities, which can be roughly understood as the counterpart of windows in desktop applications. To introduce activities and their supported user interactions, we use general concepts like screen, widget, and event.

A **screen** is an activity, a menu or a dialog [7]. An activity is defined by subclassing the `android.app.Activity` class, and a dialog by subclassing `android.app.Dialog`. There are two kinds of menus. Option menus are related to activities, and an activity can only have at most one option menu. Option menus are typically initialized in the callback method `onCreateOptionMenu`. Context menus are associated with widgets of activities, and are initialized in method `onCreateContextMenu`. One can register a context menu for a widget by calling method `registerForContextMenu`. All the statically created activities must be declared in the app's manifest file, and each activity will be assigned a unique id to mark its uniqueness.

A screen is *active* if it can receive user inputs and interact with the user. Given any point in time, each app has at most one active screen.

During testing, the active screen provides the execution context for the next test script action. If a screen is *inactive*, the next test script action will fail if it targets a widget on that window.

Screens consist of **widget**s. Widgets have properties (e.g., `ID` and `title`) and users can trigger events on widgets. In general, Android apps deal with two types of events: widget events and system default events. Widget events are triggered through user interactions with the corresponding widgets. A widget event can be identified by the widget triggering the event and the event type, and a widget may trigger multiple types of events. For example, clicking on and long clicking on a button will trigger two different events on the same button. System default events are not associated with any widget, but are triggered by pressing the physical buttons like `BACK` or `HOME` on mobile devices. An event in general relates two screens: One is the currently active screen of the app and the

other is the screen to become active in response to the event. We refer to these two screens as the source and the target screen of the event, respectively. The source screen and the target screen of an event may be the same.

B. Android GUI Test Scripts and Test Actions

To automate the system testing of Android apps on the GUI level, developers often write test scripts, which can then be executed by various testing engines. Although different GUI testing engines support different sets of features for test script construction, most test scripts are essentially sequences of test actions and each test action consists of two important parts: a widget locator and an event trigger. When applied on an active screen, a widget locator returns the widget on the screen that satisfies certain criterion; An event trigger signals an event of specified type on a widget. From an active screen, the execution of a test action then involves first locating a target widget and then firing the specified event on that widget. The execution fails if multiple or no widget can be found or the located widget does not support the type of event to be fired. Properties that are often used to construct locators include, e.g., `ID`, `text`, `type`, and `XPath`. In the remaining of this paper, we refer to the collection of these properties as *locator properties*.

III. ChaTeM in Action

In this section, we use a simple Android App named NotePad to show how ChaTeM works in helping maintain GUI test scripts for mobile applications. The NotePad application was also used as the running example to demonstrate the work flow of the ATOM GUI test script maintenance technique [6]. We adopt the same example in this paper to make the differences between ATOM and ChaTeM more distinct and easier to understand.

NotePad is a simple app for note taking. Figure 2 shows four activities in the base version of the app for adding, editing, and deleting notes. On each of the first three activities, a user may call up a menu by pressing the Menu physical key, as shown on the bottom of the corresponding activities.

In the base version of NotePad, the initial activity shown when the application is first launched is activity SC_a, which lists all the notes previously saved in the app. From there, a user may choose to either add a new note or edit an existing note, by clicking on the "Add note" button or a note entry from the list. A new note will be opened in activity SC_b for editing, where the user may save or discard the changes made till a certain point by clicking on the button "Save" or "Discard" on the activity. An existing note, however, will be opened in activity SC_c for editing. There, the user may save the changes, revert the changes, delete the note, or edit the title of the note, by clicking on the corresponding button on the activity.

Figure 3(a) shows three test scripts written in Robot Framework[1] for testing the base version of NotePad [6]. All the three scripts start execution from the initial activity SC_a and each of them essentially contains a sequence of *action*s, one action per line. For example, the first action in TS_1 is to press the Menu physical key on activity SC_a, and the second is to click on the menu button with text "Add note". For ease of presentation, we refer to a widget simply by the text on it, when the meaning is clear from the context. Test scripts also often use texts to identify the widgets to operate on, and other properties used for widget identification in test scripts include, e.g., the IDs and the names.

TS_1 creates a new note, adds some text, and then saves the note; TS_2 is the same as TS_1, except that the changes are discarded at the end; TS_3 execute assumes the presence of a note item named note1. It opens a note named note1, presumably stored in the app before testing, by clicking on the note item, then clicks on Delete to remove the note. Although more sophisticated mechanisms may be employed to devise stronger oracles, the execution of a test script is considered successful if no error is triggered in the process. Since test scripts TS_1, TS_2, and TS_3 are written against the base version of NotePad, they all run successfully as intended.

An updated version of NotePad incorporates the following changes to the GUI of the base version. First, the Add note menu button on SC_b is moved to Add; Second, instead of deleting a note immediately after a user clicking on the Delete button on SC_c, the app opens a dialog with Yes and No buttons and only proceed with the action if Yes is clicked; Third, the Discard menu item is removed from SC_c.

With such changes to the GUI, some of the action sequences given by the original test scripts no longer describe feasible interactions with the app. That is, they become *obsolete*. Particularly, TS_1 and TS_2 will both fail on the updated GUI as SC_a no longer has a menu item Add note; while TS_3 will not fail, it does not complete the deletion action either.

Taken two models encoding behaviors of the app in two versions, ChaTeM computes automatically the changes one has to make to get the upated GUI from the base version, determines which parts of the test scripts are affected by the changes, generates corresponding maintenance actions to mitigate the impact caused by each change, and applies the generated actions to make the obsolete test scripts in sync with the updated GUI. Figure 3(b) shows the result test scripts produced by ChaTeM, with added and modified actions highlighted. TS_1 is updated to reflect the change of menu item name from Add note to Add; Besides being updated in the same way as in TS_1, TS_2 is also truncated, with infeasible events removed from the script; TS_3 is extended with the action of clicking on the Yes button on the confirmation dialog, and therefore successfully deletes note1.

