# UI Test Migration Across Mobile Platforms

***Abstract*—Writing UI tests manually requires significant effort. Several approaches have tried to address this problem in mobile apps: by exploiting the similarities of different apps within the same domain on a single platform, they have shown that it is possible to transfer tests that exercise similar functionality between the apps. A related recent technique enables transfer of UI tests uni-directionally, from an open-source iOS app to the same app implemented for Android. This paper presents MAPIT, a technique that expands existing work in three important ways: (1) it enables bi-directional UI test transfer between pairs of "sibling" Android and iOS apps; (2) it does not assume that the apps' source code is available; (3) it is capable of transferring tests containing oracles in addition to UI events. MAPIT runs existing tests on a "source" app and builds a partial model of the app corresponding to each test. The model comprises the app's screenshots, obtainable properties of each screenshot's constituent elements, and labeled transitions between the screenshots. MAPIT uses this model to determine the corresponding information on the "target" app and generates an equivalent test, via a novel approach that leverages computer vision and NLP. Our evaluation on a diverse set of widely used, closed-source sibling Android and iOS apps shows that MAPIT is feasible, accurate, and useful in transferring UI tests across platforms.**

## I. Introduction

Writing UI tests manually requires significant effort. This is an especially acute problem on mobile platforms given their rapid app-development lifecycle. A popular alternative is to automatically generate the UI tests, e.g., by relying on model-based or random testing [1], [2]. While these approaches have been shown effective for generating UI tests with high code coverage, they cannot generate *usage-based tests* that target an app's specific functionality [3], such as login, sign-up, make reservation, etc. It has been shown that such usage-based tests are highly valuable to developers and testers [3], [4].

Recent work has demonstrated the possibility of generating usage-based tests through test reuse [5], [3], [6], [7], [8], [9]. The key insight behind these approaches is that apps within a single domain (e.g., news, shopping, etc.) share similar functionalities, resulting in conceptually similar UIs. Guided by this insight, prior work has primarily focused on transferring existing usage-based tests to a new app *within the same domain on a single platform* [5], [6], [8], [7], [3]. Specifically, all of these approaches leverage existing usage-based tests from a source Android app to automatically generate equivalent tests for a target Android app.

A largely unexplored variant of this problem is transferring tests written for an app implemented on one platform (e.g., Android), to the same app *implemented for another platform* (e.g., iOS). We refer to such pairs of apps as *sibling* apps. A solution to this problem would be valuable since many companies, as well as independent developers, target multiple platforms to maximize their user-base. *Cross-platform* transfer has unique challenges as compared to test transfer within Android alone. First, different platforms employ different technologies, such as various app-development frameworks and programming languages, which add significant complexity to the problem. Second, iOS is a closed-source platform, which has led to fewer and more limited tools for analyzing iOS apps compared to Android. Finally, most iOS apps themselves are closed-source, making any code-based analysis impossible.

The closest attempt at this problem is TestMig [10], which has addressed *uni-directional* test transfer for sibling iOS and Android apps. TestMig has three important limitations. First, it assumes the availability of both the Android and iOS apps' source code, which, as mentioned, is especially unlikely for iOS apps. Second, TestMig only covers transferring tests in one direction (iOS to Android). Supporting the other direction (Android to iOS) is inherently challenging: unlike Android, for which many open-source reverse engineering tools are readily available (e.g., bytecode decompilers, Soot [11], Gator [12]), iOS is a closed platform with a smaller developer base and static analyses that existing test transfer techniques rely on are not an option. Third, TestMig only targets UI events, but cannot transfer test oracles or system events. Oracles are responsible for evaluating the outcomes of tests and are therefore an essential part of usage-based testing. Inability to migrate system events additionally limits the set of test cases that can be transferred. For instance, navigating to the previous screen is an Andriod system event that frequently occurs in UI tests.

To address these limitations, we have developed MAPIT, a novel approach for *bi-directional transfer* of usage-based tests across *different mobile platforms*, with *no source code required* on either platform. Specifically, MAPIT takes as input (1) the binaries of the sibling apps-under-test implemented for both iOS and Android as well as the (2) pre-existing tests for one of these platforms, and automatically generates equivalent tests for the other platform. The transfer process comprises two major phases. First, MAPIT dynamically extracts a GUI model of the app on the source platform while executing the source test. The GUI model contains the app screen bitmaps, information regarding the relevant widgets contained on each screen (e.g., widget images and descriptive attributes), and the events that cause transitions from one app screen to the next. Second, based on this GUI model, the source test is migrated to the target platform by mapping the GUI widgets from the source app to the most similar widgets in the sibling app, using a combination of computer vision and NLP techniques. This process additionally recognizes and transfers oracle and system events.

