TestMig: Migrating GUI Test Cases from iOS to Android

Xue Qin
xue.qin@utsa.edu
University of Texas at San Antonio
TX, USA

Hao Zhong
zhonghao@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Xiaoyin Wang
xiaoyin.wang@utsa.edu
University of Texas at San Antonio
TX, USA

ABSTRACT

Nowadays, Apple iOS and Android are two most popular platforms for mobile applications. To attract more users, many software companies and organizations are migrating their applications from one platform to the other, and besides source files, they also need to migrate their GUI tests. The migration of GUI tests is tedious and difficult to be automated, since two platforms have subtle differences and there are often few or even no migrated GUI tests for learning. To address the problem, in this paper, we propose a novel approach, TestMig, that migrates GUI tests from iOS to Android, without any migrated code samples. Specifically, TestMig first executes the GUI tests of the iOS version, and records their GUI event sequences. Guided by the iOS GUI events, TestMig explores the Android version of the application to generate the corresponding Android event sequences. We conducted an evaluation on five well known mobile applications: 2048, SimpleNote, Wire, Wikipedia, and WordPress. The results show that, on average, TestMig correctly converts 80.2% of recorded iOS UI events to Android UI events and have them successfully executed, and our migrated Android test cases achieve similar statement coverage compared with the original iOS test cases (59.7% vs 60.4%).

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**;

KEYWORDS

Test Migration, Mobile Apps, GUI Testing

ACM Reference format:
Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: Migrating GUI Test Cases from iOS to Android. In *Proceedings of 28th International Symposium on Software Testing and Analysis, Beijing, China, July 2017 (ISSTA'19),* 13 pages.
https://doi.org/10.1145/3092703.3092725

ISSTA'19, July 2017, Beijing, China

ACM ISBN 978-1-4503-5076-1/17/07. . . $15.00
https://doi.org/10.1145/3092703.3092725

1 INTRODUCTION

Nowadays, a lot of mobile software producers are developing their apps for both Apple iOS and Android. Among the top 10 (July 1^{st}, 2017) apps that are not provided by Google [8], 8 apps have their corresponding iOS versions in the Apple Store. The only two exceptions are Clean Master [4] (a system management tool) and Kika Emoji Keyboard [11] (a keyboard app for inputting emotional symbols), which are closely tied to the underlying system. While Android has a market share over 80%, iOS devices and apps are widely reported [2] to have a much higher profit margin. Due to this long-term evenly matched competition between Apple iOS and Android, it is important for software producers to target both platforms for broader user groups.

The iOS and Android versions of an app are often not developed and released simultaneously. Among the 8 top apps mentioned above with both iOS and Android versions, 5 apps (Snapchat [14], Pandora [13], Instagram [9], Crossy Road [5], and WhatsApp [18]) have their iOS versions released first, averagely 9 months before their Android versions are released. The remaining 3 apps (FaceBook [6], FaceBook Messenger [7], and Spotify [15]) have their iOS versions and Android versions released at the same time. FlappyBird [1], one of the most successful mobile application developed by a personal developer, has its iOS version released in May 2013, and its Android version released 7 months later. Some common reasons for the asynchronous development may include the strategical emphasis on users from one platform, the lack of resources or expertise, and limited time.

Due to the above facts, the migration of software cross the two platforms becomes a common and important task in mobile software development, especially from iOS to Android. In the literature, many approaches have been proposed to support cross-platform compilation and execution of apps, such as Cordova [3], and Unity3D [17]. However, cross-platform execution is often too inefficient for real-world usage scenarios [3]. Furthermore, while developers can benefit much from code migration tools, the fully automation of behavior-preserving cross-platform code migration is still far from being practical [49]. Therefore, in the practice of code migration across the two platforms, tedious and error-prone manual effort is still unavoidable, and testing is necessary to ensure the quality of migrated apps.

Automatic test generation [23, 42], although solving a more general problem, suffers from various issues (e.g., how to handle logins and generate valid user inputs), and often cannot achieve sufficient coverage [31]. In this paper, to reduce the testing effort in migrating applications, we propose a novel approach, TestMig, to automatically generate GUI tests for

an application's Android version, when its iOS GUI tests are available. The two **key insights** behind TestMig are: (1) *to facilitate users, iOS and Android versions of the same application typically have similar GUI structure and interaction patterns (also supported by Joorabchi et al.'s study [36]), so event sequences in an application's iOS GUI tests can be largely reused for its Android version*; and (2) *iOS GUI tests may contain valuable knowledge such as testing accounts for login and meaningful user input data, which helps to resolve well-known limitations in automatic test generation techniques.* The idea of migrating GUI tests is general and applies to both directions of test migration, we implement TestMig to migrate iOS GUI tests to Android GUI tests because (1) facts mentioned above show that iOS versions are often developed earlier, and (2) there are more open-source automatic GUI explorers (e.g., UIAutomator [16] and MonkeyRunner [12]) for Android, facilitating the implementation of TestMig. A GUI test case contains two parts: an event sequence and its test oracles. In our paper, we consider the migration of only the event sequences, and leave the migration of test oracles for future work. The major complication of migrating test oracles is that wrongly migrated test oracles may cause false positives in testing, which we will discuss in Section 5. Compared with unit tests, GUI test oracles sometimes require manual inspection, and existing studies [23, 42] also show that some oracles (e.g., a program shall not crash) do not need to be migrated.

The basic design of TestMig contains three components: the iOS test recorder that records the iOS GUI event sequences triggered by iOS GUI tests, the converter that converts iOS GUI events to Android GUI events, and the explorer that explores the Android version under the guidance of the converter. For each iOS test case to be migrated, the converter and explorer will take the test case's recorded iOS event sequence as input. During the exploration of the Android version, at each GUI state, the explorer sends to the converter a list of Android GUI events that can be triggered at the state, and the converter will tell explorer which event to trigger based on the remaining iOS GUI events in the iOS event sequence. Although conceptually TestMig just translates event sequences from iOS to Android, the test migration process faces the following two major technical challenges.

TC1: GUI design changes. Although application versions on different platforms shall have similar functionalities, their GUIs often have subtle differences, since programmers often change their applications (e.g., replacing tabs in iOS to action bars in Android) to satisfy users' habits. As a result, some mappings between GUI events are not one-to-one (e.g., as shown in Figure 1, iOS users need just to tap once on the tab "details", when they fetch the product detail, but Android users need to tap on the action bar and then tap the item "details" from the drop-down list). In such cases, our converter needs to consider all different compositions of follow-up GUI events to decide the correct event to trigger.

TC2: mapping of GUI controls. To migrate test cases from one platform to the other, mappings between GUI controls in two application versions are necessary but unavailable. Traditional code migration techniques [49, 59, 60] between platforms and languages must collect many cross-platform applications as their training data, when they mine mappings between API methods. Although migrated applications present instances for API mappings, they rarely present the mappings of GUI controls, which are required to migrate GUI tests. As a result, we have to propose an approach that does not rely on migrated code.

To overcome **TC1**, TestMig's converter leverages the sequence transduction technique [47] during the guidance of exploration. Based on a converting probability matrix (called *transduction model*) between elements (called *words*) in two domains, the sequence transduction technique synthesizes the optimized sequence in the target domain with overall highest converting probability, while considering many-to-many mapping up to the maximal size of *word*. In our application scenario, we can deem iOS GUI events and Android GUI events as two domains, and the remaining iOS GUI event sequences as element sequences. However, the transduction model still relies on predefined conversion probability between UI events, typically acquired through training which is infeasible as mentioned in **TC2**. To overcome **TC2**, TestMig uses similarity between labels of GUI controls to estimate converting probability between GUI events. To sum up, this paper makes the following main contributions.