Although the results are the same for GUI test script maintenance with ATOM [6] and ChaTeM, the two techniques require different input information and use different models for script maintenance.

Elaborate!

[1] http://robotframework.org/

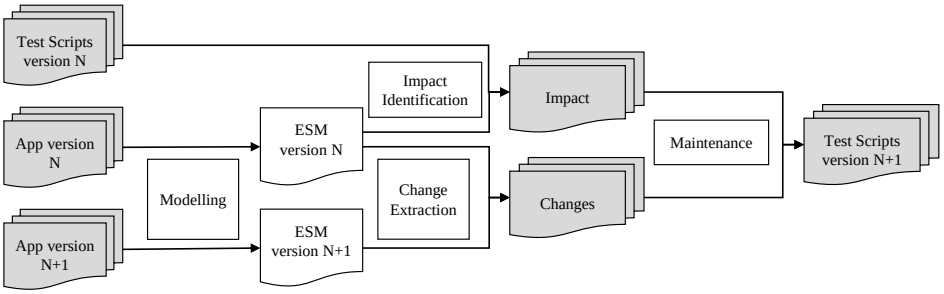
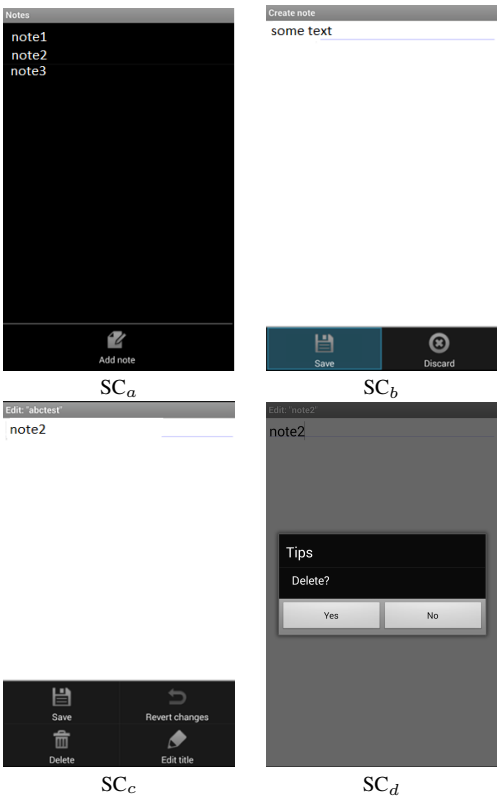


Fig. 1. The overall framework of CHATEM.



SC_a SC_b SC_c SC_d

Fig. 2. Three screens and their corresponding menu items in the NotePad app.

TS_1
1 Press Keycode MENU
2 Click Element name=Add note
3 Input Text id=some text
4 Press Keycode MENU
5 Click Element name=Save

TS_2
1 Press Keycode MENU
2 Click Element name=Add note
3 Input Text id=some text
4 Press Keycode MENU
5 Click Element name=Discard

TS_3
1 Click Element name=note1
2 Press Keycode MENU
3 Click Element name=Delete

(a) Base version

TS_1
1 Press Keycode MENU
2 Click Element name=Add
3 Input Text id=some text
4 Press Keycode MENU
5 Click Element name=Save

TS_2
1 Press Keycode MENU
2 Click Element name=Add
3 Input Text id=some text
4 Press Keycode MENU
5 Click Element name=Save

TS_3
1 Click Element name=note1
2 Press Keycode MENU
3 Click Element name=Delete
4 Click Element name=Yes

(b) Updated version

Fig. 3. Test Scripts for NotePad

IV. APPROACH

In this paper, we propose the CHATEM approach to maintaining test scripts based on event sequence models (ESMs) of evolving mobile apps. CHATEM requires as input the ESMs for the base version and the updated version of the app under consideration as well the test scripts for the base version. Taking those inputs, CHATEM identifies changes between the two versions, relates the identified changes to test actions affected by the changes, constructs alternative test actions, and replaces the obsolete test actions with the alternatives to produce the updated test scripts that are suitable to run on the updated version of app. Besides maintaining existing test scripts, CHATEM also generates new test scripts to exercise the new event handlers, widgets, and screens added to the GUI.

Figure 1 shows the overall framework of CHATEM. Four main steps are involved in applying CHATEM: constructing ESMs by combining automated model extraction and manual confirmation; extracting changes to the GUI by comparing the ESMs; identifying impacts of the changes on test scripts by simulating the tests on the base version ESM; maintaining the test scripts by constructing alternative test actions and using them to replace the obsolete test actions. The rest of this section first defines the event sequence model that we use to model the GUIs of mobile apps and then explains in detail each individual step.

A. Event Sequence Model (ESM)

We adopt the event sequence model from [6] to abstract possible behaviors of an Android app. For the sake of completeness, this section also includes the definition of ESM.

Let $\boldsymbol{W}$ be the set of widgets in an app, and $\boldsymbol{E}$ the set of event types on $\boldsymbol{W}$, an event sequence model for an app is a non-deterministic finite state machine $\mathcal{M} = \langle \Sigma, \boldsymbol{S}, \{s_0\}, \boldsymbol{C}, \boldsymbol{F} \rangle$, where

- $\Sigma = \boldsymbol{W} \times \boldsymbol{E}$ is the set of events in the app;
- $\boldsymbol{S} \subseteq 2^{\boldsymbol{W}} (s_i \cap s_j = \emptyset, \forall s_i \in \boldsymbol{S}, s_j \in \boldsymbol{S}, i \neq j)$ is the set of screens in the app;
- $s_0 \in \boldsymbol{S}$ is the initial screen;

- $\boldsymbol{C} \subseteq \boldsymbol{S} \times \Sigma \times \boldsymbol{S}$ is a set of *connection*s between screens. Given a connection $c = \langle s_1, \sigma, s_2 \rangle \in \boldsymbol{T}$, we call s_1, σ, and s_2 the *source*, the *cause*, and the *destination* of c, respectively.
  A connection $c = \langle s_1, \sigma = \langle w, e \rangle, s_2 \rangle \in \boldsymbol{C}$ in the model indicates that, on active screen s_1, triggering event type e on widget w will make screen s_2 become active.
- $\boldsymbol{F} = \boldsymbol{S}$ is the set of final screens.

