ATOM: Automatic Maintenance of GUI Test Scripts for Evolving Mobile Applications

Xiao Li*[1], NaNa Chang*, Yan Wang*, Haohua Huang*, Yu Pei†[2], Linzhang Wang*[3], Xuandong Li*[4]
*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
Jiangsu Novel Software Technology and Industrialization, Nanjing, China
†Department of Computing, The Hong Kong Polytechnic University, Hong Kong S.A.R., China
lx.lily.lee@gmail.com[1] yupei@polyu.edu.hk[2] lzwang@nju.edu.cn[3] lxd@nju.edu.cn[4]

***Abstract*—The importance of regression testing in assuring the integrity of a program after changes is well recognized. One major obstacle in practicing regression testing is in maintaining tests that become obsolete due to evolved program behavior or specification. For mobile apps, the problem of maintaining obsolete GUI test scripts for regression testing is even more pressing. Mobile apps rely heavily on the correct functioning of their GUIs to compete on the market and provide good user experiences. But on the one hand, GUI tests break easily when changes happen to the GUI; On the other hand, mobile app developers often need to fight for a tight feedback loop and are left with limited time for test maintenance.**

In this paper, we propose a novel approach, called ATOM, to automatically maintain GUI test scripts of mobile apps for regression testing. ATOM uses an event sequence model to abstract possible event sequences on a GUI and a delta ESM to abstract the changes made to the GUI. Given both models as input, ATOM automatically updates the test scripts written for a base version app to reflect the changes. In an experiment with 22 versions from 11 production Android apps, ATOM updated all the test scripts affected by the version change; the updated scripts achieve over 80% of the coverage by the original scripts on the base version app; all except one set of updated scripts preserve over 60% of the actions in the original test scripts.

I. INTRODUCTION

Modern software development practices like continuous integration often have regular and frequent regression testing as an integrated part to ensure that changes to a program do not break existing functionality. For regression testing to be effective and efficient, the tests need to be updated to reflect the evolved program behavior or specification. Such maintenance of regression tests, however, is expensive, largely because it often requires manual effort. Sometimes the cost is so high that engineers would rather write new tests than to update the old ones [1].

With the ever growing popularity of mobile devices, mobile applications, or apps, are becoming indispensable in our personal lives and at work. Most mobile apps interact with users through rich graphical user interfaces (GUIs), making GUI testing an essential part of the regression testing for those apps. GUI test scripts typically refer to exact sequences of actions to be performed on specific GUI widgets and are highly sensitive to changes in the structure or workflow of the application GUI. In practice, on the one hand, GUIs of most mobile apps are constantly tuned to provide better user experiences, obsolescing many GUI test scripts and creating a high demand for the maintenance of those scripts; On the other hand, due to fierce competition on the market, mobile app developers tend to fight for a tight feedback loop and release more often, leaving limited time for regression testing in general and test maintenance in particular. The acute contradiction between the increased demand and the limited time for GUI test maintenance renders the manual way of updating such tests much less appealing, if not impractical. Mobile app developers are in dire need of effective and efficient techniques for GUI test maintenance.

Techniques have been developed in recent years to automatically generate test scripts for mobile apps [2]–[4]. Such techniques can be used to help alleviate the problem, but they do not outdate the requirements for maintaining GUI test scripts. First, generating enough test scripts to achieve high coverage of the app in testing is a demanding task. Always generating new test scripts for each regression testing can be prohibitively expensive. Second, regression test scripts often contain manually created or customized scripts that incorporate valuable expert knowledge about the application domain and are less likely to be generated automatically. Throwing away such scripts is not desirable in many cases.

Researchers have proposed different techniques to repair such obsolete GUI test scripts for regression testing. For example, Memon [5] present Regression Tester that uses dynamic analysis to extract an event-flow graph (EFG) to model possible event sequences that may be executed on a GUI, and repairs obsolete test scripts based on the EFG using four user-defined transformations. Due to inherent limitations in dynamic analysis techniques and EFG models, Regression Tester, however, does not directly apply to manually scripted test cases [6]. Gao et al. [6] present the SITAR system to interactively repair obsolete low level test scripts. SITAR does this by mapping low level test scripts to an EFG model for the GUI, repairing the model-level test cases, and then synthesizing low level test scripts again. If a test script action cannot be mapped to the model, e.g., due to the incompleteness of the model, user input is required. SITAR constructs the EFG in the same way as Regression Tester. A more detailed review

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.22

SC$_a$ SC$_b$ SC$_c$ SC$_d$

Fig. 1: Three screens and their corresponding menu items in the NotePad app.

of techniques for test script repair is included in Section V-D.

In this paper, we propose a novel approach, called ATOM, to automatically maintain GUI test scripts of mobile apps for regression testing. ATOM uses an event sequence model (ESM) to abstract possible event sequences in a GUI and a delta ESM (DESM) to abstract the changes made to a GUI. Given the ESM for the base version of an app and the DESM for the changes introduced in an updated version, ATOM automatically updates the test scripts written for the base version app. ATOM achieves this by first computing the simulations of test scripts on the ESM, then updating simulations to reflect changes in the DESM, and in the end synthesizing test scripts based on the updated simulations. Unlike SITAR, ATOM automatically searches for alternative maps when a test action does not have a direct map in the updated model. After the maintenance, the scripts are able to test most remaining parts of the app in the updated version and preserve most of the actions.

The input models required by ATOM may be constructed manually or through automatic mechanisms. We consider the overhead of model construction acceptable, even when it is done manually, for two reasons. First, an ESM (or a DESM) has a direct connection with the corresponding app, lowering the bar for model construction. Second, the construction of an ESM is only necessary when ATOM is applied to an app for the first time. In subsequent uses, only DESMs for the changes need to be built. As differences between two adjacent versions of an app are often small, we expect they are relatively easy to model. During test script maintenance, ATOM also merges the input ESM and DESM to produce an ESM for the updated version app. Such ESM can be used as input for the next application of ATOM.

We have implemented the approach into a tool, also called ATOM. We applied ATOM to maintain GUI test scripts of 11 production apps from a Chinese Android market, using in total 22 different versions, i.e., 2 versions from each app. As the result, ATOM was able to update all the test scripts affected by the changes between versions; the updated scripts achieve over 80% of the coverage by the original scripts on the base version app; all except one set of updated scripts preserve over 60% of the actions in the original test scripts.

The remainder of this paper is organized as follows: Section II illustrates ATOM from a user's perspective using an example mobile app. Section III describes the individual steps of our approach. Section IV reports on the experiment we conducted to evaluate the effectiveness of ATOM. Section V reviews related work in GUI testing for mobile apps. Section VI concludes the paper and presents future work.

II. AN ATOM EXAMPLE

In this section, we use a simple Android App named NotePad to demonstrate from a user's perspective how ATOM can be used to automatically maintain GUI test scripts during the evolution of mobile applications.