We empirically evaluated MAPIT on 25 pairs of sibling Android and iOS apps. We selected among the most popular commercial apps spanning 5 app categories. For each pair of sibling apps, we transferred 4 test cases corresponding to representative usage scenarios in both directions to evaluate MAPIT's ability to correctly transfer both (1) individual events and test oracles, as well as (2) complete tests. In total, our evaluation yielded

bidirectional transfers of 200 test cases, including 828 UI events, 176 oracle events, and 50 system events. Overall, MAPIT achieved over 75% event mapping accuracy and showed to be useful in reducing the required manual effort by over 55%. Furthermore, 58 (29%) complete test cases were transferred correctly, which in those instances completely eliminates the manual effort. Note that, even if a test is not completely transferred correctly, the reduction in manual effort MAPIT affords is proportional to the fraction of individual events that were successfully transferred. In these cases the developer can complete the partially transferred test by manually inspecting it and only modifying the events tagged as incorrectly transferred.

This paper makes the following contributions:

- A novel technique for bi-directional, cross-platform transfer of usage-based tests for closed-source apps.
- A novel UI widget mapping solution that combines pluggable computer vision and NLP techniques.
- An extensible approach for transferring test oracles and system events across mobile platforms.
- An empirical evaluation on 25 closed source real-world apps, and a public repository with MAPIT's implementation and artifacts to foster future research [13].

Section II presents our work's background via an example and introduces the key terminology. Section III presents our approach and Section IV its empirical evaluation. Section V overviews the related work. Section VI concludes the paper.

## II. Background and Terminology

Figure 1 shows the screenshots of the login pages of Etsy, a popular shopping app, on Android (left) and iOS (right). Although the two login pages are not identical, they share significant similarities in (1) the appearances of their UI widgets, (2) the textual data describing these widgets, and (3) the widgets' position on the respective screens. Such, and even greater, pairwise similarities between "sibling" apps are common.

Let us assume that a test of Etsy's login functionality exists on Android, and that we want to automatically transfer it to iOS. The widgets that are involved in the login test are framed and labeled for both platforms in Figure 1. We will use this scenario to introduce the terms and describe the concepts that are relevant to our approach.

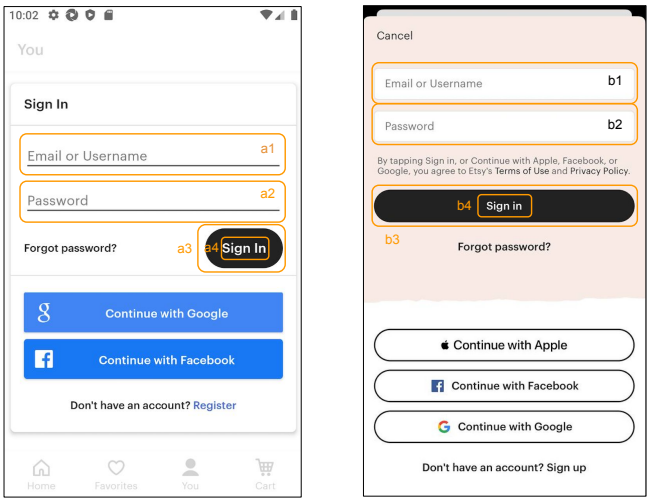


Fig. 1: Login pages of Etsy on Android (left) and iOS (right).

The *source app* is the app with existing tests that are to be transferred to the *target app*. *Source platform* and *target platform* are the platforms on which the *source* and *target* apps run, respectively. The *source test* is the existing test to be transferred, while the *target test* is the transferred test. A *ground-truth test* is an existing test for the *target app* that tests the same functionality as the transferred *source test*. A *test scenario* is an informal description of a test case in natural language. For instance, the login *test scenario* consists of entering username and password and clicking the "login" button. A *ground-truth test* is thus used for evaluating the success of a test transfer corresponding to the same *test scenario* in the two sibling apps. Note that the *source test* that is transferred, e.g., from iOS to Android, serves the *ground-truth test* when transferring the same *test scenario* in the opposite direction.