- A study of the mapping between the GUI structures and events of iOS and Android.
- A novel approach to automate the migration from iOS GUI tests to Android GUI tests.
- An evaluation on five popular open source mobile applications that have both iOS and Android versions. The results show that on average our approach is able to successfully migrate 80.2% of the recorded iOS UI events to Android UI events, and our migrated test cases achieved a similar test coverage as the original test cases (59.7% vs 60.4%).

2 MOTIVATION

As the two most popular mobile platforms, the UI frameworks of Android and iOS share lots of common features. First, although referred to with different names, both UI frameworks have a three-level GUI hierarchy: a screen containing various GUI controls that users can interact with (called a scene in iOS, and an activity in Android), a group of GUI controls forming a functional area in the window (called a `UIView` in iOS, and a `ViewGroup` in Android), a basic GUI control (called a `UIControl` in iOS, and a `View` in Android). Here, in the framework design, the two lower levels of UI elements are instances of the same abstract class, *i.e.*, `UIControl` is a subclass of `UIView` in iOS, and `ViewGroup` is a subclass of `View` in Android. Second, iOS and Android UI frameworks share a similar group of basic GUI controls (*e.g.*, check boxes in Android and switches in iOS), and mobile style UI controls (*e.g.*, date pickers, and sliders). Although naming is slightly different, it does not need much manual effort to construct a mapping between the GUI control types of two frameworks

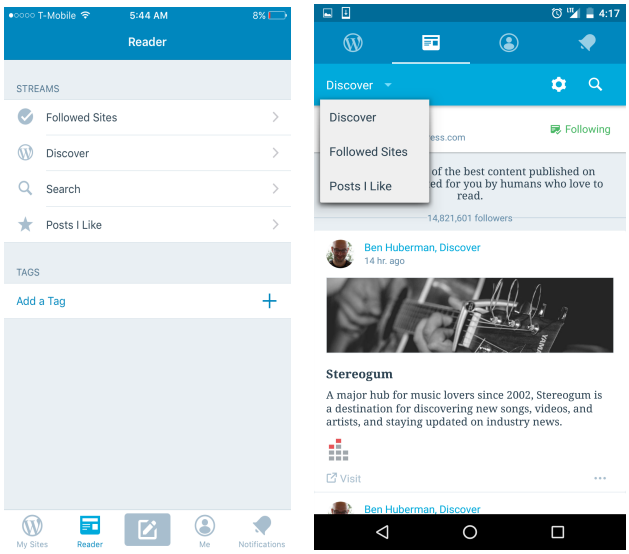

Figure 1: GUI difference between iOS and Android

(i.e., 64 types in Android and 36 types in iOS). As most UI frameworks follow similar patterns, such a mapping can also be easily inferred manually if we want to generalize our technique to other pairs of GUI frameworks.

Despite the commonality in the high level GUI model, there are delicate differences between iOS and Android GUI frameworks. First, Android supports the global return key, and supports a context menu once with the global menu key and now with the menu key in the action bar. By contrast, these keys are not supported in iOS, so iOS apps tend to define and use their own UI controls to navigate backward, and design their own UI views for context menus. Second, iOS apps and Android apps have different UI design styles. For example, while iOS apps typically use tabs at the bottom of the screen to switch between different views in one screen, Android apps often use a drop-down menu at the top corners. Note that, developers have the motivation to adapt their UI design to target platform style, so that their app can fit well with the user's habit in that platform.

Figure 1 shows two screen shots from the iOS version (left) and the Android version (right) of WordPress. The two screens are both reached by starting the app, and clicking on the "reader" button (the second icon from left in the bottom tabs of the iOS version, and the second icon from left in the top tabs of the Android version). However, they look very different. For the reading feature, the iOS version organizes the four sub-features ("followed sites", "discover", "search", "posts I like") in a list of views, and the user can click on the specific item in the list to reach the sub-feature. By contrast, the Android version directly shows the sub-feature of discover, and the user can switch to other sub-features by clicking on the items in the drop-down list (top-left corner of the screen). Furthermore, the iOS version has a "Add a Tag" feature which is not supported by the Android version.

As a result, if an iOS user wants to reach the "discover" sub-feature, she needs to trigger two events: clicking on the "reader" icon, and clicking on the list item "Discover", but an Android user needs to trigger only one event: clicking on the "reader" icon. However, for the other three features ("followed sites", "search", and "posts I like "), an iOS user still just needs to trigger two events, but an Android user needs to trigger three events: clicking on the "reader" icon, clicking on the drop-down menu in the top-left, and click on the specific menu item. From the two screen shots, we have the following observations.

- The iOS and Android versions of an app have similar features and sub-features, and they organize features in a similar way. Therefore, it is feasible to transform an iOS event trace to an Android event sequence.
- The GUI views / controls correspond to the same feature / sub-feature are very similar. For example, the "reader" icons in both versions look the same, and the list items / menu items corresponding to the four sub-features also have the same label.
- Due to various issues, to fulfill a certain task, it can take different steps in one version compared to the other. Therefore, although the mappings of the GUI views and controls are often one-to-one, the mappings between GUI events are typically many-to-many.

Based on the above observations, TestMig records the event trace in the iOS version, constructs mappings between GUI controls, and uses sequence transduction to construct the many-to-many mapping on GUI events.

3 APPROACH

In this section, we introduce the design and structure of TestMig, with its overview presented in Figure 2. The two inputs of TestMig are the iOS app with GUI tests, and the Android version of the apps. The output of TestMig is migrated Android GUI tests. The migration process consists of the following three phases. In the first phase (recording), TestMig records the iOS UI event traces during the execution of iOS GUI tests. In the second phase (exploration), for each recorded iOS UI event trace *tce*, TestMig explores the Android version of the app with the guidance of sequence converter which takes *tce* as its input. In the third phase (generation), TestMig leverages Android Studio to generate Android GUI tests from the Android GUI event trace performed during exploration. The first and third phases are based on existing tools and are thus straightforward, so the core of TestMig is the exploration phase.

At the beginning of the exploration phase, the explorer starts the app. Then at each GUI state, the explorer analyzes the runtime GUI hierarchy to collect a set of available GUI events and send them to the sequence converter. Then, the sequence converter refers to the available GUI events, the iOS GUI event trace, and a static event flow graph to determine which event should be triggered next. Note that the static event flow graph is generated by GATOR [53] to help TestMig find GUI events available in future for many-to-many event

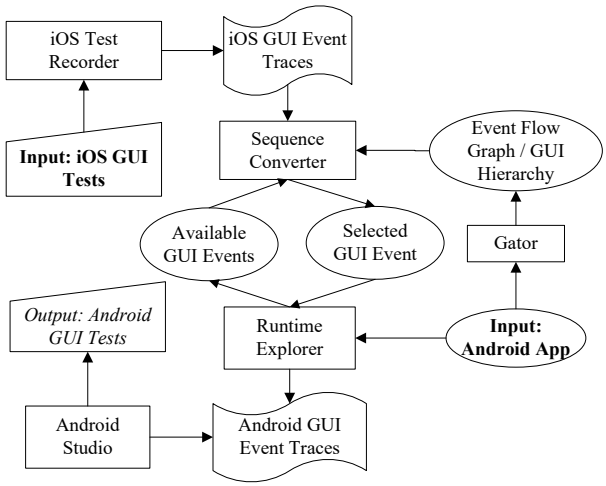

Figure 2: Approach Overview

mapping. The nodes in the event flow graph are windows (e.g., activities and dialog windows), and the edges are GUI events causing transitions between nodes. The sequence converter also needs to determine whether zero, one or multiple events should be consumed from the head of iOS GUI event trace. This process is iteratively performed until (1) the iOS GUI event trace is consumed up, or (2) the exploration cannot continue (i.e., the explorer goes outside the app or no GUI events are available at the current GUI state). The sequence converter is the essential component of TestMig. At each GUI state, its basic event selection mechanism synthesizes an Android GUI event, which is the most similar one to the prefix of the remaining iOS GUI event sequence. However, as described in the motivation example, the mapping of events is often not one-to-one. To handle the problem, TestMig considers more than one event that appears at the top of the remaining iOS GUi event sequences, and looks further into more than one Android GUI event that can be triggered after the next event. We refer to the number of events to be considered in one mapping as the *transduction length* w.