NotePad is a sample app shipped with the Android SDK[1] implementing basic functionalities for note taking. Figure 1 shows four screens from the GUI of NotePad and, on the bottom of the first three screens, the corresponding menu a user can call up by pressing the Menu physical key. Henceforth, we refer to a widget simply by the text on it when the meaning is clear from the context.

SC$_a$ is the initial screen when NotePad is launched in a typical scenario, with previously saved notes listed. A user can click on Add note on this screen to create a new note and start editing that note on screen SC$_b$. On SC$_b$, once the editing is done the user may opt to Save or Discard the changes by clicking on the corresponding menu item and return back to SC$_a$. A user can also click on a note item on SC$_a$ and open the note for editing on SC$_c$. Later on, the user can Save the changes, Revert changes, Delete the note, or Edit the title of the note. We refer to this implementation of NotePad as Version 1.0.

Figure 2(a) shows three test scripts written in Robot Framework[2] for testing NotePad Version 1.0. Each script defines a sequence of *action*s to be taken during the test, one action per line. All the three test scripts here start execution from SC$_a$. TS$_1$ first creates a new note, then inputs some text, and at the end saves the input. TS$_2$ is similar as TS$_1$, but the changes are discarded at the end. TS$_3$ assumes the presence of a note item named note1. It first opens the note by clicking on the

[1]https://developer.android.com/studio/index.html
[2]http://robotframework.org/

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) Version 1.0

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

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

(b) Version 2.0

Fig. 2: Test Scripts for NotePad

note item, then clicks on Delete to remove the note. We say a test script runs successfully if all its actions can be performed without causing any error, or fails if otherwise. On Version 1.0 of NotePad, all the test scripts run successfully.

We then modify the GUI of NotePad to produce its next version, mimicking what might happen to an app during its life cycle. The modifications include the following. First, the Add note menu button on SC_b is changed to Add; Second, a confirmation modal dialog[3] with Yes and No buttons is added after Delete is clicked on SC_c; Third, the Discard menu item is removed from SC_c. We refer to this modified implementation of NotePad as Version 2.0.

Under such changes, some of the original test scripts are now *obsolete*, i.e., they do not describe acceptable action sequences of the application. In our example, TS_1 and TS_2 will both fail as SC_a no longer has the menu item Add note; although TS_3 will not fail, it does not really delete the note either.

Taken both versions 1.0 and 2.0 of NotePad, a model describing the feasible event sequences in Version 1.0, and another model capturing the changes introduced by Version 2.0 as the input, ATOM automatically updates the existing test scripts to stay in sync with the application. During the process, ATOM preserves the actions when possible and updates or extends them when necessary. Figure 2(b) shows the result test scripts produced by ATOM, with added and modified actions highlighted. TS_1 is updated to reflect the change of menu item name from Add note to Add; Besides of 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.

[3]A modal dialog is a dialog that disables the rest of the application. A user must interact with the dialog before s/he can go back to the parent application. Without loss of generality, ATOM treats a modal dialog as a screen.

III. How ATOM Works

Let us now describe the process of applying ATOM to maintain GUI test scripts when an app evolves from a base version to a new version. Figure 3 summarizes the information used or produced during the process.

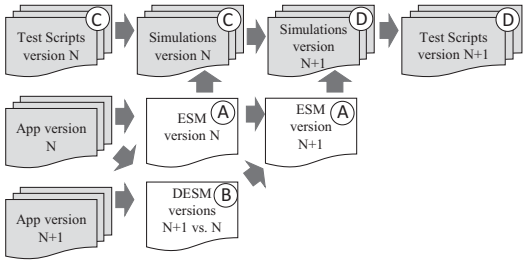


Fig. 3: Information used or produced during an application of ATOM. The letter on the top-right corner of a symbol denotes the subsection where more description about the corresponding information is available.

A. Event Sequence Model

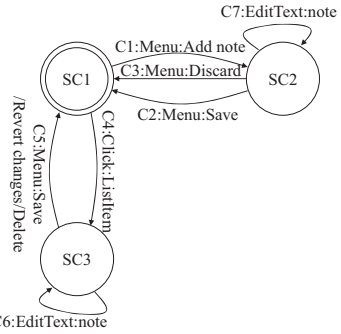


Fig. 4: Partial ESMs for Notepad versions 1.0 (ESM_1). SC_a is the initial screen. Labels on transitions give the corresponding event types and widget names.

To achieve automatic maintenance of the test scripts, ATOM makes use of event sequence models (ESMs) to describe the behaviors of an app. To be both powerful enough to support test script maintenance and simple enough to facilitate manual or automatic construction, an ESM leaves out information about the internal states of an app (e.g., variables) and focuses on the GUI elements like widgets and screens as well as events on them.

Formally, let $\boldsymbol{W}$ be the set of widgets in an app and $\boldsymbol{E}$ the set of event types on $\boldsymbol{W}$, a ESM for the app is a non-deterministic finite state machine $\mathcal{M} = <\Sigma, \boldsymbol{S}, \{s_0\}, \boldsymbol{C}, \boldsymbol{F}>$, 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 $c = <s_1, \sigma, s_2> \in \boldsymbol{C}$, we call s_1, σ, and s_2 the *source*, the *cause*, and the *destination* of c, respectively.
- $\boldsymbol{F} = \boldsymbol{S}$ is the set of final screens.

In everyday use, an event may transit an app from one screen to different others based on the specific program state in which the event is triggered. For example, depending on whether a mobile device is connected to the Internet through WIFI or not, a click on a link may cause the link to be opened on a new screen or a dialog to pop up to let the user decide whether the link should be opened at all. The nondeterminism of the model is to reflect such possibility.

The model does not distinguish a particular set of screens as final, as a script may stop execution at any screen during testing. For example, a partial ESM for the three screens SC_a, SC_b, and SC_c in Figure 1 is shown in Figure 4. Here Σ includes editing and clicking events on various text fields and buttons, $\boldsymbol{S} = \{\text{SC}_a, \text{SC}_b, \text{SC}_c\}$, SC_a is the initial screen, and each connection is labeled with its ID, the event type, and widget ID.

A non-empty sequence $\epsilon = c_0 c_1 \dots c_n$ ($n \geq 0$, $c_i \in \boldsymbol{C}$, $0 \leq i \leq n$) of connections is called a *path* on ESM $\mathcal{M}$, if the destination of c_j is equal to the source of c_{j+1} for all $0 \leq j < n$. ϵ is called a *run* of the model, denoted as $\epsilon \models \mathcal{M}$, if it starts from the initial screen of $\mathcal{M}$. Runs of a model capture important event sequences that can be triggered on the app's GUI. For example, the sequence of connections $c_1 c_7 c_2$ in ESM_1 forms a run and indicates that, a click on Add note on SC_a will bring a user to SC_b, where multiple editing events are possible without causing any screen transition; A click on Save, however, will bring the user back to SC_a.