The contents of a given screen of an app form an *app state*. *Equivalent states* on the *source* and *target* apps are the states intended to provide the same or equivalent functionalities, usually with similar-looking UIs. For instance, Figure 1 shows two *equivalent states* of the Etsy app on Android and iOS since they both target login functionality.

An *event* is defined as a 4-tuple $(a, w, t, o)$. Each *event* has one required element: $a$, which is the type of action associated with the *event*. The remaining three elements are optional: $w$ is the UI widget; $t$ is the input text associated with the *event*, such as text entered by the keyboard; and $o$ is the oracle type. MAPIT supports three types of events: (1) *UI events*, for which the supported action types are `click` and `keyboard input`; (2) *oracle events*, which are the assertions in the UI tests that determine whether a test should pass or fail, for which the supported action type is `oracle`; and (3) *system events*, for which the supported action types are `back` and `enter`. Note that $w$ is optional since not all events have an associated widget (e.g., *system events*).

If the *event* is an *oracle event*, $o$ contains the specific oracle type. To demonstrate MAPIT's ability to transfer oracles, in this paper we focus on a representative cross-section of assertion types identified by prior work [5], [8], which will be detailed in Section III.

As an example, the *UI event* of entering the username "Usr_1" for logging into Etsy on Android is represented as a 4-tuple `(a:''keyboard input'', w:a1, t:''Usr_1'', o:–)`.

Finally, each *UI test* consists of a sequence of 4-tuple *events*.

## III. Approach

Figure 2 shows an overview of MAPIT's workflow. The input to MAPIT is three-fold: (1) *Source Test* written for the source platform, (2) *Source App* that runs on the source platform, and (3) its sibling, *Target App* that runs on the target platform. MAPIT automatically transfers the *Source Test* through two major phases: (1) *Source Data Extraction*, during which the data needed for the test transfer is dynamically extracted from the *Source App*, and (2) *Test Migration*, during which the extracted data is used to generate the *Target Test*,

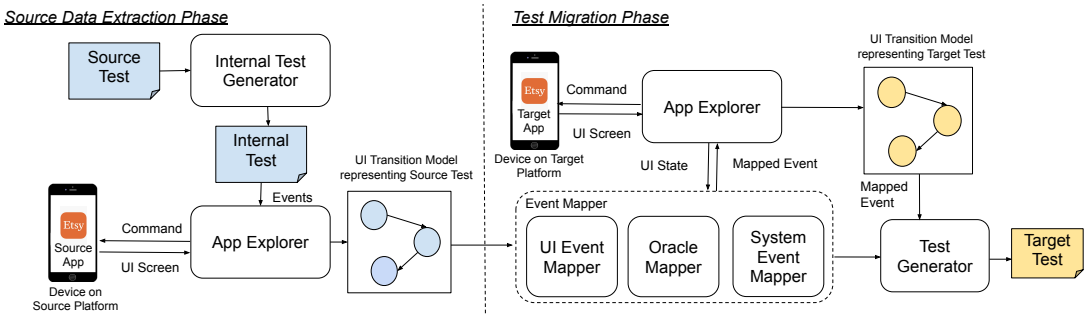


Fig. 2: High-level workflow of MAPIT.

the *Source Test*'s equivalent on the target platform. The remainder of this section details the two phases.

In developing MAPIT, we had to make several implementation decisions driven by the third-party technologies on which we relied. We highlight those whenever they are instrumental in enabling a particular facet of MAPIT. Overall, MAPIT is implemented in 4.5K SLOC of Python, additionally integrating off-the-shelf tools for mobile app monitoring, analysis, and testing.

### A. Source Data Extraction

During the source data extraction phase, the *Internal Test Generator* component first transforms the pre-existing *Source Test* to the *Internal Test*, which is captured in MAPIT's internal representation for test cases. This internal representation is both programming language- and testing framework-independent, a critical requirement of cross-platform test transfer. Based on the *Internal Test*, the *App Explorer* component gradually generates a *UI Transition Model*, which consists of the observed UI states of the app and the transitions between them. Each transition represents one event (e.g., button click) within the corresponding test case. The *UI Transition Model* is generated by executing each event of the *Internal Test* on the source platform, and dynamically extracting the requisite information from the source app. We describe the *Internal Test Generator* and *App Explorer* components in more detail next.