Algorithm 1 Exploration Algorithm

```
1:  procedure CONVERT(I, G, w)
2:      Explorer.startApp()
3:      while I is not Empty do
4:          E ← Explorer.getAvailableEvents()
5:          evt, I' ← SELECTEVENT(I, G, E, w)    ▷ Select
    the best event evt, I' is the remaining iOS event trace
    after evt is triggered
6:          if evt is NULL then
7:              break    ▷ Stop Conversion if cannot proceed
8:          end if
9:          G' ← Explorer.trigger(evt)
10:         G ← G', I ← I'
11:     end while
12: end procedure
```

3.1 Runtime Exploration

Our exploration algorithm is presented in Algorithm 1. The three inputs are I, the recorded iOS event trace, G, the static event flow graph generated by GATOR, and w, the transduction length. At the beginning of the conversion, the explorer will start the app (Line 2). Here, "Explorer" refers to our explorer component, which uses UIAutomator [16] to communicate with the Android app. Then, while I is not consumed up, the explorer will fetch the set of available GUI events at current GUI state (Line 4), and select a best event to trigger with the $SELECTEVENT$ procedure (Line 5). $SELECTEVENT$ will also return the remaining iOS GUI event sequence I' by removing the mapped iOS GUI events from the head of I. Then TestMig will trigger evt. Note that when triggering evt, TestMig is able to locate the actual set of available events after evt, so it will refine the estimated event flow graph G to G' based on the information to remove false positives in G (Lines 9 and 10). In addition, it will update I as I' at the end of the iteration (Line 10), and this process consumes a subsequence of the iOS GUI event sequence. In any case when $SELECTEVENT$ returns NULL (i.e., when its exploration goes to a dead end or outside the app, the explorer will return an empty set and $SELECTEVENT$ will return NULL). When it happens, TestMig stops the conversion and reports the migrated sequences (Line 7).

3.2 Event Selection

We present the detailed process of our event selection (i.e., the $SELECTEVENT$ procedure) in Algorithm 2. Our basic idea is to select a pair of iOS GUI event sequence $prefix$ and Android GUI event sequence $path$ (within transduction length w) that has the highest similarity (Lines 10-19). Note that we calculate the similarity based on the transduction probability in sequence transduction, which will be introduced in Section 3.3. When calculating the transduction probability (Line 13), we will also retrieve the mappings between events. Then, we return the first event $first$ in the selected sequence $path$, and the updated iOS GUI event sequence I with events mapped to $first$ removed (Lines 16 and 20). Before calculating the similarity, TestMig first collects in P all event sequences in G starting with any event in the available event set E within transduction length w (Lines 5-8). It then collects in H all prefixes of the remaining iOS GUI event sequence I within transduction length w (Line 9). Note that because Gator does not handle fragments now, it may also have some false negatives (some actually trigger-able events are missing from G), and some events in E may not exist in G. To make sure we consider all events in E, we add the full set of E into P when initializing it (Line 5). Finally, note that Algorithms 1 and 2 are both conceptual descriptions for clarity. The performance optimization in implementation is not reflected. For example, TestMig stores calculated transduction probabilities so that they are not re-calculated in future event selections.

3.3 Transduction Probabilities between GUI Event Sequences

In the event selection procedure, we need to calculate the similarity (i.e., the transduction probability) from an arbitrary iOS GUI event sequence to an arbitrary Android GUI event sequence (Line 13 of Algorithm 2). TestMig defines the similarity matrix of the one-to-one mappings between iOS GUI events and Android GUI events, and then calculates the transduction probability between event sequences based on the similarity matrix.

3.3.1 Similarity between GUI Events. **Interchangeable UI-Control categories**. TestMig uses the similarity between an iOS UI control (C_{iOS}) and an Android UI control (C_{droid}) to denote the probability of converting an event on C_{iOS} to the corresponding event on C_{droid}. For example, the similarity between a button B_{iOS} and a menu item M_{droid} is used to denote the probability of converting a tap event on B_{iOS} to a tap event on M_{droid}. To make sure a specific event appears on both UI-controls, TestMig restricts that the UI-control mapping between UI controls shall accept the same set of UI events. For example, it is unreasonable to map a text box to a button, because the button does not accept input texts.

For the restriction purpose, we define *Interchangeable UI-Control categories* as a collection of UI controls which accept the same set of UI events. For example, a button and a menu item belongs to the same category, because they both accept the tap event (also the long tap event). Specifically, TestMig considers the following four interchangeable UI-control categories: *editables* which include text boxes, date pickers, and number pickers; *clickables* which include buttons, and menu items; *selectables* including check boxes, and drop down lists; and *swipables* which include swipe views, and sliders. We calculate the similarity only for UI controls within an interchangeable UI-control category, so if two UI controls belong to two different categories, their conversion probability is set as 0. It should be noted that, to make our transduction model flexible enough for UI design changes, we make the interchangable UI-control categories rather general.

Similarity Calculation. For each iOS UI control, TestMig calculates its similarity with all the android UI controls in the same interchangeable UI-Control category. In particular, TestMig uses **UI control attributes**, which include the ID, label, and file names of image resources, to generate features for similarity calculation. As most IDs, titles and file names are written in camel names, to make the calculation more robust, TestMig splits all IDs, titles and file names by non-alpha-numeral letters, and at capital letters to generate a list of tokens. Then, its uses these tokens instead of the original value of the IDs, titles and file names as features. To differentiate tokens from different information sources, it adds a header to each token to indicate the token source. For example, if the ID of a UI control is "inputName", the string is split to three features "ID:input" and "ID:Name", in which the header "ID:" indicates that the three tokens are from the ID attribute (not title or file name). After features are generated, TestMig uses the standard tf-idf formula [19] to weight each feature, where tf is the appearance frequency of the feature in the attribute, and idf is the inverted document frequency of the feature (here we consider all UI control attributes as the whole set of documents). The similarity is calculated with cosine similarity formula based on the weights.