The connection between a mobile App and its ESM is straightforward, lowering the bar for model construction. If the model is not available already, the construction of the whole ESM is only necessary when ATOM is applied to an app for the first time. This is because, when using ATOM, the model is incrementally maintained together with the test scripts (see Section III-B) and is suitable for use in the next maintenance.

In our experimental evaluation (see Section IV), we manually created the ESMs for the subject apps. Another viable way is to first use tools like GUI Ripper [7] to build an initial model and then adjust that model to meet the requirement of ATOM. The construction of an automatic ESM extraction tool belongs to the future work.

B. Changes as a Delta-ESM

Many different reasons may cause test scripts to break during the evolution of an app from a base version to a new one. In this work, we focus on cases where the reason is in changes to the GUI of the app. We model the changes by following a similar idea as described in Section III-A and construct a delta ESM (DESM). A DESM specifies all the changes to the connections of an ESM as well as the involved screens.

Given an ESM $\mathcal{M} = <\Sigma, \boldsymbol{S}, \{s_0\}, \boldsymbol{C}, \boldsymbol{F}>$, a delta-ESM $\Delta_{\mathcal{M}}$ relevant to $\mathcal{M}$ is a septuple $<\Sigma_\Delta, \boldsymbol{S}_\Delta, \emptyset, \boldsymbol{C}_\Delta, \emptyset, \boldsymbol{l}, \boldsymbol{r}>$, where $<\Sigma_\Delta, \boldsymbol{S}_\Delta, \emptyset, \boldsymbol{C}_\Delta, \emptyset>$ is also a finite state machine similar to an ESM but with no initial or final screen; $\boldsymbol{C}_\Delta$ is the set of all changed connections with respect to $\boldsymbol{C}$; $\boldsymbol{l} : \boldsymbol{C}_\Delta \rightarrow \mathit{Bool}$ is a total function and it partitions $\boldsymbol{C}_\Delta$ into two groups: the set $\boldsymbol{C}_+$ of connections producing true values are newly introduced by the changes, and the set $\boldsymbol{C}_-$ producing false are those to be removed; A modification to a connection is modeled in a DESM as two *related* changes, one deleting the original connection and the other adding the modified. $\boldsymbol{r} : \boldsymbol{C}_- \rightarrow \boldsymbol{C}_+$ is a partial function, it maps a connection deletion to its related addition, when applicable; Screens associated with at least one of the changed transitions constitute $\boldsymbol{S}_\Delta$. In addition, we denote the set of added and deleted screens as $\boldsymbol{S}_+$ and $\boldsymbol{S}_-$, respectively. Consider for example the changes to NotePad Version 1.0, as described in Section II, they can be depicted by the DESM shown in Figure 5. Edges in thick solid line model added connections, and those in dashed line model deleted ones.

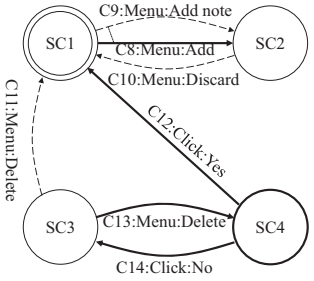


Fig. 5: A delta-ESM modeling the changes to NotePad Version 1.0.

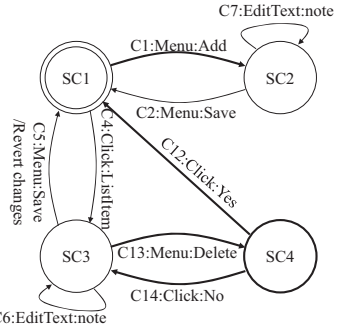


Fig. 6: Partial event sequence models for Notepad versions 2.0 (ESM_2).

An DESM captures changes introduced by the new version, relative to a base version. Such info is valuable in both understanding the impact of the changes on test scripts and devising adjustments to the test scripts to reflect the changes.

Once we have an ESM $\mathcal{M} = <\Sigma, \boldsymbol{S}, \{s_0\}, \boldsymbol{C}, \Sigma>$ modeling the behaviors of the base version app and a DESM

$\Delta_{\mathcal{M}} = <\Sigma_\Delta, \boldsymbol{S}_\Delta, \emptyset, \boldsymbol{C}_\Delta, \emptyset, \boldsymbol{l}, \boldsymbol{r}>$ capturing the changes made to them, we can easily *merge* (we use $\bigoplus$ to denote the operation) the two and construct the ESM for the new version app. The new ESM can be computed as $\mathcal{M} \bigoplus \Delta_{\mathcal{M}} = <\Sigma \cup \Sigma_\Delta, \boldsymbol{S} \cup \boldsymbol{S}_+ - \boldsymbol{S}_-, \{s_0\}, \delta \cup \boldsymbol{C}_+ - \boldsymbol{C}_-, \boldsymbol{S} \cup \boldsymbol{S}_+ - \boldsymbol{S}_->$. Note that in this process, connections from $\boldsymbol{C}_-$ first get their IDs from their matches in $\mathcal{M}$ and then pass on the IDs to their related connections in $\boldsymbol{C}_+$. Therefore, modified connections will preserve their IDs in the new model.

C. Test Scripts and Their Simulations

A test script describes a sequence of actions to be taken to exercise an app during testing. Each action has a type and a target descriptor: the type specifies the nature of the action, e.g. whether it is to click an element or to input some text; the target descriptor describes the widget on which the action is performed. The execution of a test script generates a sequence of events on the GUI of the app, which can be used to simulate its run on the ESM of the app.

In ATOM, the correspondence between ESM events and test script actions is defined in the form of a mapping relation in configuration files. In this way, we can easily use high level events in ESMs to keep the models small. For example, instead of using two low level events of pressing the Menu physical button and pressing on the Save menu item that directly match test script actions, an ESM can use just one high level event clicking the Save menu item. Such design also makes it easy to extend ATOM to handle test scripts in other syntaxes.

Formally, let $\boldsymbol{T}$ be the set of action types and $\boldsymbol{D}$ the set of target descriptors, the set of possible actions is then $\boldsymbol{A} \subseteq \boldsymbol{T} \times \boldsymbol{D}$. Given an ESM $\mathcal{M} = <\Sigma, \boldsymbol{S}, \{s_0\}, \delta, \boldsymbol{F}>$, a test script $\boldsymbol{K} = k_0 k_1 \dots k_h$ $(h \geq 0, k_i \in \boldsymbol{A}$ for $0 \leq i \leq h)$, and a mapping relation $\boldsymbol{R} : \boldsymbol{A} \times \Sigma$, we can easily find a sequence of events $\boldsymbol{R}(\boldsymbol{K})$ on $\mathcal{M}$ by repeatedly applying $\boldsymbol{R}$ to actions in $\boldsymbol{K}$. Using $\boldsymbol{R}(\boldsymbol{K})$ we can derive a run $\boldsymbol{P}$ on $\mathcal{M}$ that models the intended execution of $\boldsymbol{K}$. We call $\boldsymbol{P}$ the simulation of $\boldsymbol{K}$ on $\mathcal{M}$. Nondeterminism in $\mathcal{M}$ can be resolved by human input or knowledge about the actual execution trace of $\boldsymbol{K}$ on the app.