#### *Internal Test Generator*

As mentioned above, this component translates the *Source Test* into the language- and platform-independent *Internal Test*. Figure 3 illustrates this with an example of translating a partial test of Etsy's login functionality, written in Python for the Appium testing framework [14] (Figure 3-a), to MAPIT's internal format (Figure 3-b). A test is represented internally by MAPIT as a sequence of 4-tuple *event*s, as defined in Section II, with the event elements $w$, $t$, and $o$ being optional. Note that each widget $w$ contains the information used to locate this widget in the source test, such as `accessibility id`, `resource id`, `XPath`, or coordinates. Thus $w$ is represented as a `locator-type` and its corresponding `locator-value`.

**(a)**

```
el1 = Appium.webdriver.find_element_by_id("com.etsy.android:id/edit_password")
el1.send_keys("password")
el2 = Appium.webdriver.find_element_by_id("com.etsy.android:id/button_signin")
el2.click()
el3 = Appium.webdriver.find_element_by_accessibility_id("You tab, 4 of 5")
el3.click()
WebDriver.wait(Appium.webdriver,10).until(EC.visibility_of_element_located(By.id,
"com.etsy.android:id/username"))
```

**(b)**

```
{"action": "keyboard input",
 "Widget": {"locator-type": "resourceId","locator-value": "com.etsy.android:id/edit_password"},
 "text-input": "password"}
{"action": "click",
 "Widget": {"locator-type": "resourceId","locator-value": "com.etsy.android:id/button_signin"}}
{"action": "click",
 "Widget":{"locator-type": "accessibilityId","locator-value": "You tab, 4 of 5" }}
{"action": "oracle",
 "Widget":{"locator-type": "id","locator-value": "com.etsy.android:id/username"}
 "oracle": "widget-displayed"}
```

Fig. 3: Translating (a) Etsy's login test written in Python for the Appium framework to (b) MAPIT's internal representation.

As a proof of concept, MAPIT currently includes support for translating tests written in Python for Appium [14] and the Robot [15] framework. These two frameworks are widely used in mobile-app testing. The translation in each case is done by mapping framework-specific tests to MAPIT's internal test representation through regular expression matching. For instance, the first two lines of the Appium Python test shown in Figure 3-a are matched by the regular expressions `''.*driver.find_element_by_(.*)\(\"(.*)\"\)''`, and `''el.*\.(.*)\( (.*)\)''` respectively, and the `action`, `locator-type`, `locator-value`, and `input` elements are extracted from them to form the first event in MAPIT's internal test format shown in Figure 3-b. This mechanism can be easily extended to translate UI tests written in other programming languages and/or for other testing frameworks.

#### *App Explorer*

As shown in Figure 2, the *App Explorer* component interacts with a mobile device and gradually generates the *UI Transition Model* of the *Source App* while executing the sequence of *Events* in the *Internal Test*. Specifically, *App Explorer* consists of three sub-components, as shown in Figure 4: *Event Executor*, *State Extractor*, and *Model Generator*.

*1) Event Executor:* This sub-component is tasked with initiating and maintaining an active connection with the mobile device. It takes the *Events* from MAPIT's *Internal Test* as input, and transforms each event to the corresponding commands that are transmitted to the device and executed. *Event Executor* uses Appium [14] for device communication. In turn, Appium relies on Android Debug Bridge [16], a tool for communicating with Android devices, and Web Driver Agent [17], an interface for remotely interacting with iOS devices.

*2) State Extractor:* For each event triggered by *Event Executor*, *State Extractor* captures and processes the data associated with the current device screen, in order to generate

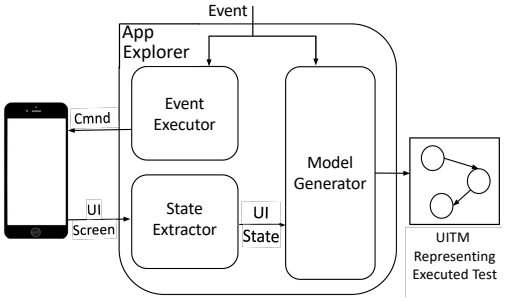

Fig. 4: *App Explorer*'s internal architecture.

the current *UI State*. A UI state $S$ consists of (1) the app's current screenshot, (2) the graph representing the screen's UI layout hierarchy, and (3) all UI widgets that exist on the current screen. Figure 5 shows an example of the UI state extracted from the Etsy app at the beginning of a login test scenario.