Algorithm 2 Event Selection Algorithm

```
1: procedure SelectEvent(I, G, E, w)                    ▷
   I is the remaining iOS GUI event sequence, G and w's
   meanings are the same as in Algorithm 1, and E is the
   set of available events
2:     if E is empty then
3:         return (NULL,NULL)
4:     end if
5:     P ← E
6:     for Each e ∈ E do
7:         P ← P ∪ GETPATHS(G, e, w)    ▷ GETPATHS
   fetches all paths in G starting with e with length up to
   w
8:     end for
9:     prefix ← HEAD(I, w)    ▷ HEAD fetches I's prefix
   with length up to w
10:    max ← 0
11:    evt_remain ← NULL
12:    for Each path ∈ P do
13:        map, prob ← TRANSPROB(path, prefix)
14:        if prob > max then
15:            first ← HEAD(path, 1)
16:            evt_remain ← (first, I − map[first])
17:            max ← prob
18:        end if
19:    end for
20:    return evt_remain
```

Empty UI Control. An iOS UI event may not be converted to an Android UI event and vice versa. As described in Figure 1, this may happen due to different user habits in iOS and Android. To allow n-to-m mappings (e.g., mapping a sequence of three iOS GUI events to a sequence of two Android GUI events), TestMig introduces a special empty UI control (for both iOS and Android) that can be mapped with any of the four interchangable UI-control categories. However, the empty UI control allows only an empty UI event χ, and the similarity between a UI event and the empty UI event is defined as a constant p_χ. To minimize the side effect of the empty UI event, we set the similarity from any UI event to the empty UI event as the minimum positive similarity among all UI controls.

3.3.2 Sequence Transduction Probability. After calculating similarities between single GUI events, TestMig calculates the transduction probability between sequences as their similarity, with the probabilistic sequence transduction. Probabilistic sequence transduction [47] is a model that automatically translates an element sequence from one element space to the

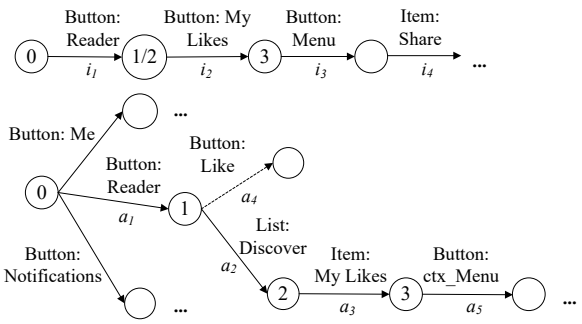

Figure 3: Selection of the next GUI Action

other. It is widely used in machine translation, speech recognition, bioinformatics and other applications. Consider two spaces of elements $E = \{e_1, e_2, ..., e_M\}$, $F = \{f_1, f_2, ..., f_N\}$, and a sequence S in space E ($s = s_1 s_2 ... s_K, s_i \in E$). A sequence transducer converts S to a sequence in space F by finding the sequence $T = t_1 t_2 ... t_L, T_i \in F$ with the highest conversion probability $p(S|T)$. Specifically, the probability $p(S|T)$ can be recursively calculated with the formulas below.

$$P(\epsilon|\epsilon) = 1 \tag{1}$$

$$P(S_{1,K}|T_{1,L}) = (Max_{i=L-W}^{L} P(S_{1,K-1}|T_{1,i}) \times P(s_K|T_{i+1,L}))^{1/Max(K,L)} \tag{2}$$

Here, we use $S_{i,j}$ to denote the subsequence of any sequence S from the i^{th} element to the j^{th} element. When $i = j$, $S_{i,j}$ denotes a single element s_i, and when $i = j + 1$, $S_{i,j}$ denotes an empty sequence ϵ. W is the maximal number of elements in F that a single element in E can be converted to (i.e., the transduction length w). The basic idea of the Formula 2 is to split sequence T into two parts in different ways and maximize the probability of converting the first $K-1$ elements of S to the first part of T and converting the K^{th} element of S to the second part of T. The former probability can be calculated recursively using Formula 2, until the trivial case in Formula 1 is reached. Finally the value is normalized by the length of sequences mapped (i.e., $Max(K, L)$), so that the formula does not bias to shorter mappings. For $P(s_K|T_{i+1,L})$, we calculate it by inserting empty GUI events χ before and after s_K to map the length of $T_{i+1,L}$.

3.4 Running Example

This subsection describes how TestMig performs sequence conversion, with a running example, i.e., the exploration of WordPress's GUI as shown in Figure 1. Figure 3 shows a part of a recorded iOS GUI event sequence I (top part) and the corresponding part of the extracted event flow graph G (bottom part). Here we assume the transduction length is 2.

At the beginning of sequence conversion, the remaining iOS GUI event sequence is the whole sequence, and the explorer reaches state 0 of the event flow graph after starting the app. Then the explorer returns three available events `Button:Me`, `Button:Reader`, and `Button: Notifications`. After that, the event selection module crops the event graph G to get the paths within length 2 in G that starts with one of the three events. In particular, we get a_1, a_1a_4, and a_1a_2 starting with event `Button:Reader`. Note that we do not list all the paths in the figure due to space limit. When calculating the transduction probability with length-2 $prefix$ i_1i_2, we can see that the pair $i_1i_2 \rightarrow a_1a_4$ has the highest probability. So a_1 is triggered, and its mapped iOS GUI event i_1 is consumed. TestMig reaches state 1.

TestMig tries to convert i_2i_3 from state 1, and the cropped subgraph from G includes a_2, a_2a_3, but not a_4 because a_4 is a false positive of Gator and is automatically removed as TestMig finds it to be not trigger-able at state 1. Note that if Gator does not report a_4 as false positive, TestMig still gets state 1 because the pair $i_1i_2 \rightarrow a_1\chi$ has the second highest transduction at state 0. Recall that the empty event χ can be mapped with any event during sequence transduction. During sequence transduction, although i_2 cannot be mapped to a_2, but the sequence $\chi i_2 i_3$ can be mapped to $a_2a_3\chi$, with i_2 mapped to a_3. So a_2 is triggered but no iOS GUI event is consumed as a_2 is mapped to χ, and TestMig reaches state 2. From state 2, pair $i_2i_3 \rightarrow a_3a_5$ has the highest transduction probability so a_3 is triggered and i_2 is consumed.

To show the power of TestMig on selecting the correct path, let us assume that a_4 is not a false positive but a trigger-able event. In such a case, at state 1, a_4 has a higher similarity with i_2. However, since we perform sequence transduction, the probability of pair $\chi i_2 i_3 \rightarrow a_2a_3\chi$ is $P_\chi^{2/3} \approx 0.13$. This value is comparable with the probability of pair $i_2i_3 \rightarrow a_4\chi$, which is $(P_\chi \times P(\text{“Posts I Like”}, \text{“Like”}))^{1/2} \approx 0.17$, although we will still fail to choose the correct path. However, when the transduction length becomes 3, the probability of pair $\chi i_2 i_3 i_4 \rightarrow a_2a_3a_4\chi$ will be $(P_\chi^2 \times P(\text{“ctx\_Menu”}, \text{“Menu”}))^{1/4} \approx 0.20$. This is higher than the probability of pair $i_2i_3i_4 \rightarrow a_4\chi\chi$, which is $(P_\chi^2 \times P(\text{“Posts I Like”}, \text{“Like”}))^{1/3} \approx 0.11$. So the correct path $a_2a_3a_4$ is chosen.

4 EMPIRICAL EVALUATION

To evaluate our approach, we conducted an empirical evaluation on five open source mobile software projects. Specifically, we try to answer the following three research questions:

- **RQ1:** How effective is our approach on migrating UI test cases?
- **RQ2:** How does the parameter of our approach, i.e., the transduction length (w), influence the effectiveness of our approach?
- **RQ3:** Why does our approach fail to migrate some UI test cases?

4.1 Study Setup

In our evaluation, we used five popular open source applications which have both iOS and Android versions.[1] WordPress has a small GUI test suite with only 5 test cases. Wire has a GUI test suite but it is not in its code base and the developers do not want to make it open. Therefore, we manually

[1] All the source code and GUI tests we used in our evaluation are packaged and available at the anonymized project website https://sites.google.com/site/testmigicse2019.