D. Test Script Maintenance

The maintenance of test scripts is done in two phases based on simulations of test scripts. In the first phase, by comparing the simulation of a test script on an ESM and the changes made to that model in a DESM, ATOM first identifies how the changes affect the simulation, then synthesizes a new simulation that is in line with the changes based on the previous one. In the second phase, causes, i.e., events, of the connections from the new simulation are collected and mapped to test script actions according to the mapping relation $\boldsymbol{R}^{-1}$ between GUI events and script actions from Section III-C.

Algorithm 1 shows detailed steps in the first phase. The algorithm takes as input the ESM $\mathcal{M}$ for the base version of app, the DESM Δ of the new version with respect to $\mathcal{M}$, and the simulation $\boldsymbol{P}$ of a test script on $\mathcal{M}$, and outputs the updated simulation $\boldsymbol{P'}$. First, $\mathcal{M}$ and Δ are merged to produce the updated ESM $\mathcal{M}'$ for the new version (line 2). Then, the algorithm iterates through the simulation $\boldsymbol{P}$ and updates every event in order (lines 3 through 50). Particularly, if an event p_i is not affected by the change (line 6), it is appended to the result simulation directly (line 7 through 9); Or, if p_i is modified (line 10), then the modified connection is appended to the result simulation (lines 11 through 13).

Otherwise, p_i is changed in other ways or deleted in the new version, and the paths before and after p_i are now disconnected. The algorithm constructs an alternative path to reconnect them by finding an intermediate state $interState$ that is connected to both paths. Three different cases are considered here. Let e be the cause of p_i. First, if e transits the source of p_i to another state in $\mathcal{M}'$ (line 16), then this new destination state is used as the $interState$, and the algorithm exploits a broad first search to find a path from $interState$ to the original destination of p_i. The search is restricted to paths no longer than $MaxPathLength$ to keep the cost low and the alternative paths easy to understand, and the first hit, if any, will be appended together with the connection between the source of p_i and $interState$ to the result simulation. Then the iteration proceeds to the next connection (lines 17 through 24). Second, if e transits in $\mathcal{M}'$ another $preState$ to p_i's destination (line 25), then the algorithm tries to construct an alternative path connecting the source and destination of p_i through $preState$ in a similar way as described in the previous case (lines 26 through 33); Third, if e is not associated with either the source or destination of p_i, the algorithm searches for a short path connecting p_i's source state with any state from the path starting from p_i's destination, with the hope to preserve as many original connections as possible (lines 35 through 47). If such effort fails, the simulation is truncated (line 48).

IV. EVALUATION

To empirically evaluate the *effectiveness* of ATOM in test scripts maintenance we have conducted experiments that applied ATOM to 11 production mobile apps. This section reports on the experiments and provides preliminary assessment of the approach.

A. ATOM Implementation

In its current implementation, ATOM automatically maintains scripts that are based on the Robot Framework to test the GUI of Android apps. The Robot Framework is a generic, keyword-driven, test automation framework. In ATOM, it uses the Appium[4] open source test automation framework to drive the Android app under testing and it communicates with Appium through the AppiumLibrary. Our approach, however, is not limited to any specific testing framework. Support for other testing frameworks can be easily added by defining the necessary mapping between test script actions and ESM connections.

All the experiments ran on a Windows 8 machine with 3.1 GHz Intel dual-core CPU and 8 GB of memory. ATOM

[4]http://appium.io/

Algorithm 1 Maintaining an ESM Path

Input: $\mathcal{M} = <\Sigma, \boldsymbol{S}, \{s_0\}, \boldsymbol{C}, \boldsymbol{F}>$,
$\Delta_{\mathcal{M}} = <\Sigma_\Delta, \boldsymbol{S}_\Delta, \emptyset, \boldsymbol{C}_+ \cup \boldsymbol{C}_-, \emptyset, \boldsymbol{l}, \boldsymbol{r}>$,
$\boldsymbol{P} = p_0 p_1 \dots p_l$ $(0 \leq l)$ on $\mathcal{M}$
Output: Path $\boldsymbol{P'}$ on $\mathcal{M} \bigoplus \Delta_{\mathcal{M}}$

$\boldsymbol{P'} \leftarrow [\,]$
$\boldsymbol{S'} = \boldsymbol{S} \cup \boldsymbol{S}_\Delta$, $\boldsymbol{C'} \leftarrow \boldsymbol{C} \cup \boldsymbol{C}_+ - \boldsymbol{C}_-$
$i \leftarrow 0$, $srcState \leftarrow s_0$
while $i \leq l$ **do**
  $destState \leftarrow p_i.destination$
  if $p_i \notin \boldsymbol{C}_-$ **then** ▷ p_i not affected
    $\boldsymbol{P'} \leftarrow$ CONCAT$(\boldsymbol{P'}, [p_i])$
    $srcState \leftarrow destState, i \leftarrow i + 1$
    continue
  else if $\boldsymbol{r}(p_i) \neq$ **null then** ▷ p_i modified
    $\boldsymbol{P'} \leftarrow$ CONCAT$(\boldsymbol{P'}, [\boldsymbol{r}(p_i)])$
    $srcState \leftarrow destState, i \leftarrow i + 1$
    continue
  end if
  $e \leftarrow p_i.cause$ ▷ p_i otherwise changed or deleted
  if POSTSTATE$(\mathcal{M}', srcState, e) \neq$ **null then**
    $interState \leftarrow$ POSTSTATE$(\mathcal{M}', srcState, e)$
    $path \leftarrow$ SHORTESTPATH$(\mathcal{M}', interState, destState)$
    if $path \neq$ **null then**
      $c \leftarrow$ CONNECTION$(\mathcal{M}', srcState, e, interState)$
      $\boldsymbol{P'} \leftarrow$ CONCAT(CONCAT$(\boldsymbol{P'}, [c]), path)$
      $srcState \leftarrow destState, i \leftarrow i + 1$
      continue
    end if
  else if PRESTATE$(\mathcal{M}', e, destState) \neq$ **null then**
    $interState \leftarrow$ PRESTATE$(\mathcal{M}', p_i, destState)$
    $path \leftarrow$ SHORTESTPATH$(\mathcal{M}', srcState, interState)$
    if $path \neq$ **null then**
      $c \leftarrow$ CONNECTION$(\mathcal{M}', interState, e, destState)$
      $\boldsymbol{P'} \leftarrow$ CONCAT(CONCAT$(\boldsymbol{P'}, path), [c])$
      $srcState \leftarrow destState, i \leftarrow i + 1$
      continue
    end if
  else
    $hasFound \leftarrow$ **false**
    for $j \leftarrow i + 1, l$ **do**
      $interState \leftarrow p_j.source$
      $path \leftarrow$ SHORTESTPATH$(\mathcal{M}', srcState, interState)$
      if $path \neq$ **null then**
        $\boldsymbol{P'} \leftarrow$ CONCAT$(\boldsymbol{P'}, path)$
        $srcState \leftarrow interState$
        $i \leftarrow j, hasFound \leftarrow$ **true**
        break
      else
        $j \leftarrow j + 1$
      end if
    end for
    if not $hasFound$ **then** $i \leftarrow l + 1$ **end**
  end if
end while