Specifically, *State Extractor* first captures the bitmap of the current screen (shown in the center of Figure 5) and extracts its *page source*, which contains the screen's UI information as an XML hierarchy [18]. It then iterates through the UI layout contained in the page source and builds a graph based on the UI element hierarchy (shown on the left of Figure 5). While iterating through the UI layout, *State Extractor* extracts the boundaries of each UI widget and crops the captured screenshot to get the image representing the widget on the current screen. Also extracted and stored are each widget's descriptive attributes in the UI layout, such as `resource id`, `name`, coordinates, element type, and whether it is interactable (shown on the right of Figure 5). Finally, if the widget has any visible text on its image, such text is captured using OCR. To this end, we leveraged the Tesseract OCR engine [19].

*3) Model Generator: App Explorer*'s third sub-component incrementally generates a *UI transition model* (*UITM*), based on the *UI State*s extracted from *State Extractor* (e.g., recall Figure 5)) and their corresponding *Events* obtained from *Internal Test Generator* (recall Figure 2). *UITM* is a linear FSM representing the transitions in the app taken while executing each event. *UITM* has no back transitions: app state associated with a given screen is captured separately each time the screen is visited.

Specifically, $UITM(A, T)$ is the transition model of app $A$ associated with UI test case $T$, which is a sequence of events $e_1, e_2, ..., e_n$. The initial UI state of the app when executing $T$ is annotated as $S_0$. Every state $S_i$ is reached by successfully executing event $e_i$ in state $S_{i-1}$. If the event sequence contains *n* events, the *UITM* will contain $n + 1$ states: $S_0$, $S_1$, ..., $S_n$. Figure 6 shows the *UITM* extracted from the Android version of Etsy, representing a login test case consisting of five events.

### B. Test Migration

In MAPIT's second phase, the source test is migrated to the target platform. This is done by transforming $UITM(A, T)$, extracted during the previous phase, into $UITM(A', T')$, where $A'$ is the sibling app of $A$ and $T'$ is the test generated by MAPIT to target the same functionality on $A'$ that $T$ targeted on $A$.

Test migration is accomplished iteratively via three principal components: *App Explorer*, *Event Mapper*, and *Test Generator*

**Algorithm 1:** High-Level Test Migration Process

**Input:** App $A'$, UI Transition Model $UITM(A, T)$
**Output:** Test $T'$, UI Transition Model $UITM(A', T')$

1 $UITM(A', T') = \emptyset$;
2 $S'_0 = extract\_current\_state(A')$;
3 $UITM(A', T').add\_state(S'_0)$;
4 **foreach** $(S_i, e_{i+1})$ *in UITM(A, T)* **do**
5 $\quad e'_{i+1} = map\_event(S_i, e_{i+1}, S'_i)$;
6 $\quad execute\_event(A', e'_{i+1})$;
7 $\quad T'.add\_event(e'_{i+1})$;
8 $\quad S'_{i+1} = extract\_current\_state(A')$;
9 $\quad UITM(A', T').add\_state(S'_{i+1})$;
10 $\quad UITM(A', T').add\_transition(S'_i, S'_{i+1}, e'_{i+1})$;
11 **end**

(recall Figure 2). A high-level summary of this phase is provided in Algorithm 1. $UITM(A', T')$ is initialized with $S'_0$, which is the initial state of the target app (Lines 1-3). As depicted in Figure 2, this state is extracted using a second instance of the *App Explorer* component discussed in the previous phase; this instance of *App Explorer* is responsible for interacting with the target device. For each state $S_i$ and transition-triggering event $e_{i+1}$ in $UITM(A, T)$, the *Event Mapper* component finds the equivalent $e'_{i+1}$ event in the current state $S'_i$ on the target platform (Line 5). $e'_{i+1}$ is executed by *App Explorer*, resulting in the transition from $S'_i$ to $S'_{i+1}$ (Line 6). Additionally, $e'_{i+1}$ is added to the target test $T'$ by the *Test Generator* component (Line 7). At this point, $S'_{i+1}$ becomes the current state and its corresponding information will be extracted by *App Explorer* (Line 8), and added to $UITM(A', T')$ as a new state connected to $S'_i$ via the transition corresponding to $e'_{i+1}$ (Lines 9-10). When the final state of the source app's model is reached, all events in the source test have been migrated to the target platform.