Table 1: Evaluation Subjects

Project Name	Domain	Size (LOC)	#Test Cases	#GUI Events	iOS	Android
2048	Game	1.9K	5	59	30a0f15	0fd6786
SimpleNote	Notepad	17.9K	6	130	4.4.2	1.5.7
Wikipeia	Knowledge media	59.5K	42	475	5.6.0	2.6.203
Wire	Communication	95.7K	32	316	2.41	2.41.359
WordPress	Personal blog	115.3K (80.8K)	31	431	8.3	8.3

enriched the test suite of WordPress, and prepared a test suite for Wire. Please note that it is not easy to find usable experimental subjects because most popular mobile applications are not open source. Even if an application is open source, it may not have GUI (and corresponding GUI tests) and may use test libraries that are close-sourced. Although we cannot select them as subjects, close-sourced apps can use TestMig in their development, since their source files are available in their own programming contexts. Therefore, the requirement of open-source projects is a difficulty only for evaluation, not for developers to adopt our approach.

The basic information of the projects are presented in Table 1. In Columns 1-5 of the table, we present the subject's name, domain, size (Lines Of Code), number of GUI test cases, and number of GUI events triggered in iOS GUI test execution, respectively. From the table, we found that these subjects cover five different major app categories, with their iOS versions' sizes ranging from 1.9k to 95.7K lines of code. For WordPress, a portion of the app's GUI are written with web views, and are thus not covered by the original iOS GUI tests. Therefore, when calculating coverage, we excluded the code portion only reachable from web views (based on a conservative call-graph), and considered only 80.8K lines of code. In Columns 6 and 7, we present the iOS version (commit ID) and Android version (commit iD) we used in our evaluation. When choosing Android versions, we always use the stable Android version (if there exists one) or commit ID after the iOS version within smallest time gap.

During our evaluation, for each subject, we compiled the iOS version in XCode with iOS 10.3 and Swift 3, and executed GUI tests on an iPhone 7 simulator to acquire logs containing GUI event sequences. Then, we used TestMig to explore the corresponding Android app according to each GUI event sequence (each GUI event sequence corresponds to a test case). For each generated Android test case, we executed it and manually examined whether it performed the same GUI interaction as the iOS test case from which it was migrated. A test case is considered successfully migrated if all UI events in the iOS test case are correctly mapped to android UI events (including correctly mapping an iOS UI event to an empty UI event), and all mapped android UI events are successfully invoked. In the manual inspection, we have two students watch executions of both the iOS test case and the migrated Android test case, and report at which GUI event the two executions start to differ. When different events are reported, the execution will be watched again and discussed for the final decision. To reduce the inspection effort, when multiple transduction lengths are observed, we start from the longest transduction length (w=5), because it is supposed to have the highest migration rate. Then we label the triggered Android GUI events in the migrated test cases with correct or wrong. The other tests (w¡5) are compared with labeled events and are only manually checked if they trigger a different event where the reference test case triggers a wrong event.

4.2 Measurements

To answer question RQ1, we need to develop a number of measurements to measure the effectiveness of the test case migration. The most straightforward measurement is the *Test Migration Rate (TMR)*, which measures what proportion of iOS test cases are successfully migrated.

TMR considers only fully successfully migrated tests. In reality, if a test case is partially migrated, it may still cover some code and find some bugs. We consider an iOS test case to be partially migrated if the first k (k larger than 0) iOS UI events are correctly mapped to Android UI events and are correctly executed. For a given iOS test t as a GUI event sequence, we use $partial(t)$ to denote the length of longest prefix of t that are correctly mapped and executed, and $length(t)$ to denote the length of t. We then calculate the migrated proportion of test t ($prop(t)$) as $partial(t)/length(t)$, and define the *Partial Migration Rate (PMR)* of a set of test cases as the arithmetic average of $prop(t)$ for all test case t in the test suite. Finally, we define PMR of a test suite $T = t_1, t_2, ..., t_n$ in formula below:

$$PMR = \frac{\sum_{i=1}^{n} prop(t)}{n} \quad (3)$$

From the formula, we found that PMR deems all test cases with equal weights. As test cases with longer GUI event sequences may provide more code coverage, a migration successful rate on GUI events can be helpful. We further defined *Event Migration Rate (EMR)* of a test suite as follow:

$$EMR = \frac{\sum_{i=1}^{n} partial(t_i)}{\sum_{i=1}^{n} length(t_i)} \quad (4)$$

Since the ultimate goal of the test case migration is to cover the code of the Android version, we use the test coverage of the generated Android test cases as a side measurement for the effectiveness of our approach. In particular, we use statement coverage (reported by Jacoco [10]) as XCode also reports statement coverage of the original iOS tests so we are able to compare the coverage values.

Table 2: Migrated Android test cases

Project	Size (LOC)	#PMT	#Event
2048	1.7K	5	54
SimpleNote	9.3K	6	88
Wikipeia	73.2K	42	433
Wire	71.2K	32	240
WordPress	109.2K (77.5K)	31	404

4.3 Migration Effectiveness

The basic information of the migrated Android test cases is presented in Table 2. Columns 1-4 provides the subject's name, the size of Android version, the number of partially migrated tests (#PMT), and the number of GUI events invoked in the migrated tests, respectively. Comparing the table with Table 1, we have the following observations. First, the Android versions of 2048, Wikipedia, and WordPress have similar sizes with their iOS versions, but the Android versions of SimpleNote and Wire are much smaller than their iOS versions. This reflects that some features may be missing in their Android versions, compared with their iOS versions. This may have impact in the test migration results as we will introduce later. Second, TestMig is able to fully or partially migrate all iOS GUI tests, which shows that the iOS and the Android versions of the five subject apps all have very similar main activities. Third, the migrated tests trigger less events than original tests as some tests are not fully migrated. Note that the numbers in Column 4 cannot be directly used for calculating EMR, as one iOS events can be translated to 0 to W (longest transduction length) Android GUI events.

Our evaluation results on migration rates are presented in Figure 4. In the figure, for each subject, the three columns from left to right represent the TMR, PMR, and EMR of the project, respectively.

From the figure, we found that, in 2048, Wikipedia, and WordPress, TestMig achieves over 80% on all three migration rates: TMR values are 80.0%, 83.3%, and 83.9%; PMR values are 88.0%, 91.2%, and 89.1%; and EMR values are 89.8%, 89.3%, and 87.5%. Generally, if an EMR value of a project is more than its PMR value, it denotes that TestMig achieves better results on longer test cases, and vice versa. For SimpleNote and Wire, our approach has lower migration rates with 66.7% and 65.6% TMR, 73.5% and 69.6% PMR, and 64.6% and 69.9% EMR. We found that the Android versions of SimpleNote and Wire are still under development, and some modules are not migrated from their iOS versions. As TestMig cannot migrate a test case that invokes a non-existing modules, it achieves relatively low PMR and EMR values. However, this actually does not hurt the effectiveness of migrated tests, and non-existing moduless do not need to be tested. In Figure 5, we present an exemplar missing modules in SimpleNote, where the "Tag Edit" button appears in the iOS version (the left screenshot), but does not appear in the Android version (the right screenshot).