return $\boldsymbol{P'}$
CONCAT$(path_1, path_2)$ ▷ The concatenation of $path_1$ and $path_2$
SHORTESTPATH$(\mathcal{M}, srcState, destState)$ ▷ The shortest path from ▷ $srcState$ to $destState$ on $\mathcal{M}$: shorter than **MaxPathLength**, or **null**
PRESTATE$(\mathcal{M}, e, destState)$ ▷ The state s in $\mathcal{M}$ such that ▷ $<s, e, destState>$ is a connection in $\mathcal{M}$, or **null**
POSTSTATE$(\mathcal{M}, srcState, e)$ ▷ The state s in $\mathcal{M}$ such that ▷ $<srcState, e, s>$ is a connection in $\mathcal{M}$, or **null**
CONNECTION$(\mathcal{M}, srcState, e, destState)$ ▷ The connection ▷ in $\mathcal{M}$ from $srcState$ to $destState$, with cause e
MaxEditDistance $\leftarrow 10$
MaxPathLength $\leftarrow 2$

was the only computationally-intensive process running during the experiments. ATOM spent less than one second to finish updating all the test scripts of each app.

B. Measures

One goal of ATOM is to assist test script maintenance so that the confidence provided by the test scripts in the correctness of the app, in terms of the coverage of screens and connections in ESMs, could be preserved as much as possible after the maintenance. A good coverage of the models by the updated test scripts, however, is not enough by itself. Such scripts, e.g., when produced by an automatic generation process, may exercise very different behaviors of the app than those exercised by the original test scripts, which incorporate valuable knowledge about the application. Therefore, in addition to expecting the updated test scripts to cover comparable percentage of the ESMs as before maintenance, we also envision updated scripts to retain most of the action sequences from the previous test scripts. We adopt two metrics accordingly to measure the effectiveness of the maintenance process.

Formally, let $\boldsymbol{S}_c$ be the set of screens that the updated ESM shares with the base ESM, $\boldsymbol{S}_v$ the set of screens visited by the original test scripts, and $\boldsymbol{S}'_v$ the set of screens visited by the updated test scripts. The *screen coverage preservation* (SCP), calculated as $|\boldsymbol{S}'_v \cap \boldsymbol{S}_c| / |\boldsymbol{S}_v \cap \boldsymbol{S}_c|$, measures, among all the screens shared by the updated ESM and the base ESM, how many percent of previously covered screens are still covered by the tests after maintenance. Similarly, we can define the *connection coverage preservation* (CCP) to measure, among all the connections shared by the updated ESM and the base ESM, how many percent of previously covered connections are still covered by the tests after maintenance.

Let $\boldsymbol{A}_t$ be the set of all test actions from the base version test scripts, $\boldsymbol{A}_e \subseteq \boldsymbol{A}_t$ the set of effective test actions that will be exercised if the base test scripts are run directly on the updated app, and $\boldsymbol{A}'_e$ the set of test actions that are executed by the maintained test scripts. The *test action preservation* (TAP), calculated as $|(\boldsymbol{A}'_e - \boldsymbol{A}_e) \cap \boldsymbol{A}_t| / |\boldsymbol{A}_t - \boldsymbol{A}_e|$, measures, among all the test actions that would be lost if without maintenance, how many percent are now rescued into the updated tests.

Based on the measures, we devise our experiments to answer the following two research questions:

- **RQ1**: Does using ATOM lead to updated test scripts with high screen and connection coverage preservation?
- **RQ2**: Does using ATOM lead to updated test scripts with high test action preservation?

C. Subjects

We select as the experiment subjects 11 popular production mobile apps from a Chinese Android market. Table I lists all the apps and briefly describes each app. Even though we deliberately select apps with various sizes and from different categories, they can hardly represent the wide range of all apps. The selection of subjects presents threats to the generalizability of our results, which we discuss in Section IV-F.

For each app, we randomly pick two adjacent releases that we can download from the market, and treat the earlier release

TABLE I: Description of the mobile apps as subjects.

App (Acronym)	Brief Description
Bilibili (BB)	A video sharing website.
GNotes (GN)	A simple note app.
Wannianli (WN)	A calendar app.
YoudaoNote (YD)	A cloud-based note app.
Wechat Phonebook (PB)	A phonebook app.
Changba (CB)	A Karaoke app.
Baidu Music (BD)	A music player.
365 Calendar (CA)	A calendar app.
Ctrip (TR)	An online travel agent.
WizNote (WZ)	A cloud-based information management app.
TickTick (TT)	A to-do list app.

TABLE IV: Experimental results.

APP	SCP	CCP	TAP
Bilibili	1(7/7)	1(11/11)	0.26(35/134)
GNotes	0.93(14/15)	1(23/23)	0.87(138/159)
Wannianli	1(13/13)	1(36/36)	0.95(91/96)
YoudaoNote	0.94(17/18)	1(31/31)	0.91(116/128)
Wechat Phonebook	0.85(11/13)	0.91(21/23)	0.95(37/39)
Changba	1(9/9)	1.06(19/18)	0.74(97/131)
Baidu Music	1(12/12)	1(25/25)	0.90(79/88)
365 Calendar	0.85(9/11)	0.88(22/25)	0.65(31/48)
Ctrip	0.92(12/13)	0.91(19/21)	0.96(50/52)
WizNote	1(12/12)	1.05(22/21)	0.93(82/88)
TickTick	0.92(11/12)	1(17/17)	0.73(16/22)
Total:	0.94(127/135)	0.98(246/251)	0.78(772/985)

as the base version, while the later one as the updated version. By reading the change log as well as actually playing with the apps, we identify for each app a list of changes to their GUI. We then asked a group of three post-graduate students to manually build an ESM for parts of each app that are affected by the changes. To ensure the correctness of the model, we trained the students using the NotePad app from Section II as an example and asked another student to review the models. In this step, we get a partial ESM for each of the 11 apps. Even though the models are incomplete, they are already useful in test script maintenance, as shown by the results presented in Section IV-E. This also speaks in favor for the usability of ATOM. The same group of students then constructed the DESM for each app based on the changes.