Note system default events are modeled in an ESM with the help of a set $\boldsymbol{D}$ ($\boldsymbol{D} \subseteq \boldsymbol{W}$) of dummy widgets. For each screen $s_i \in \boldsymbol{S}$, there exists exactly one dummy widget d_i such that $d_i \in s_i$. System default events like BACK and HOME can, and can only, be triggered on dummy widgets. In this way, we unify the process of widget events and system default events in CHATEM. We do not differentiate the two types of events hereinafter.

Model Construction In the work on ATOM [6], we require as input an event sequence model (ESM) for the app under consideration. In the experiments to evaluate the effectiveness of ATOM, we manually constructed ESMs for the subject mobile apps through investigating both the documentations and the actual behaviors of the apps, and Delta ESMs (DESM) that capture the changes through side-by-side comparison of two app versions. Although most DESMs are much smaller than ESMs, manually constructing the DESMs turns out to be a non-trivial task, requiring considerable knowledge about not only the correspondence between the base version model and the base version app but also how the changes affect the model.

In view of the latest development in GUI model construction for mobile apps, we employ a three-step process in this paper to semi-automatically construct ESMs for apps. First, we use an automated model extraction tool, e.g., Gator [7], to construct an initial ESM for the app under maintenance. Due to limitations of such tools, the result model may miss feasible behaviors that are important for the maintenance task and/or include infeasible behaviors. To rectify the initial model, the second phase of model construction involves executing the app to confirm important behaviors of the ESMs. In this phase, we use the base version test scripts to exercise the app and mark observed behaviors in the models as feasible. Finally, we manually inspect the unmarked events in the models and add important behaviors that are missing. After the three steps, we get the input model for CHATEM. The constructed model contains enough useful information to effectively help test script maintenance, as observed in our experiment. For future work, we plan to investigate how the completeness of ESMs affects the performance of CHATEM and how to effectively enhance the completeness of the constructed ESMs.

The above model construction process is conducted based on the binary form of mobile apps, e.g., apk files of Android apps. Operating on the binary level, rather than on the source code level, makes our approach applicable to a wider range of apps.

B. Change Extraction

Since its ultimate goal is for test script maintenance, CHATEM focuses on GUI changes that influence the execution of test scripts. Given two models $\mathcal{M} = \langle \Sigma, \boldsymbol{S}, \{s_0\}, \boldsymbol{T}, \boldsymbol{S} \rangle$ and $\mathcal{M}' = \langle \Sigma', \boldsymbol{S'}, \{s'_0\}, \boldsymbol{T'}, \boldsymbol{S'} \rangle$ for the base and updated version GUI, CHATEM first extracts changes to the GUI at the screen, widget, and connection levels, respectively.

1) Changes to screens.: CHATEM identifies three sets of screens from both models: $\boldsymbol{S}_+$ contains screens added to the base version app, $\boldsymbol{S}_-$ contains screens removed from the base version app, and $\boldsymbol{S}_\times$ contains screens that are changed between the two versions.

CHATEM relates screens from different versions based on their IDs. That is, two screens are only considered related if they share the same ID. Correspondingly, $\boldsymbol{S}_+$ can be computed easily as $\{s : \boldsymbol{S'} | \forall x \in \boldsymbol{S} : s.id \neq x.id\}$, and $\boldsymbol{S}_-$ as $\{s : \boldsymbol{S} | \forall x \in \boldsymbol{S'} : s.id \neq x.id\}$. All screens within $s \in \boldsymbol{S} - \boldsymbol{S}_-$ are considered retained into the updated version (with or without changes).

Apparently, for each retained screen $s \in \boldsymbol{S} - \boldsymbol{S}_-$, there exists $s' \in \boldsymbol{S'} - \boldsymbol{S}_+$ such that $s.id = s'.id$. s_1 and s_2 are *equivalent*, denoted as $s_1 = s_2$, if the sets of widgets in s and s' are equivalent (see Section IV-B2). Otherwise, s' is the *derivation* of s in the updated version, and s the *origin* of s' in the base version. We use a one-to-one function ξ to encode the derivation relation between screens, and $\boldsymbol{S}_\times$ can be computed as the domain of ξ.

The above change extraction process is based on the assumption that IDs of screens stay the same across different versions. In cases where screen IDs get changed, CHATEM also accepts extra input regarding the *matching* relation between the IDs before and after the changes. Given that IDs assigned to screens seldom change during the evolution of mobile apps, and that most apps have rather limited number of screens, the overall cost for CHATEM to correctly identifying the changes to screens should be easily affordable.

2) Changes to Widgets.: Changes may also take place on the widget level. Similar to the extraction of changes to GUI screens, CHATEM extracts three categories of changes to widgets, denoted using three sets: $\boldsymbol{W}_-$ is the set of widgets that are removed from the base version, $\boldsymbol{W}_+$ is the set of widgets that are added to the updated version, and $\boldsymbol{W}_\times$ is the set of widgets that are retained but with modifications.

CHATEM considers all widgets that are part of an added screen to be also added, i.e., $\forall w.w \in s \wedge s \in \boldsymbol{S}_+ \rightarrow w \in \boldsymbol{W}_+$, and all widgets that are part of a removed screen to be removed, i.e., $\forall w.w \in s \wedge s \in \boldsymbol{S}_- \rightarrow w \in \boldsymbol{W}_-$. The process for deciding whether a widget on a retained screen s_1 ($s_1 \in \boldsymbol{S}_\times$) is added, removed, or retained with modifications involves identifying two types of relations between widgets: the equivalence relation and the derivation relation.

- Equivalence relation. Two widgets w_1 and w_2 ($w_1 \in s_1 \wedge w_2 \in \xi(s_1)$) are considered equivalent if and only if all their properties have the same values, they trigger the same set of events, and their event handlers for each

event transit the app to screens with matching IDs (see Section IV-B1).