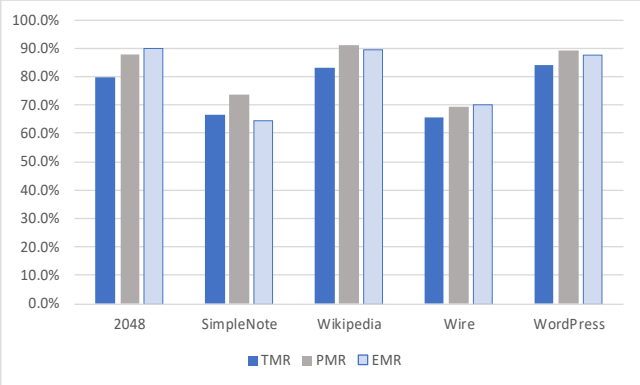

Figure 4: The overall TMR, PMR, and EMR

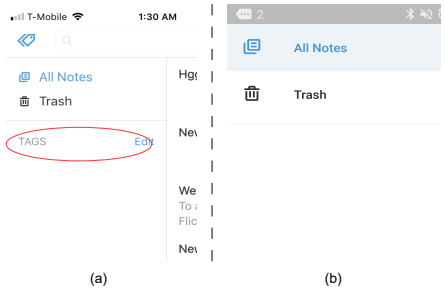

Figure 5: Example of a missing feature

The test coverage of the migrated Android tests are presented in Figure 6. In the figure, for each subject, the left column represents the test coverage of the iOS test cases on the iOS version, and the right column represents the test coverage of the migrated test cases on the Android version.

From the figure, we found that, for the three apps such as 2048, Wikipedia, and WordPress, our migrated Android tests achieved almost the same coverage as their iOS tests. In particular, the migrated tests achieve test coverage of 72.3%, 54.2%, and 55.4%, compared with the test coverage of 75.6%, 62.0%, and 59.6% for the original iOS tests. An interest finding is that in SimpleNote and Wire, which have lower migration rates, our migrated test cases achieve even higher coverage than the original iOS tests. As we mentioned earlier, the Android versions of SimpleNote and Wire are still under development, so it is easier to achieve a higher code coverage as their total code sizes are smaller. By contrast, Wikipedia has a larger Android version than iOS version, so the achieved test coverage is relatively low despite the high migration rates. However, we believe that the cases of Wire and SimpleNote are more common for real-world migration scenarios, as the migrated versions tends to have fewer features than the original versions, at least during or shortly after the migration, when test migration is mostly required.

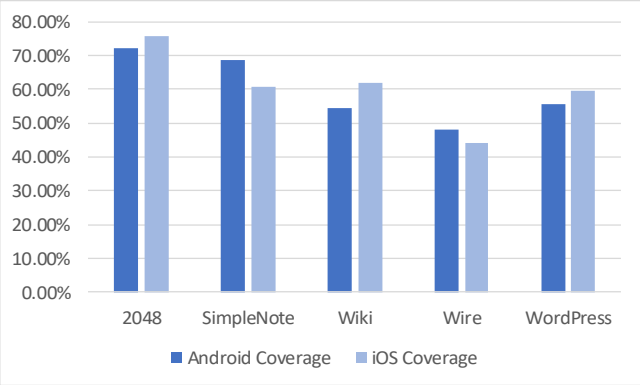

Figure 6: Statement Coverage

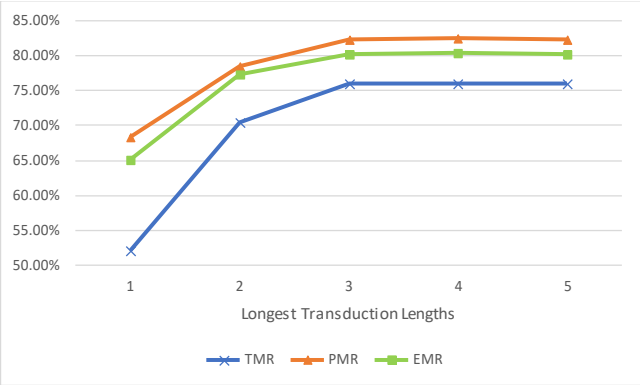

Figure 7: The impact of W on migration rates

Here, the test coverage of the original iOS GUI tests is not very high (ranging from 44.3% to 75.6%), so the test coverage of the generated Android test cases are also not very high. However, as revealed in previous studies [56], manual tests often achieve relatively lower coverage but cover most popular and error-prone software features. Our approach achieves similar test coverage (not only coverage values, but also the covered code) on the new platform. On the execution time of TestMig, it ranges from 10 minutes (2048) to 65 minutes (Wikipedia), which are acceptable for the application scenario.

4.4 Impact of Transduction Lengths

To explore RQ2, we studied how the average TMR, PMR, and EMR of four subjects change, when the transduction length w increases from 1 to 5. Here, we consider w value from only 1 to 5, since it is unlikely that an iOS GUI event is translated to more than five Android GUI events, according to our inspection.

Figure 7 shows the results. From the figure, we can observe that, as w increases, all three migration rates grows more and more slowly, and the migration rate becomes very stable after w reaches 3. This actually shows that almost all successfully translated iOS GUI events are mapped to at most three Android events. Therefore, we believe that 3 is a proper value for longest transduction sequence, and in our evaluation we use w=3 as the default value of TestMig. Furthermore, the figure shows that TestMig gains a lot on all migration rates (18.2, 10.2, and 12.3 percentage points gain for TMR, PMR, and EMR, respectively) when w increases from 1 to 2. The result shows that one-to-one event match is not sufficient, and our sequence-transduction-based exploration mechanism is helpful.

We further inspected the many-to-many mappings that are observed by TestMig, and found that they account for 65 out of 1125 mappings (5.8%, including 58 1-to-many mappings and 7 many-to-1 mappings). Note that many-to-many mappings can be naturally decomposed to 1-to-1 mappings and many-to-1 / 1-to-many mappings naturally by sequence transduction. Despite the sparsity of 1-to-many and many-to-1 mappings, they have a large impact on the test migration rate. Figure 7 shows that turning off many-to-1 / 1-to-many mapping (using transduction length 1) will cause TMR to drop 23 percentage points. The reason is that, as long as there is one many-many mapping in the migration process of a test case, it will cause a migration failure if not properly handled. Furthermore, failing to handle a 1-to-many / many-to-1 mapping will cause failures on converting all following events, no matter they should be mapped 1-to-1, 1-to-many, or many-to-1.

4.5 Categorization of Migration Failures

In our evaluation, we failed to fully migrate 26 out of 116 test cases. The main reason is that the generated Android UI event sequence stuck at a certain place and cannot invoke the next UI event, or the event sequence goes to a "wrong way". It should be noted that, although failures in code migration may significantly undermine the technique's usability, failures in test migration may not affect usability much, as long as the failing rate is relatively low. The reason is that, an unsuccessfully migrated test case is still a test case and may detect bugs in the target app version, although it may not be as good as the successfully migrated ones, and miss part of the consistency checking between versions. Since it partially leverages information from the original test, it may be still better than automatically generated test cases (e.g., on passing log-in guard). Furthermore, as shown in Figure 4, our PMR values are larger than TMR values, indicating that TestMig may have successfully migrated a large portion of a test case before the failure happens.

To answer question RQ3, we further investigated these, and found the root cause mainly falls into 3 categories.