*Event Mapper* is the core component of MAPIT. It dynamically maps each event from the source platform to its equivalent event that is executable in the current state of the target app. It takes each state and event in the extracted $UITM(A, T)$ from the previous phase, as well as the corresponding state in the target app, and outputs the mapped event. Recall from Section II that MAPIT supports the transfer of three types of events: (1) UI, (2) oracle, and (3) system events. Correspondingly, as shown in Figure 2, *Event Mapper* consists of *UI Event Mapper*, *Oracle Event Mapper*, and *System Event Mapper* sub-components. Mapping a source event is handled differently by its corresponding mapper based on the event type. We now detail each of these mappers. We will then elaborate on *Test Generator*, the final component in MAPIT's architecture.

#### *UI Event Mapper*

Recall from Section 2 that an event is defined as a 4-tuple ($a$, $w$, $t$, $o$), several of whose elements may be optional. All elements of the 4-tuple that are present in a given event need to be mapped from the source to the target platform.

For UI events, the action type $a$ can be `click` or `keyboard input`. In both of these cases, the element $a$ will be migrated to the target platform as-is. For the `keyboard input` action type, the event will also contain the text input $t$, which remains unchanged in the mapped event.

To map a source UI widget $w$ to its most similar widget in the current UI state of the target app, MAPIT leverages both visual and textual information. As shown in Figure 7, it does so via *Visual Comparator*, *Textual Comparator*, and *Widget Selector* sub-components. In the ensuing discussion, we refer to the combination of these three sub-components as *UI Widget Mapper*, to distinguish them from the entire *UI Event Mapper*: although a key function of *UI Event Mapper* is indeed the mapping of UI widgets $w$, it is also responsible for mapping the other elements of an event $(a, t, o)$ to the target platform. We now elaborate on each of the three sub-components of *UI Widget Mapper*.

*1) Visual Comparator:* This sub-component extracts the cropped image and coordinates of widget $w$ from source app's state $S$, and all widgets that exist in target app's current UI state $S'$. It then calculates a *visual similarity score* with respect to $w$ for each widget of $S'$ and ranks the target widgets accordingly.

The intuition behind this component is that equivalent widgets on different platforms tend to have very similar looks by design. Visual similarity of two widgets is computed as the weighted average of their (1) image, (2) screen location, and (3) size similarities. Computing the latter two scores is relatively straightforward. The *proximity* score of two widgets is computed based on the Euclidean distance [20] of their locations on the screen. The *size* similarity score is determined based on the difference in widget sizes (normalized by the device size).

To compute the *image* similarity score, *Visual Comparator* leverages the *key points* and *feature descriptors* extracted from both the source and destination widget images. An image's key points are its pixels that have a prominent difference of intensity with their adjacent pixels [21]. Feature descriptors are numerical representations that encode data about each key point's neighborhood [21] and are used widely for image comparison [22]. For detecting an image's key points and subsequently its feature descriptors we use ORB [23] algorithm, which is an state-of-the-art image matching technique that has shown to be highly efficient.

Once the sets of feature descriptors corresponding to source and destination images have been obtained, *Visual Comparator* computes the Hamming distance [24] for each pair of source and destination descriptors. It then ranks the destination descriptors for each source descriptor based on the computed distance. Good matches between these two sets of descriptors are next determined via Lowe's ratio test [25], which is widely used in image matching tasks. In this test, for each descriptor of the source image, the two closest matches among the destination descriptors per the computed Hamming distance are selected. If the value of $\frac{distance(\ closest\ match\ )}{distance(\ second\ closest\ match\ )}$ is less than a customizable ratio, then the closest match is considered to be a *good* match. We empirically explored different ratios, and set the value to $0.8$ in our evaluation reported in Section IV. At the end of this process, each descriptor of the source image is labeled as *matched* if it has a good matching destination descriptor.