- Derivation relation. Given two widgets w_1 and w_2 ($w_1 \in s_1 \wedge w_2 \in \xi(s_1)$), CHATEM considers w_2 to be the derivation of w_1 if 1) w_1 and w_2 are not equivalent and 2) the majority[2] of their locator properties (see Section II-B) have *comparable* values. Two String-typed values v_1 and v_2 are comparable if and only if they have at least half of the words in common; Two values of other types are comparable if and only if they are equivalent. The criteria CHATEM adopts are picked based on our empirical experience. Design of better criteria belongs to future work. We overload the function ξ to also encode the derivation relation between widgets.
  Consider the Notepad app from Section III for example, although the text of the menu item was changed from "`Add note`" to "`Add`", the menu item's other locator properties like `ID` and `XPath` remained the same. Therefore, CHATEM is able to correctly recognize the derivation relation between the items.

Given a retained screen s_1 and its derivation $s_2 = \xi(s_1)$, identification of equivalent widget pairs from s_1 and s_2 is straightforward. Afterwards, CHATEM checks each pair of remaining widgets from the two screens, and decides whether there is a derivation relation between them. Naturally, such checks are only performed between widgets of the same type. For example, CHATEM never attempts to check whether a text field is a derivation of a button.

In the end of this step, all widgets from the base version GUI and with derivations are added to $\boldsymbol{W}_\times$; remaining widgets from the base version GUI and with no equivalences are added to $\boldsymbol{W}_-$; extra widgets from the updated version app and with no equivalences are added to $\boldsymbol{W}_+$. In this way, CHATEM identifies all the widgets that are added, removed, and retained with modifications.

3) Changes to Event Handling.: Since actions in test scripts essentially trigger events on the GUI during testing, CHATEM also extracts changes on the event level.

Given the identified changes to screens and widgets, CHATEM can already derive quite a number of changes to the event handling of the app. In particular, if a screen is added to or removed from the base version app, all event handlers related to the screen are added or removed; If a widget is removed from the base version app, all event handlers associated with that widget are removed; If a widget is added to the updated version app, all event handlers associated with the widget are added; It, however, is not always obvious how modifications to a widget will affect the event handling of the app.

Next, we introduce how CHATEM identifies changes to event handling on modified widgets, based on the type of event a handler handles and the target screen to which the handler will transit the GUI.

[2]That is more than 50%.

Let w_1 and w_2 be two widgets from screens s_1 and s_2 ($s_2 = \xi(s_1)$, $w_2 = \xi(w_1)$), ϵ an event that can be triggered on them, and two screens s_1' and s_2' the destination screens of triggering ϵ on w_1 and w_2, respectively. Connection $c_2 = \langle s_2, \langle w_2, \epsilon\rangle, s_2'\rangle$ is the *derivation* of connection $c_1 = \langle s_1, \langle w_1, \epsilon\rangle, s_1'\rangle$, if and only if $s_2' = \xi(s_1')$. That is, the two connections c_1 and c_2 have the derivation relation if and only if the destination screens of the two connections have also the derivation relation. Otherwise, CHATEM deems c_1 as being removed and c_2 as being added.

In this way, CHATEM computes three more sets to capture changes on the event handling level: $\boldsymbol{C}_-$ is the set of connections that are removed; $\boldsymbol{C}_+$ is the set of connections that are added; $\boldsymbol{C}_\times$ is the set of connections that are retained, but with changes to their source widgets or destination screens.

C. Test Simulation and Change Impact Identification

The goal here is to identify ranges within test scripts that need to be revised.

To understand how the extracted changes affect the execution of test scripts, first we have to find out how each test exercises the app, i.e., which screens and widgets a test interacts with and which connections the test actions cover. We discover such information through test simulation.

Given the ESM $\mathcal{M} = \langle \Sigma, \boldsymbol{S}, \{s_0\}, \boldsymbol{T}, \boldsymbol{S}\rangle$ for a base version app, a test script t and its sequence of test actions $[a_1, a_2, \ldots, a_n]$ for the app, CHATEM starts from s_0 and constructs the simulation of t on $\mathcal{M}$. The construction is done in an iterative process which handles the simulation of one action at a time. Let screen s_i be the currently active screen of the app and $a_i = \langle l_i, e_i\rangle$ be the next action execute at the beginning of an iteration, test simulation decides the widget w_i on s_i based on widget locator l_i, and then follows the connection as defined by the handler for event e_i on w_i, reaching a (possibly different) screen s_j. The next iteration of test simulation will then use s_j as the active screen and simulate action a_{i+1}. The process continues until all the actions in t have been processed. Note the above simulation can always continue till the end of each test script, since all test scripts are used for model construction (see Section IV-A).

During test simulation, we record for test action a the following information: i) the active screen s right before a's execution, ii) the locator l used in a, iii) the type of the event triggered by a, and iv) the destination screen s'. Base on the recorded information, we are able to associate a connection $\langle s, \sigma, s'\rangle$ to a. In change extraction, we collected already the changes applied to the screens, widgets, and event handling of the base version app. By relating the associated connections of test actions to the extracted changes, CHATEM can easily identify which test actions are affected by the changes: If the associated connection of an action is affected by changes, we need to update the test action during test maintenance.

In particular, let $\mathcal{S} = \langle s_0, \sigma_0, s_1\rangle, \langle s_1, \sigma_1, s_2\rangle, \ldots, \langle s_{n-1}, \sigma_{n-1}, s_n\rangle$ be the sequence of connections that t exercises on $\mathcal{M}$ and Θ_Δ ($\Theta \in \{\boldsymbol{S}, \boldsymbol{W}, \boldsymbol{C}\}$ and $\Delta \in \{+, -, \times\}$) be the sets

capturing the changes, to identify test actions affected by the changes, CHATEM goes through the following steps:

1. CHATEM adds to set $\boldsymbol{R}_{-}$ ranges within $\mathcal{S}$ where all intermediate screens are removed but both end screens are retained. In other words, a sequence $s_i, s_{i+1}, \ldots, s_{j-1}, s_j$ $(j - i \geq 2)$ is added if and only if screens $s_i, s_j \in \boldsymbol{S} - \boldsymbol{S}_{-}$ but $s_k \in \boldsymbol{S}_{-}$ $(i < k < j)$. Since all the intermediate screens of the sequence are removed, CHATEM needs to find a new action sequence to navigate the updated version app from $\xi(s_i)$ to $\xi(s_j)$.