Missing Features. The first category of migration failures are due to missing features in Android versions, so that an original GUI event cannot be matched to anything. This reason accounts for 14 out of 26 migration failures, including 8 migration failures in SimpleNote and Wire. Please note that these migration failures are harmless, because the testing

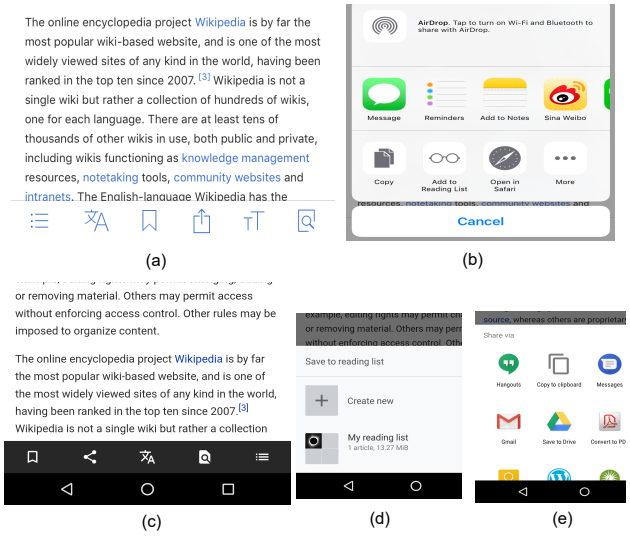

Figure 8: Example of a misleading GUI event

and migration themselves are unnecessary due to the missing features.

System Events. The second category of failures are due to TestMig's inability to handle system events, so the Android tests will stuck when a system event is simulated in iOS tests. This accounts for 8 out of 26 migration failures, including 5 migration failures from Wire, which is a communication software and relies more on system events. In the future, we plan to further map system events between Android and iOS to reduce such failures.

Similar UI Events. The third category of failures happen when multiple GUI events will enable similar following GUI events (accounting for 3 migration failures). Longer transduction lengths will typically solve such issues, but when the future events are not visible by GATOR, TestMig will make mistake. Figure 8 shows an example from Wikipedia, where the user is reading an article. Figure 8a and Figures 8b are screenshots from iOS, and the rest are from Android. In this example, the menu at the bottom of Figure 8a and Figures 8b have different orders in iOS and Android. In the iOS version, the third and fourth icons from left, triggers "save to history" and "share", respectively. In the iOS version, they are the first and second icons. All the icons have simple ids as `item1` through `item5` / `item6`, so the similarity measurement does not work on them, and we have to rely on the sequence transduction to find the similarity on following GUI events. In the Android version, the following event of clicking on the "save to history" button is "save to reading list", which is presented in Figures 8d. However, in the iOS version, this feature happen to be deleted, and the "share" feature is followed by the screen shot in Figures 8b, which can trigger a GUI event to "Add to reading list". Here, as the reading list in the iOS version is the reading list of the system instead of the Wiki app, it is a part of "share" feature. In such a case, TestMig mistakenly explores to the "save to list" feature, instead of the "share" feature in the original iOS test case. Since the events after "share" are outside the app, they are not visible to GATOR, and TestMig cannot get sufficient guidance. To reduce such failure, it is possible to extend GATOR with some common system GUI events for further guidance of TestMig.

4.6 Threats to Validity

There are two major threats to the internal validity of our evaluation. First, there can be mistakes in our data processing and bugs in the implementation of TestMig. To reduce this threat, we carefully double checked all the code and data in our evaluation. Second, the manual validation of test and event migrations may involve subjective bias. From our experience, the GUI semantics of the iOS version and the Android version are very similar although the design can be different. As a result, it is not difficult to determine whether two features from two versions can be matched to each other. We further reduce this threat by having multiple evaluators examining the test executions. The major threat to the external validity of our evaluation is that our conclusion may apply to only the apps being evaluated. To reduce this threat, we use apps from different domains to cover more testing scenarios.

5 DISCUSSION

System Events. TestMig currently does not convert system events such as incoming calls and low battery warnings. One reason is that we did not observe much test code related to system events in our subject projects. To support the conversion of system events, we need to construct their mappings between iOS and Android. The good part of system events is that, unlike GUI controls and events, they are predefined in the system and platform API, so we can construct a mapping in advance (although it may still need updates over time due to new iOS and Android versions). Furthermore, we need to record the system events being invoked when executing the iOS tests. After that, we should be able to use the same approach presented in this paper to convert a trace of interleaving UI and system events from iOS to Android. A potential technical challenge in the process lies in the dependencies and changes on the order of GUI events and system events in Android and iOS. However, GUI events and system events are often independent from each other (e.g., a phone call can come at any time) so such cases should be rare, and the order changes can be solved using the same idea of sequence transduction as described in this paper.

Migration of UI Test Oracles. Automatic test oracle is an essential part in automatic software testing, but has been one of the most difficult problem in the area. In our approach, we focus on the migration of iOS UI event traces to explore the Android app, and do not consider the migration of test oracles (e.g., assertions in the iOS test script). However, just as using any other automatic UI-test-case generation tool, developers can still use with our approach the general automatic UI test oracles such as crashes, unhandled exceptions, bad presentation, and unresponsive UI controls. We observe at least two more technical challenges in the migration of

UI test oracles. First of all, test-oracle migration requires more precise mapping of UI controls cross platforms. In the transduction of UI event traces, we can explore the Android app with multiple mapping options to double confirm the mapping. This is not possible for test oracles and the imprecision in mapping will cause imprecision in the test results. Second, values in the UI test assertions may be platform specific. For example, some assertions refer to absolute positions of UI controls and padding sizes, which are affected by the resolutions and various default settings which are different between iOS and Android. However, despite the technical challenges, migration of UI test oracles is still potentially feasible. We plan to work in near future on overcoming the challenges mentioned above and examining the feasibility of UI-test-oracle migration.

UI-Test-Case Migration in General. Migration of UI test cases among various UI frameworks is a general problem, and our approach just solves an specific instance of migrating iOS UI test cases to Android. One reason that we start from iOS to Android test migration is the existence of UIAutomator [16] (available for Android) which extracts runtime GUI-view hierarchy and provides nice APIs. iOS provides a similar driver in XCodeTest with GUI but no APIs are available. Another reason is our observation that commercial app developers often start from iOS (See Section 1), maybe because iOS is more profitable.

6 RELATED WORKS

Test Migration. Researchers have proposed techniques on migrating or learning to generate GUI tests from other apps. In particular, Behrang and Orso [27, 28] proposed to migrate tests between different Android apps of the same category, based on mapping of GUI events. Although TestMig is also based on mapping of GUI events, we further developed the event mapping between iOS and Android GUI infrastructures. Furthermore, we use sequence transduction to tolerate design difference between the two infrastructures. Additionally, Mariani et al. [43] developed Augusto, a technique to generate GUI tests for common app features that are reusable among different apps. Rau et al. [52] proposed to learn GUI interaction rules from tests of other apps. However, these approaches do not migrate individual test cases but try to learn general testing rules.

Mobile GUI Analysis. Since the emergence of graphical user interfaces, there have been a large number of research efforts focusing on the extraction of an abstract model to describe the GUI, and summarize the relations among the GUI controls [38, 55]. To describe the runtime behavior of GUI, a number of models which summarize possible GUI event sequences at runtime have been studied, mainly by researchers working on GUI testing. These models may be in forms of automaton [32, 57], grammar [25], and AI Planning [39], etc. Also, a lot of techniques for the automatic extraction of these models are proposed, either based on dynamic analysis [20, 45] or static analysis [29, 54]. Based on these models, various researchers have developed techniques for automatic GUI testing of Android apps. GUIRipper [21] (MobiGUITAR [22]), A3E [26], and SwiftHand [30] build finite state models for UI and generate events to systematically explore states in the model. Contest [24] generates events based on a concolic execution approach and prunes search spaces. Liu et al. [41] introduced new ways to generate text inputs. These GUI analysis techniques may further enhance TestMig, but they do not directly work on test migration.