Finally, *Visual Comparator* computes a normalized image similarity score for each destination image with respect to the source widget $w$. This score is calculated as the ratio of *matched* source descriptors to the maximum number of descriptors extracted from the source and destination images.

*2) Textual Comparator:* This sub-component determines a *textual similarity score* between two widgets. It leverages the widgets' textual data from their respective *UITMs* extracted by MAPIT's *App Explorer* (recall Section III-A). A widget's textual data consists of the values of its *textual attributes*. These attributes are a subset of the information extracted by *State Extractor* (recall Figure 4) for each widget and include: (1) `content descriptor`, (2) `resource id`, and (3) `text`, for the widgets on Android; and their iOS counterparts (1) `accessibility id`, (2) `name`, and (3) `label`. As discussed previously, any text extracted from a widget's image is also included in its textual data.

For illustration, in the Etsy example from Figure 1, the textual data describing the ``Sign In'' button on Android ($a3$) is {`content descriptor:-`,

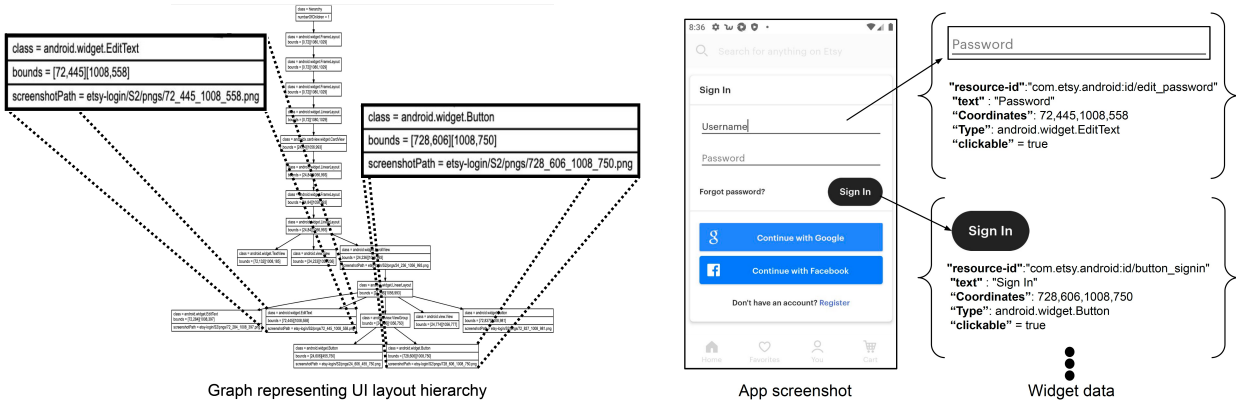


Fig. 5: UI state captured from Etsy's login page. The two UI layout hierarchy elements highlighted on the left correspond to the widget data shown on the right.

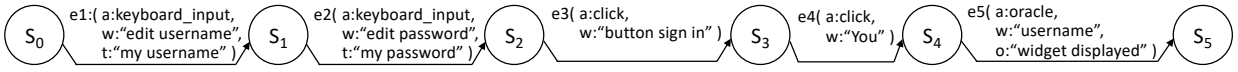


Fig. 6: UITM extracted from Etsy's login test case. Each state $S_0$-$S_5$ is in the format shown in Figure 5.

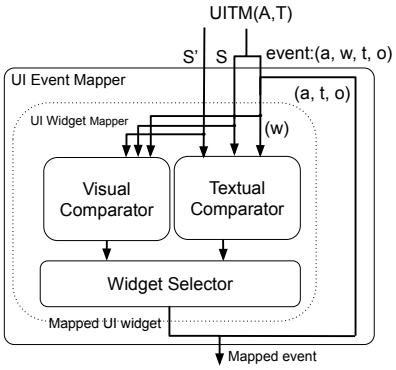


Fig. 7: *UI Event Mapper*'s internal architecture.

`resource id:''com.etsy.android:id/button_signin'', text:-, widget text:''Sign In''}` while the textual data describing the corresponding button on iOS ($b3$) is {`accessibility id:-, name:''Sign in'', label:''Sign in'', widget text:''Sign in''`}. Note that a UI widget need not have all of the mentioned textual attributes. Also, the values of multiple attributes may be identical. In the above case, the Android widget ($a3$) does not have the `content descriptor` attribute, while the values of `name`, `label`, and `widget text` attributes for the iOS widget ($b3$) are the same.