2. CHATEM also adds to set $\boldsymbol{R}_{-}$ the single-action ranges whose associated connections are removed from the base version app. That is, test action a_i with the associated connection c_i is added, as range $\langle t, i, i+1 \rangle$, if and only if $c_i \in \boldsymbol{C}_{-}$.

3. CHATEM adds to set $\boldsymbol{R}_{\times}$ the single-action ranges whose associated connections are modified between the base and updated versions. That is, test action a_i with the associated connection c_i is added, as range $\langle t, i, i+1 \rangle$, if and only if $c_i \in \boldsymbol{C}_{\times}$.

At the end of this step, CHATEM produces two sets $\boldsymbol{R}_{\times}$ and $\boldsymbol{R}_{-}$ of ranges within the test scripts that need to be updated and replaced, respectively.

D. Maintenance Action Construction and Application

Given set $\boldsymbol{R}_{-}$ of ranges within test scripts to be replaced, maintenance action construction involves mainly building an alternative connection sequence $c, ..., c'$ in $\mathcal{M}'$ for each range $r = \langle t, x, y \rangle \in \boldsymbol{R}_{-}$, such that $c.source = \xi(t[x].source) \wedge c'.destination = \xi(t[y].destination)$. For that purpose, CHATEM performs a breadth-first search on $\mathcal{M}'$ for a path from $\xi(t[x].source)$ to $\xi(t[y].source)$. If the search is successful, the result connection sequence will be used to guide the construction of the replacement test actions for range r. If no such path can be found, the maintenance of test t is unsuccessful and all test actions following the range r in t will be discarded. To make the maintenance action easy to understand, CHATEM restricts that alternative paths should have no more than MAXPATHLENGTH (set to 2 by default) connections.

For each single-action range $a = \langle l, e \rangle$ in $\boldsymbol{R}_{\times}$, let s be the source screen of a and w the widget that a interacts with, if locator l resolves to the derivation of w in $\mathcal{M}$, no change to the action is required. Otherwise, CHATEM revises a to use a locator based on $\xi(w)$.

Since new behaviors may be added due to the changes, CHATEM also implements a simple random approach to extend the test scripts after maintenance by triggering events on added widgets or covering added screens.

V. EXPERIMENTAL EVALUATION

To evaluate the performance of CHATEM in maintaining GUI test scripts, we conducted experiments that applied CHATEM to 16 mobile apps. This section reports on the experiments and provides preliminary evaluations of CHATEM's effectiveness.

A. Implementation

Although the underlying technique should be applicable to maintenance of tests written in different script languages, the current implementation of CHATEM targets scripts based on the Robot Framework for testing GUIs of Android apps. In particular, the Appium open source test automation framework and the AppiumLibrary are used in CHATEM to drive and communicate with the Android app under testing. For future work, we will add also support for other GUI testing frameworks.

The CHATEM technique is not concerned with the construction of ESMs of the Android apps. Nevertheless we created a prototype tool based on GATOR[3] to facilitate the model construction process in an semi-automatic way.

All the experiments ran on an Ubuntu 16.04 machine with 3.1 GHz Intel dual-core CPU and 16 GB of memory. CHATEM was the only computationally-intensive process running during the the experiments. CHATEM spent less than one minute to finish maintaining all the test scripts of each app.

B. Measures

Since CHATEM and ATOM [6] essentially serve the same purpose, i.e., to automatically maintain scripts for GUI testing of Android apps, we adopt in this work the same three measures for performance as used in [6]. Such design enables us to directly reuse the experimental results from [6] for comparison between CHATEM and ATOM.

The three measures are screen coverage preservation (SCP), connection coverage preservation (CCP), and test action preservation (TAP). For the completeness of this section, we briefly explain the rationale behind the measures. Detailed definitions of the measures can be found in [6]. SCP and CCP measure how many percentage of screens and connections covered by the original test scripts on the base version app are still covered by the tests on the updated version app after maintenance. The higher the SCP and CCP, the more behaviors of the app are still tested. TAP measures how many percentage of the actions in the original test scripts are rescued into the updated tests. The higher the TAP, the more we preserve from previous testing effort.

Based on these measures, we designed our experiments to answer the following research questions:

- **RQ1**: How effective is CHATEM in test scripts maintenance, in terms of SCP, CCP, and TAP, respectively?
- **RQ2**: How does CHATEM compare with ATOM in terms of effectiveness in test script maintenance?

C. Subjects

To facilitate the comparison with previous work, we include the 11 Android apps that were used for the evaluation of ATOM [6]. We also select 5 more production apps from the subjects for evaluating Gator[3] [7]. Therefore, in total we use 16 Android apps for the evaluation. Table I lists all the apps and briefly describes each app.

[3]http://web.cse.ohio-state.edu/presto/software/gator/

TABLE I
DESCRIPTION OF THE MOBILE APPS AS SUBJECTS.