Cross-Platform Code Migration. The academia has noticed the difficulties in the library migration, and many research efforts have been made on the topic. There have been many empirical studies [51] [44] [48] to explore the prevalence of backward incompatibility in library upgrade. Furthermore, Linares-Vsquez et al. [40] studied the relationship between change proneness of APIs methods and the success of client software using the methods. The research topic more relevant to our work is support for library migration, including the mapping of APIs between two consecutive versions of a software library. Godfrey and Zou [34] proposed a number of heuristics based on text similarity, call dependency, and other code metrics, to infer evolution rules of software libraries. Later on, S. Kim et al. [37] further improved their approach to achieve fully automation. Dagenais and Robillard [33] proposed *SemDiff*, which infers rules of framework evolution via analyzing and mining the code changes in the software library itself. Wu et al. developed *AURA* [58], which further involves multiple rounds of iteration applying call-dependency and text-similarity based heuristics on the code of software library itself. HIMA [46] further enhances *AURA* by involving historical information [35] between two consecutive versions of software libraries. Nguyen et al. [50] proposed techniques to mine code-change patterns for API migrations, and to recommend code changes. Although they are relevant, all these techniques cannot be applied directly to test migration because unlike APIs, GUI events are unique for each app so no training or preparation steps can be taken.

7 CONCLUSION

In this project[2], we propose a novel approach that migrates iOS UI test cases to Android UI test cases. Specifically, we record the iOS UI event sequences invoked during the iOS UI testing, and use the sequence transduction technique to convert this sequence to a sequence of Android UI events. Then, we explore the Android version of the software with the Android UI events and record the test cases. We evaluate our approach on four popular cross platform mobile apps, and the result shows that our approach can successfully convert on average 80.2% of the iOS UI test cases to Android test cases, and achieve an average test coverage of 59.7%, which is close to the 60.4% average test coverage of the original iOS test cases on the IOS versions of the apps.

[2]The UTSA authors are supported in part by NSF awards 1748109 and 1846467. Hao Zhong is supported by National Key R&D Program of China No. 2018YFC0830500, and National Nature Science Foundation of China No. 61572313.

REFERENCES

[1] Flappy bird. https://en.wikipedia.org/wiki/Flappy_Bird, 2013.
[2] iOS torches Android when it comes to developer profits. http://bgr.com/2016/07/20/ios-vs-android-developers-profits-app-store-google-play/, 2016.
[3] Apache cordova. https://cordova.apache.org/, 2017.
[4] Clean master. https://play.google.com/store/apps/details?id=com.cleanmaster.mguard, 2017.
[5] Crossy road. https://play.google.com/store/apps/details?id=com.yodo1.crossyroad, 2017.
[6] Facebook. https://www.facebook.com/, 2017.
[7] Facebook messager. https://www.messenger.com/, 2017.
[8] Google play store. https://play.google.com/store, 2017.
[9] Instagram. https://www.instagram.com/, 2017.
[10] Jacoco. http://www.eclemma.org/jacoco/, 2017.
[11] Kika emoji. https://play.google.com/store/apps/details?id=com.qisiemoji.inputmethod, 2017.
[12] Monkey runner. https://developer.android.com/studio/test/ monkeyrunner/index.html, 2017.
[13] Pandora radio. https://www.pandora.com/, 2017.
[14] Snapchat. https://www.snapchat.com/, 2017.
[15] Spotify. https://www.spotify.com/, 2017.
[16] Ui automator. https://developer.android.com/training/testing/ui-automator.html, 2017.
[17] Unity 3d. https://unity3d.com/, 2017.
[18] Whatsapp. https://www.whatsapp.com/, 2017.
[19] A. Aizawa. An information-theoretic perspective of tf–idf measures. *Information Processing & Management*, 39(1):45–65, 2003.
[20] 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*, pages 258–261, 2012.
[21] 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*, pages 258–261. ACM, 2012.
[22] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE software*, 32(5):53–59, 2015.
[23] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.
[24] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.
[25] M. Auguston, J. B. Michael, and M.-T. Shing. Environment behavior models for scenario generation and testing automation. *SIGSOFT Softw. Eng. Notes*, 30(4):1–6, 2005.
[26] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, volume 48, pages 641–660. ACM, 2013.
[27] F. Behrang and A. Orso. Automated test migration for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 384–385. ACM, 2018.
[28] F. Behrang and A. Orso. Test migration for efficient large-scale assessment of mobile app coding assignments. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 164–175. ACM, 2018.
[29] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in android applications with permission event graphs. In *Network and Distributed System Security Symposium*, 2013.
[30] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, volume 48, pages 623–640. ACM, 2013.
[31] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015.
[32] J. Clarke. Automated test generation from a behavioral model. In *Proceedings of the Pacific Northwest Software Quality Conference*, 1998.
[33] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. ICSE*, pages 481–490, 2008.
[34] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE TSE*, 31(2):166–181, February 2005.
[35] F. Hassan and X. Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1078–1089. IEEE, 2018.
[36] M. E. Joorabchi, M. Ali, and A. Mesbah. Detecting inconsistencies in multi-platform mobile apps. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 450–460, 2015.
[37] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *Proc. WCRE*, pages 143–152, 2005.
[38] A. Kull. Automatic gui model generation: State of the art. In *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, pages 207–212, 2012.
[39] W. K. Leow, S. C. Khoo, and Y. Sun. Automated generation of test programs from closed specifications of classes and test cases. In *Proceedings of the 26th International Conference on Software Engineering*, pages 96–105, 2004.
[40] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 477–487, 2013.
[41] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng. Automatic text input generation for mobile testing. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 643–653. IEEE, 2017.
[42] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609, 2014.
[43] L. Mariani, M. Pezzè, and D. Zuddas. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 280–290. IEEE, 2018.
[44] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 70–79, 2013.
[45] 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*, WCRE '03, pages 260–269, 2003.
[46] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 353–363, Piscataway, NJ, USA, 2012. IEEE Press.
[47] M. Mohri. Finite-state transducers in language and speech processing. *Comput. Linguist.*, 23(2):269–311, June 1997.
[48] S. Mostafa, R. Rodriguez, and X. Wang. Experience paper: a study on behavioral backward incompatibilities of java software libraries. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 215–225. ACM, 2017.
[49] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining api usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 457–468, 2014.
[50] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 302–321, 2010.
[51] S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 378–387, 2012.
[52] A. Rau, J. Hotzkow, and A. Zeller. Poster: Efficient gui test generation by learning from tests of other apps. In *2018 IEEE/ACM*

40th International Conference on Software Engineering: Companion (ICSE-Companion), pages 370–371. IEEE, 2018.

[53] A. Rountev and D. Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 143:143–143:153, New York, NY, USA, 2014. ACM.

[54] S. Staiger. Reverse engineering of graphical user interfaces using static analyses. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 189–198, 2007.

[55] X. Wang, X. Qin, M. B. Hosseini, R. Slavin, T. D. Breaux, and J. Niu. Guileak: Tracing privacy policy claims on user input data for android applications. In *Proceedings of the 40th International Conference on Software Engineering*, pages 37–47. ACM, 2018.

[56] X. Wang, L. Zhang, and P. Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 199–210, 2015.

[57] L. White and H. Almezen. Generating test cases for gui responsibilities using complete interaction sequences. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 110–123, 2000.

[58] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A hybrid approach to identify framework evolution. In *Proc. ICSE*, pages 325–334, 2010.

[59] H. Zhong and H. Mei. An empirical study on API usages. *IEEE Transaction on Software Engineering*, 2018.

[60] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proc. ICSE*, pages 195–204, 2010.