Table II lists, for each app, the base version (Base), and the updated version (Updated). For each ESM of the base version, the number of screens (#S) and connections (#C) in the model are listed; For each DESM, the numbers of added (#S_+) and deleted (#S_-) screens as well as the numbers of added (#C_+), deleted (#C_-), and modified (#C_M) connections are also included in the table. In total, all the apps have 176 screens and 416 connections.

We also asked another group of four post-graduate students to write test scripts for the apps. Table III reports for each app the total number of scripts (#K) as well as the minimum (Min), maximum (Max), average (Avg), and total (Sum) number of actions in those scripts. As a measure of the quality of the test scripts, we also report in the table the coverage of screens (S) and connections (C) by each set of test scripts (Cov). We have created 145 scripts totaling 1837 actions, and they cover 99.52% of the screens and 95.33% of connections in the base ESM.

The focus of ATOM is to help maintain the test scripts that become obsolete after changes happen to an app. To avoid the influence of test scripts that were not impacted by the changes, we first run all the test scripts on the updated apps to identify the ones that need maintenance. Table III lists for each app the number of test scripts affected by the changes (K_a), and the same statistics for the affected script files, including the minimum (Min_a), maximum (Max_a), average (Avg_a), and total (Sum_a) number of actions in a single script. The number of changed test script actions (#Chg) and the breakdown into the numbers of deleted (Del) and modified (Mod) actions are also included in the table. *In the experiment, we consider only the affected scripts*.

D. Experimental Protocol

To get an idea of the extend to which the changes affect the test scripts, we first calculate the screen and connection coverage of the base ESM by the original test scripts. Next, we apply ATOM to automatically maintain the test scripts. The updated test scripts are then compared with the original ones using the two metrics defined in Section IV-B. All the comparisons are done on test scripts both as a whole and individually.

E. Experimental Results

In this section, we report the results of our evaluations, with the aim of answering the research questions listed at the end of Section IV-B.

Table IV summarizes the values of the metrics we use to measure the effectiveness. For each app, the table presents the screen coverage preservation (SCP), the connection coverage preservation (CCP), and the test action preservation (TAP) of the maintenance process. Each entry in column SCP is in form $x(y/z)$, where x is the value of the metric, while $y = |\boldsymbol{S}'_v \cap \boldsymbol{S}_c|$ is the number of shared screens that are covered by the updated scripts and $z = |\boldsymbol{S}_v \cap \boldsymbol{S}_c|$ is the number of shared screens visited by the original scripts. Similar information is listed also for the CCP value of each app. Each entry in column TAP is also in form $x(y/z)$, where x is the value of the metric, while $y = |(\boldsymbol{A}'_e - \boldsymbol{A}_e) \cap \boldsymbol{A}_t|$ is the number of extra test actions rescued by the maintenance process and $z = |\boldsymbol{A}_t - \boldsymbol{A}_e|$ is the number of test actions that would be lost if without maintenance.

RQ1: screen and connection coverage preservation. From Table IV, we can see that ATOM managed to achieve SCP and CCP values higher than 0.80 and 0.9, respectively, for all apps. The overall SCP and CCP are also high: 0.94 and 0.98, respectively. Such results strongly suggest ATOM is effective in preserving the coverage of the ESM during test script maintenance.

Figure 7 plots the distribution of the coverage preservation values as four histograms. The x-axis of Figure 7(a) represents the SCP value of the updated test scripts for an app, and y-axis represents the number of apps producing such values.

TABLE II: Basic information about different versions of the experiment subjects.

APP	Base	Updated	ESM		DESM				
			#S	#C	$\#S_+$	$\#S_-$	$\#C_+$	$\#C_-$	$\#C_M$
Bilibili	4.12.1	4.13.0	15	36	1	5	1	16	1
GNotes	1.0.2	1.0.3	17	36	0	2	7	11	2
Wannianli	4.4.2	4.4.6	13	39	1	0	2	0	10
YoudaoNote	5.1.0	5.2.0	18	43	1	0	7	6	8
Wechat Phonebook	3.5.1	4.2.0	13	26	1	0	4	1	22
Changba	6.7.1	7.0.0	18	68	3	3	12	16	3
Baidu Music	5.6.6.1	5.7.0.3	15	33	0	2	0	5	7
365 Calendar	6.0.2	6.2.3	18	40	2	3	4	5	3
Ctrip	6.15.2	6.16.0	19	35	0	0	0	0	5
WizNote	7.1.0	7.1.6	15	30	0	2	0	4	6
TickTick	2.6.6	2.6.7	15	30	2	0	4	1	4
Total:	–	–	176	416	11	17	41	65	71

TABLE III: Basic statistics for all the test scripts and for affected test scripts only.

APP	#K	#A				Cov		$\#K_a$	$\#A_a$						
		Min	Max	Avg	Sum	S	C		Min_a	Max_a	Avg_a	Sum_a	#Chg	#Del	#Mod
Bilibili	15	3	20	13.93	209	100	100	12	3	20	15.67	188	59	56	3
GNotes	9	7	33	20.89	188	100	91.67	8	7	33	22.38	179	25	10	15
Wannianli	9	8	26	12.89	116	100	92.31	9	8	26	12.89	116	12	0	12
YoudaoNote	12	6	16	10.67	128	100	86.05	12	6	16	10.67	128	35	8	27
Wechat Phonebook	7	3	9	5.86	41	100	92.31	7	3	9	5.86	41	33	1	32
Changba	27	5	21	12.07	325	100	100	14	5	17	12.64	177	41	31	10
Baidu Music	20	3	13	4.7	94	100	100	18	3	13	4.89	88	23	6	17
365 Calendar	8	4	44	16.25	130	100	97.5	4	10	44	22.5	90	14	9	5
Ctrip	12	4	29	18.83	226	94.74	94.29	5	4	29	14.8	74	7	2	5
WizNote	12	15	27	20.67	248	100	96.67	8	19	27	21.5	172	15	4	11
TickTick	14	3	19	9.43	132	100	100	4	7	19	12	48	5	1	4
Total:	145	3	44	12.68	1837	99.52	95.53	101	3	44	14.17	1301	269	128	141

Figure 7(b) shows similar distribution but at the level of individual test scripts. Figures 7(c) and 7(d) are counterparts of Figures 7(a) and 7(b) for CCP values.

A value larger than 1 indicates the coverage is actually increasing after the maintenance. The increase may happen, e.g., when the alternative path found for a deleted connection visits other screens that were not visited by the existing test scripts. Four scripts had very low SCP and CCP values (< 0.1) because many of the screens and connections were removed by the changes.