App	Brief Description	Base version	Updated version
Bilibili	A video sharing website.	4.12.1	4.13.0
GNotes	A simple note app.	1.0.2	1.0.3
Wannianli	A calendar app.	4.4.2	4.4.6
YoudaoNote	A cloud-based note app.	5.1.0	5.2.0
Wechat Phonebook	A phonebook app.	3.5.1	4.2.0
Changba	A Karaoke app.	6.7.1	7.0.0
Baidu Music	A music player.	5.6.6.1	5.7.0.3
365 Calendar	A calendar app.	6.0.2	6.2.3
Ctrip	An online travel agent.	6.15.2	6.16.0
WizNote	A cloud-based information management app.	7.1.0	7.1.6
TickTick	A to-do list app.	2.6.6	2.6.7
APV	A file management app.	0.4.0	-
NotePad	A simple note app.	1.3	-
OpenManager	A server management app.	2.1.8	-
SuperGenPass	A password-generation app.	2.2.2	-
BarcodeScanner	A barcode scanner app.	4.4	-

TABLE II
MODEL INFORMATION

App	Base version		Updated version	
	S	C	S	C
Bilibili	19	41	15	25
GNotes	18	38	16	33
Wannianli	14	40	15	42
YoudaoNote	19	44	19	45
Wechat Phonebook	14	27	13	26
Changba	19	69	19	64
Baidu Music	16	34	14	29
365 Calendar	19	42	18	40
Ctrip	19	34	19	34
WizNote	15	29	13	25
TickTick	15	33	17	36
APV	7	18	6	18
NotePad	10	29	10	27
OpenManager	11	24	11	24
SuperGenPass	8	23	8	23
BarcodeScanner	9	17	9	17
Total	232	542	222	508

For the 11 subject apps from [6], we reuse the same test scripts and the same base and updated version apps as were used in [6] to run CHATEM, which enables us to make a direct comparison on the basis of existing results [6].

As for the remaining 5 apps, their corresponding versions from [7] are used as the base version, and two graduate students with over three-year experience in Android development were asked to modify the GUIs of the apps to produce the updated versions. The graduate students had no knowledge about the scripts for testing the apps, which are also adopted from [7]. The process described in Section IV-A is then employed to prepare the ESMs for the base and updated versions of each app.

Detailed information about the test scripts used in the experiments can be found in [6] and [7], respectively. For space reasons, we do not reproduce the same information in this paper.

D. Experimental Results

This section reports the result of our experiments to answer the question listed in Section V-B.

RQ1: Effectiveness of CHATEM. Table IV summarizes the values of measures that we use for evaluating the effectiveness of CHATEM. For each app, the table lists the SCP, CCP, and TAP of the maintenance result. Each entry in column SCP is in form $x(y/z)$, where x is the value of the measure, y is the number of shared screens that are covered by the scripts after maintenance, and z is the number of shared screens covered by the original scripts in the base version app. Values listed in column CCP are similar as those in column SCP, but regarding shared connections. Each entry in column TAP is also in form $x(y/z)$, where x is the value of the measure, y is the number of extra test actions retained by the maintenance process, and z is the number of test actions that would be lost if no maintenance is performed.

From the table, we can see that CHATEM is able to achieve SCP and CCP values larger than 0.8 for all the subject apps, and the overall SCP and CCP values are larger than 0.95, which strongly suggests that CHATEM is effective in retaining the coverage of the ESM during test maintenance. Also, TAP values are larger than 0.6 for all but one app, and the overall TAP value is close to 0.8, which indicates that CHATEM is effective in rescuing test actions that would be lost if no maintenance is performed.

CHATEM *is effective in retaining the coverage of screens and connections and rescuing test actions during test script maintenance.*

RQ2: Comparison with ATOM. Table IV also presents the maintenance results of ATOM [6] on the same set of subject faults. The values for the top 11 apps are reproduced from [6], while those for the bottom 5 apps are gathered after applying the tool on the 5 apps. Values of the same measure from different tools are listed side by side to facilitate the comparison.

From the table, we can see that, although ATOM already does a good job in achieving high SCP, CHATEM is able to match, and sometimes even outperform, ATOM. As for CCP, CHATEM performs as good as or better than ATOM on all apps. In terms of TAP, the performance of CHATEM is almost the same as that of ATOM.

To quantify the difference in maintenance performance with CHATEM and ATOM, we performed a pair-wise comparison of the SCP, CCP, and TAP measures. In particular, we

TABLE III
TEST SCRIPTS FOR BASE VERSION

App	Scripts number	$Min_{TestActions}$	$Max_{TestActions}$	Total number of test actions	State Coverage	Connection Coverage
Bilibili	16	4	21	233	1	1
GNotes	9	8	34	197	1	0.89
Wannianli	9	9	27	125	1	0.93
YoudaoNote	12	9	17	140	1	0.83
Wechat Phonebook	7	4	10	48	1	0.92
Changba	27	6	22	353	1	1
Baidu Music	20	4	14	114	1	1
365 Calendar	10	5	45	218	1	0.97
Ctrip	13	5	30	247	0.94	0.94
WizNote	12	16	28	260	1	0.96
TickTick	15	4	21	161	1	1
APV	9	10	14	105	1	1
NotePad	14	7	13	133	1	1
OpenManager	13	6	10	104	1	1
SuperGenPass	10	8	15	107	1	1
BarcodeScanner	8	7	11	71	1	1
Total	204	4	45	2616	0.99	0.96