> ATOM *is effective in achieving high screen and connection coverage preservation when maintaining test scripts.*

RQ2: test action preservation. TAP values in Table IV are greater than 0.6 for all apps except one. A closer look at that app (BB) reveals that most changes between the two versions were deletions (as shown in Table II). In such cases, if ATOM fails to find alternative paths on the model to continue a test script, remaining test actions from that script will be truncated, resulting in a low TAP value. Majority of the individual scripts also have high TAP values (≥ 0.5). Those with low TAP values (< 0.4) are mostly from apps BB and CB, where many connections were removed between versions. For better understandability of the updated tests, we restricted that alternative paths ATOM uses should not be longer than 2. We expect TAP values to be higher if this restriction is relaxed or removed. In general, TAP values indicate ATOM is effective in preserving the test actions that would be lost if without maintenance.

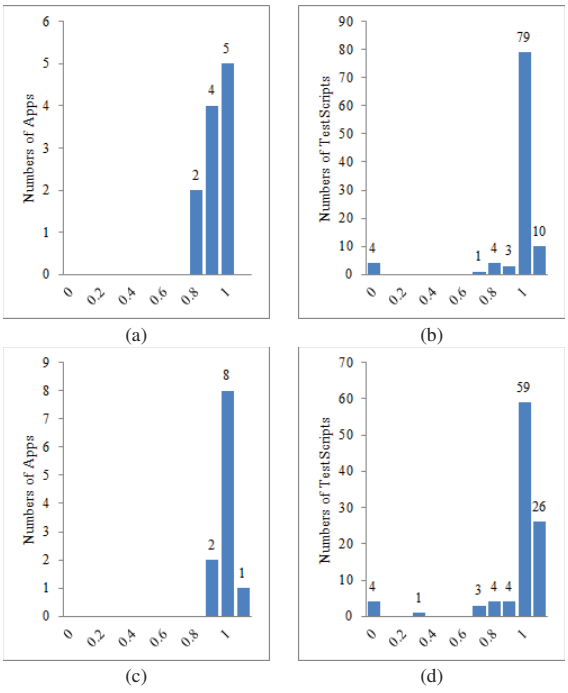
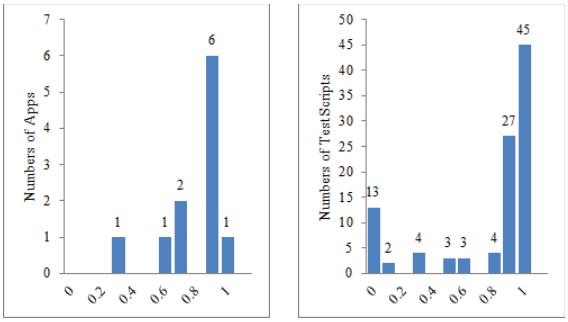


Fig. 7: Distribution of SCP and CCP values for the maintained test scripts.

Figure 8 plots the distribution of TAP values as two his-



(a) (b)

Fig. 8: Distribution of TAP values for the maintained test scripts.

tograms. The x-axis of Figure 8(a) represents the TAP value of the updated test scripts for an app, and y-axis represents the number of apps producing such values. Figure 8(b) shows similar distribution but at the level of individual test scripts.

> ATOM *is effective in achieving high test action preservation when maintaining test scripts.*

F. Threats to Validity

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

Construct: We evaluate the effectiveness of ATOM based on screen and connection coverage preservation and test action preservation after the maintenance. While the overall coverage of the new version app by updated test scripts is also an important metric, we did not use it in this work, as our focus is on the reuse of existing test scripts. Techniques for test script generation can be integrated into our approach to improve the overall code coverage by result test scripts.

Internal: The main threat to internal validity lies in the possible faults in our implementation. We endeavored to minimize such threats. We reviewed our source code and manually checked the generated updates to the original test scripts to make sure ATOM faithfully implements Algorithm 1. We also cross reviewed the models among the post-graduate students to ensure the correctness of the models.

External: The main threat to external validity is the representativeness of our evaluation subjects. Mobile apps used in this study were commercial ones selected from Chinese Android market. On the one hand, such apps are likely better representatives of commercial apps than most open source apps. On the other hand, they may introduce bias to the study, as apps from the global or other local Android app market may follow different patterns in their GUI and HCI design. Due to the closed-source nature of the selected apps, we had to prepare models and test scripts for the subjects by ourselves. Bias may be introduced there. Due to limited time, the evaluation was conducted only on the Android platform. More evaluations using other real-world mobile apps as subjects would help us reduce the threat.

V. RELATED WORK

In this section, we review testing techniques for mobile apps that are closely related to the proposed work: regression testing, change acquisition, modelbased testing, as well as repair and maintenance of test scripts.

A. Regression testing techniques

Different testing approaches have been proposed for various testing goals and testing environments. Gao et al. [8] provide a review of existing testing approaches for mobile applications. During the development and operation process, mobile apps are upgraded or changed even more frequently than those of general software applications delivered for PC or server, which bring new challenges to software testing engineers for mobile applications. Various regression testing techniques for mobile apps have been proposed for ensuring that the changes meet the evolving requirements or fix the previous identified bugs. Rapidly evolving mobile apps may cause existing test scripts to become hardly reusable or outdated for regression testing. To solve this problem, researchers have been working on regression testing techniques which verify old functionalities when modifications occur. For example, Miranda et al. [9] show that 31 regression testing techniques have been proposed in the past 15 years. In essence, such regression testing approaches can be divided into the following categories based on their goals, i.e. for test minimization, selection, prioritization, or optimization. Most of existing methods [10] [11] [12] [13] focus only on one goal. In our work, we try to take comprehensive consideration to maintain test scripts based on changes on mobile apps under test so as to reuse existing test scripts as much as possible in the regression testing.

B. Change acquisition techniques

In regression testing, whether previously created test scripts for a base app could be reused for testing the new version depends on the changes between two versions of the application under test. Identifying GUI changes between the two versions and analyzing their impact on test scripts is essential. So far, researchers have proposed different approaches. Grechanik et al. [1] implement a tool REST to identify changes in three steps: determining the modified GUI objects base on the GUI models, detecting the affected script statements, and analyzing these scripts to determine what other statements are affected as a result of using values computed by the statements that reference modified GUI objects. Raina et al. [14] introduce an automated tool for identifying and testing only the modified parts of a web application. An HTML DOM tree generator is used to generate the DOM tree and the two DOM trees in two versions are compared to locate changes in the new version. A testing tool called GUIdiff [15] runs two versions of the same application and observes the differences in the widget trees of their states. GUIdiff inspired us to search for the different versions of mobile apps in our experiment. Currently, we only focus on changes, such as addition, deletion and modification, to GUI widgets. The changes are represented in the modified model for the app under test.

C. Model-based GUI testing

Traditional manual testing is time-consuming and inefficient. Automated GUI testing techniques are widely explored in the academia, and are widely used in the industry [16]. Model-based testing is one of the supporting means of GUI test automation [17]. GUI Models are employed to represent the behavior of the application under test, which then can be used to automatically generate test cases. However, modeling the GUI-based behavior of an application can be difficult. To settle the issue, GUICC [18] determines equivalence/difference between GUI states to generate GUI graph, and updates the GUI graph based on changes. The common reverse engineering techniques for GUI testing are GUI crawling and GUI ripping [19] [7]. Model-based techniques can also be applied to regression testing. For example, Fourneret et al. [20] present a model-based regression testing technique based on behavioral models in UML/OCL. Experimental results suggest that the approach is efficient. GUI ripper, however, may produce inaccurate models. In our approach, we construct an ESM for the GUI of the original application, represent the changes in a DESM, and use model-based testing approach to facilitate automatic test maintenance.

D. Test script reuse

Test scripts are used to perform automatic regression testing. Generally, test scripts could be manually created or automatically generated with the help of record/reply tools according to specific application under test. Because of the rapid evolving and frequent changing of mobile apps, existing test scripts may become unusable for regression testing. It is mentioned in [21] that more than 74 percent of the test cases become unusable. Test script reuse is one of the most important and difficult problems in regression testing. To address this problem, Memon [5] proposed a model-based approach based on event-flow graphs (EFGs). The approach first selects unusable test script according to the original test scripts and the modified GUI, then repairs these scripts base on four user-defined transformations. Daniel et al. [22] propose a white-box approach that focuses on GUI code. They implemented a smart IDE to record the GUI refactoring actions. Then, the information is used to repair test cases. Wen et al. [23] propose an information retrieval based method LOCUS to locate bugs from software changes. LOCUS only outputs the modification content of the most suspicious file in each suspicious change.

Similar to our work, SITAR [6] repairs unusable low-level test scripts. SITAR uses QTP as the testing tool. The approach has three main steps. First, GUI ripper is used to construct an enhanced EFG with dominating relations between edges. Then, script statements are mapped to the EFG. If a match is not found, then a NULL entry is created. At last, a mapped script that has at least one NULL must be repaired. SITAR can output a sequence of events and repaired check points. Our work needs the original model as input. We choose Appium and Robotframework, which makes the mapping harder but more precise, since they do not have a hierarchical relationship. ATOM does not repair oracle, and our test cases are generated through automatic traversal. Researchers try to repair these unusable test scripts to save cost by reusing existing artifacts while without losing efficiency. Pinto et al. [24] point out that existing repairing methods that focus on assertions have limited practical applicability. All of these approaches motivated us to reuse test scripts for regression testing automatically and realistically. We employ model comparing approaches in [25] to update the modified model according to changes. The updated model are then used to maintain scripts automatically.

VI. Conclusion And Future Work

In this paper, we propose a novel approach, called ATOM, to automatically maintain GUI test scripts of mobile apps for regression testing. ATOM abstracts possible event sequences of a base version of GUI using an event sequence model (ESM) and the changes made to that GUI as a delta ESM (DESM). Given both models as input, ATOM automatically updates test scripts targeting the base version GUI to accommodate the changes. We also applied ATOM to update the GUI test scripts for 11 commercial Android apps and obtained promising results: The updated test scripts achieved high preservation of screen and connection coverages by the original tests and of the test actions from the original test scripts.

As for future work, we consider a few interesting directions. First, ATOM takes an ESM and a DESM as part of the input, while the preparation of such models may still require significant manual effort. To further streamline the maintenance of GUI test scripts, we plan to develop techniques to facilitate or even automate the construction of such models. Second, in its current design, ATOM makes no extra effort to test screens and connections added by a change. It, however, may be desirable to extend existing scripts to test also the new behaviors of the app during maintenance. To achieve this, we will adjust the existing algorithm to actively exercise those new behaviors. Third, to help developers understand whether the updated test scripts capture interesting use cases, we also plan to devise techniques to summarize in natural language what each script actually tests.

Acknowledgment

The paper was supported by the National Key Research and Development Plan (No.2016YFB1000802) and the National Natural Science Foundation of China (No.61632015, 61472179, 61561146394, 61572249). Besides, it was partially supported by the Hong Kong RGC General Research Fund (GRF) PolyU 152703/16E and The Hong Kong Polytechnic University internal fund 1-ZVJ1.

The authors thank anonymous reviewers for their efforts/advice on improving this paper.

References

[1] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving gui-directed test scripts," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 408–418, IEEE Computer Society, 2009.

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

[3] F. Gross, G. Fraser, and A. Zeller, "Exsyst: Search-based gui testing," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 1423–1426, IEEE Press, 2012.

[4] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pp. 429–440, IEEE Computer Society, 2015.

[5] A. M.Memon, "Automatically repairing event sequence-based gui test suites for regression testing," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 4, 2008.

[6] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, "Sitar: Gui test script repair," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 170–186, 2016.

[7] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 258–261, ACM, 2012.

[8] J. Gao, X. Bai, W. T. Tsai, and T. Uehara, "Mobile application testing: A tutorial," *Computer*, vol. 47, no. 2, 2014.

[9] R. H. Rosero, O. S. Gomez, 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. 675–689, 2016.

[10] H.-Y. Hsu and A. Orso, "Mints: A general framework and tool for supporting test-suite minimization," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 419–429, IEEE Computer Society, 2009.

[11] G. M. K. Chu-Ti Lin, Kai-Wei Tang, "Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests," *Information and Software Technology*, vol. 56, no. 10, pp. 1322–1344, 2014.

[12] Q. 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*, pp. 27–28, ACM, 2016.

[13] 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*, pp. 41–47, ACM, 2016.

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

[15] S. Bauersfeld, "Guidiff – a regression testing tool for graphical user interfaces," in *Proceedings of the Sixth International Conference on Software Testing, Verification and Validation*, pp. 499–500, IEEE, 2013.

[16] A. Marques, F. Ramalho, and W. L. Andrade, "Comparing model-based testing with traditional testing strategies: An empirical study," in *Proceedings of the Seventh International Conference on Software Testing, Verification and Validation Workshops*, pp. 264–273, IEEE, 2014.

[17] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: A systematic review," in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, p-p. 31–36, ACM, 2007.

[18] Y.-M. Baek and D.-H. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 238–249, ACM, 2016.

[19] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 260–, IEEE, 2003.

[20] E. Fourneret, J. Cantenot, F. Bouquet, B. Legeard, and J. Botella, "Setgam: Generalized technique for regression testing based on uml/ocl models," in *Proceedings of the Eighth International Conference on Software Security and Reliability*, pp. 147–156, SERE, 2014.

[21] A. M. Memon and M. L. Soffa, "Regression testing of GUIs," *SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 118–127, 2003.

[22] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezzè, "Automated gui refactoring and test script repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pp. 38–41, ACM, 2011.

[23] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 262–273, ACM, 2016.

[24] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 33:1–33:11, ACM, 2012.

[25] Z. Xing and E. Stroulia, "Umldiff: An algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 54–65, ACM, 2005.