TABLE IV
EXPERIMENTAL RESULTS

APP	SCP		CCP		TAP	
	ATOM	CHATEM	ATOM	CHATEM	ATOM	CHATEM
Bilibili	1(14/14)	1(14/14)	1(24/24)	1(24/24)	0.26(35/134)	0.26(35/134)
GNotes	0.94(15/16)	1(16/16)	1(30/30)	1(30/30)	0.87(138/159)	0.89(141/159)
Wannianli	1(14/14)	1(14/14)	1(32/32)	1(32/32)	0.95(91/96)	0.95(91/96)
YoudaoNote	0.95(18/19)	1(19/19)	1(31/31)	1(31/31)	0.91(116/128)	0.91(116/128)
Wechat Phonebook	0.85(11/13)	0.92(12/13)	0.91(17/19)	1(19/19)	0.95(37/39)	0.95(37/39)
Changba	1(16/16)	1(16/16)	0.98(55/56)	0.98(55/56)	0.74(97/131)	0.74(97/131)
Baidu Music	1(14/14)	1(14/14)	1(29/29)	1(29/29)	0.90(79/88)	0.90(79/88)
365 Calendar	0.88(14/16)	0.88(14/16)	0.92(34/37)	0.92(34/37)	0.65(31/48)	0.65(31/48)
Ctrip	0.95(18/19)	1(19/19)	0.91(30/32)	1(32/32)	0.96(50/52)	0.96(50/52)
WizNote	1(13/13)	1(13/13)	1(24/24)	1(24/24)	0.93(82/88)	0.93(82/88)
TickTick	0.93(14/15)	1(15/15)	1(32/32)	1(32/32)	0.73(16/22)	0.73(18/22)
APV	1(6/6)	1(6/6)	1(16/16)	1(16/16)	0.89(34/38)	0.89(34/38)
NotePad	1(9/9/)	1(9/9)	0.96(24/25)	1(25/25)	0.79(50/63)	0.79(50/63)
OpenManager	1(11/11)	1(11/11)	1(24/24)	1(24/24)	1(39/39)	1(39/39)
SuperGenPass	1(8/8)	1(8/8)	1(22/22)	1(22/22)	1(49/49)	1(49/49)
BarcodeScanmer	1(9/9)	1(9/9)	1(17/17)	1(17/17)	1(29/29)	1(29/29)
Total	0.96(204/212)	0.98(209/212)	0.98(441/450)	0.99(446/450)	0.81(973/1203)	0.81(978/1203)

performed two-sided Wilcoxon Rank Sum paired tests to check whether the differences are statistically significant. The result p-values are 0.15, 0.18, and 0.18 for SCP, CCP, and TAP, respectively, suggesting the differences in performance between the two techniques is *not* significant.

The overall performance of CHATEM *is comparable with that of ATOM.*

E. Threats to Validity

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

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 focus on the screen and transition coverage preservation as well as test action preservation. While these measures are useful in that they reflect how many app functionalities are still tested and how many test actions retained after maintenance, the measures are not concerned with the screens and transitions that are not covered by tests in the base version app. To take into account also the overall coverage information, new measures need to developed, which we leave for future work.

Another threat here concerns the construction of the models for the apps. Even with the help of tools, model extraction remains a non-trivial task. In this work, GATOR was able to produce initial models of reasonably good quality, which reduces significant amount of manual work for model confirmation. But different people may come up with different models in the end, and the amount of manual effort required may be very different with other apps. In view of the latest advancements in both tool and technique development for Android app modeling, we plan to investigate on ways to combine the strengths of various tools and techniques to produce high quality GUI models for Android apps.

A third threat to construct validity lies in the fact that test actions are considered preserved if they are executable after maintenance. Such criteria, however, might be too weak: a maintained test action may exercise a completely different functionality than before, and when that happens, the original test action is preserved on only the syntax, but not the semantics, level. Next, we plan to develop a more comprehensive measure that considers both syntactical and semantical aspects of test actions for evaluating the effectiveness of maintenance.

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

The main threat to internal validity is in the possible faults in the implementation of our approach. To mitigate the threat,

we review our code and experimental scripts to ensure their correctness before conducting the experiments.

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

Android apps used as subjects in this experiments may pose threats to external validity. The 11 Android apps from [6] were closed-source production apps and originally selected from a Chinese Android app market. To mitigate the problem, we also included 5 subject apps from existing research work [7]. Nevertheless, these subjects may not be representative of all the mobile apps. As future work, we plan to carry out more extensive experiments on also other apps from various sources.

VI. Related Work

Four research areas are closely related to this work: model-based GUI testing, Android GUI test generation, change acquisition, and test scripts maintenance. This section provides a brief review of the existing work in each area.

A. Model-based GUI testing

The application of ChaTeM involves creating a GUI Model of the app under test(AUT) to represent its behavior and using the GUI Model to automatically generate test cases. Researchers have proposed various approaches to test generation. Reverse engineering techniques, such as GUI crawling and GUI ripping, are often used to generate GUI models of the AUT for GUI testing. Amalfitano et al. [8] present an approach for model-based GUI testing which is based on a crawler that automatically creates the GUI model of the AUT, and generates test cases that can be automatically executed. Tool support for the proposed approach was also provided. Memon et al. [9] propose to use Event-flow Graphs (EFG) for modelling possible event sequences that may be executed on a GUI and present Regression Tester to create EFG based on dynamic analysis.

Static analysis techniques were also used to build GUI Models for the AUT. Yang et al. [7] proposed the Gator system to create the window transition graph (WTG) for apps, which extracts windows, events and callbacks from apps through both static and dynamic analysis of the android application. One of the key contributions of their work is in the analysis of the run-time stack for collecting information about the currently active window.

B. Android GUI test generation

Various GUI test techniques for android apps have been proposed to ensure their quality. Gao et al. [10] give a review of testing approaches for mobile apps. Approaches have been proposed for various goals and environments. Since most mobile apps are event-driven, many approaches construct models for mobile apps, e.g., using techniques reviewed in Section VI-A, and use the models to guide test generation.

Wu et al. [11] propose an approach to generate gesture events for android applications. In existing event generation techniques, gestures are generated randomly or enumerated to cover all possible cases. To reduce the amount of gesture sequences one has to examine, they present a static analysis technique to obtain the gesture information about every UI component's potentially *relevant* gestures. Miguel et al. [12] propose an approach to generate test cases based on the GUI and usage-model of Android apps.

C. Change acquisition techniques

Grechanik et al. [4] propose an approach for test script maintenance and implement a tool named REST. REST extracts the GUI Trees for the two versions of a GUI application, where the root node of the tree corresponds to the application window, leaf nodes correspond to the most primitive GUI components like buttons and text fields, and the parent-child relationship between nodes on the tree models the including relationship between the corresponding components on the GUI. Then REST identifies changes between the two trees, locates test actions that directly or indirectly depend on the modified GUI objects. Finally, REST raises warning against the identified test actions. Testers need to manually examine the reported test actions and repair them if necessary. Due to the non-linear growth in time and memory cost caused by the inherent complexity and implementation restrictions, REST is only able to handle medium-sized applications with hundreds of GUI components. Raina et al. [13] propose a tool for automatically obtaining the changes of a web application. The tool extracts changes in two steps: It first builds the DOM trees of the two versions of the web application and compares the two trees to obtain the changes of the new version application, and then only tests the modified part of the web application. Choudhary et al. [14] propose the WATER approach for web application test repair. The approach is based on differential testing. They execute the test scripts written for the base version application on both two versions. According to an analysis of the differences between the two execution, WATER gives suggestions regarding how the test scripts should be repaired. Xing et al. [15] propose an approach for detecting structural changes between two models of a Java software system. The approach outputs a change tree as the result, and it handles not only moving, adding, removing, and renaming of components, but also changes to attributes and dependencies. They evaluate the correctness and robustness of the approach using a real-world case study.

D. Test scripts maintenance

In regression testing, test scripts for the old version app often become unusable on the new version app because of changes implemented in response to evolved requirements. Roser et al. [16] provide a survey showing that 31 regression testing techniques have been proposed in the past 15 years, and that most of the regression testing research [17]–[20] were focused on test minimization. The goal of our work is to maintain obsolete test scripts based on the changes. Because of limitations of dynamic analysis techniques, Regression Tester [9] has been used only in research work, but not to generate test cases in real applications. Gao et al. [5] present

ScrIpT repAireR (SITAR) to repair unusable low level test scripts. First, SITAR maps test scripts to event sequences and the new version app to an annotated EFG, with test actions that can not be mapped to any event assigned to NULL. Then SITAR finds the events which were assigned to NULL and replaces them with manually determined alternative paths on the annotated EFG. User assistance is needed in making decisions (e.g., confirmations, modifications and additions) in the repair process.

Memon et al. [21] propose a method to obtain changes by comparing the EFGs of two versions of an application, assuming that the events and windows all have unique names and that the windows and widgets are only renamed if their functionality is changed. In our previous work [6], we get changes manually and adopt fuzzy matching between models and scripts. In this paper, we proposed a new approach to automatically extract changes from two models. The rationale behind such change is in that, although models for whole apps are larger than those for only the changes, constructing the model for an app will become more and more economic, thanks to the latest developments in automatic model construction.

VII. Conclusions and Future Work

In this paper, we propose an approach, named CHATEM, to automatically maintaining test scripts for regression GUI testing. Given the event sequence models (ESMs) for two versions of an Android app, CHATEM automatically extracts the differences between the two GUIs, generates maintenance actions for segments of affected test scripts, and applies those actions to bring the test scripts in sync with the updated app. Experimental evaluation of CHATEM on real world Android apps shows that CHATEM is effective in maintaining test scripts.

The usefulness of CHATEM largely depends on the availability and quality of ESMs for Android GUIs. At the current stage, model construction is still a semi-automatic process that involves both tool application and human confirmation. To facilitate model construction, next we plan to develop a technique that brings several state-of-the-art model construction tools together for the effect of synergy.

We conjecture that the maintenance results produced by CHATEM are easier for programmers to understand and therefore more likely to be accepted, since the maintenance actions are directly related to each segment of affected test scripts. Due to limited time, we, however, were not able to conduct experiments to prove or disprove the conjecture. In the future, we plan to carry out user experiments using more subject apps to find out what programmers like or dislike about the test scripts maintained by CHATEM.

Acknwledgement

The authors thank anonymous reviewers for their comments and advices on improving this paper.

References

[1] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software TestS,AST'11*. ACM, 2011, pp. 77–83.

[2] F. Gross, G. Fraser, and A. Zeller, "EXSYST: search-based GUI testing," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 1423–1426.

[3] 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*, 2015, pp. 429–440.

[4] 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*, 2009, pp. 408–418.

[5] 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.

[6] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and L. Xuandong, "Atom: Automatic maintenance of gui test scripts for evolving mobile applications," in *IEEE International Conference on Software Testing, Verification and Validation*, 2017, pp. 658–668.

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

[8] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI crawling-based technique for android mobile application testing," in *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, 2011, pp. 252–261.

[9] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *10th Working Conference on Reverse Engineering, WCRE 2003, Victoria, Canada, November 13-16, 2003*, 2003, pp. 260–269.

[10] J. Gao, X. Bai, W. Tsai, and T. Uehara, "Mobile application testing: A tutorial," *IEEE Computer*, vol. 47, no. 2, pp. 46–55, 2014.

[11] X. Wu, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Testing android apps via guided gesture event generation," in *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*, 2016, pp. 201–208.

[12] J. L. S. Miguel and S. Takada, "GUI and usage model-based test case generation for android applications with change analysis," in *Proceedings of the 1st International Workshop on Mobile Development, Mobile!@SPLASH 2016, Amsterdam, Netherlands, October 31, 2016*, 2016, pp. 43–44.

[13] S. Raina and A. P. Agarwal, "An automated tool for regression testing in web applications," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 4, pp. 1–4, 2013.

[14] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "Water: Web application test repair," in *Proc. 1st Int. Workshop End-to-End Test Script Eng.*, 2011, pp. 24–29.

[15] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, 2005, pp. 54–65.

[16] R. H. Rosero, O. S. Gómez, and G. Rodriguez, "15 years of software regression testing techniques - A survey," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 5, pp. 675–690, 2016.

[17] H. Hsu and A. Orso, "MINTS: A general framework and tool for supporting test-suite minimization," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 419–429.

[18] C. Lin, K. Tang, and G. M. Kapfhammer, "Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests," *Information & Software Technology*, vol. 56, no. 10, pp. 1322–1344, 2014.

[19] Q. C. D. Do, G. Yang, M. Che, D. Hui, and J. Ridgeway, "Regression test selection for android applications," in *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016*, 2016, pp. 27–28.

[20] X. Wang and H. Zeng, "History-based dynamic test case prioritization for requirement properties in regression testing," in *Proceedings of the International Workshop on Continuous Software Evolution and Delivery, CSED@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*, 2016, pp. 41–47.

[21] 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.