*Textual Comparator* first pre-processes the extracted textual data using common NLP practices, such as tokenization and stop-word elimination. In addition to general-purpose stop-words [26], we constructed a new list [13] of common stop-words in widgets' textual attributes that typically do not convey meaningful information, such as `''view''`, `''bar''`, and `''container''`. This list can be reused or extended to benefit NLP techniques in the mobile-app domain.

*Textual Comparator* computes a pairwise similarity score for a given pair of source−target widgets' textual attributes. It uses Word2Vec [27] and the standard *tf-idf* formula [28] to transform each textual attribute into its embeddings. The similarity score is then calculated based on the embeddings' cosine similarity [29]. *Textual Comparator* computes the similarity score of all pairs of textual attributes regardless of their types (e.g., it will compare `text` and `accessibility id`), to maximize the chance of discovering similar widgets. The reason is that meaningful textual values may be arbitrarily assigned to any attribute in practice. The *textual similarity score* between two widgets is then calculated as the highest cosine similarity score among the textual attribute pairs. This process naturally filters out the similarity scores calculated based on meaningless textual attributes.

As an example, the home button in Etsy on Android has `''com.etsy.android:id/menu_bottom_nav_home''` as its `resource id` and `''Home, tab 1 of 4''` as its `accessibility id`. The same button on iOS has `''Home''` as its `name` and `''1''` as its `value` attribute. After the preprocessing step, the textual attributes for the Android widget become `''Home''` and `''menu bottom nav home''`, respectively. While the cosine similarity between, e.g., `''menu bottom nav home"` and `''1''` is very low (0.007), Android's `resource id` and iOS's `name` are identical (`''Home''`). This means that the textual similarity score between the two widgets is 1.0.

*3) Widget Selector: UI Event Mapper*'s third sub-component selects the mapped UI widget based on the visual and textual similarity scores. It does so by first checking the top-ranked widgets based on the visual and textual similarity scores. If the respective top-ranked widgets are the same, *Widget Selector* will select this as the mapped widget. Otherwise, since the textual data is more informative, *Widget Selector* first checks whether the top-ranked widget in the textual similarity ranking has a score higher than a given, adjustable threshold. If such a widget $w'$ exists, then *Widget Selector* will first select all target widgets whose textual similarity score is within an adjustable proximity range of the textual similarity score of $w'$. These selected widgets are considered as *close* textual matches. *Widget Selector* will choose the widget with the highest visual similarity score among the close textual matches. If no widget's score is above the specified textual similarity threshold, then the mapped widget is the one with the highest visual similarity score.

At this point, the final mapped widget is checked for compatibility, based on whether its action type is supported on the target device. If the mapped widget cannot support the transferred action, *Widget Selector* will remove it from both rankings, and then choose another UI widget based on the above process.

*Oracle Event Mapper*

This component is responsible for mapping all four elements $(a, w, o, t)$ of an oracle event. For these events, the action type $a$ that is *oracle* and oracle type $o$ are always required, and are both migrated to the target platform as-is. Mapping the other two elements, $w$ corresponding to the widget and $t$ corresponding to text, is more challenging. MAPIT handles the transfer of these two elements depending on the oracle's type as detailed below. As explained in Section II, we currently support several common types of oracles identified by prior work [5], [8]; MAPIT can be easily extended to include additional oracle types. Table I shows the oracle types currently supported by MAPIT, divided into widget-independent and widget-dependent oracles.

The widget-independent oracle types currently supported by MAPIT are *text_existence(txt)* and *text_invisible(txt)*. These oracles, respectively, check the presence and visibility of text in the app's current state. In representing these oracles via MAPIT's internal events, the value of the parameter *txt* is captured by the oracle event's element $t$. In these cases, the text input $txt$ of the source event will be transferred to the target event as-is, and the presence or visibility of the same text checked on the target platform.

The widget-dependent oracle types MAPIT currently supports are *widget_exists(w)*, *widget_invisible(w)*, and *assert_equal(w, attr, val)*. For this group of oracles, the

| **Widget-independent Test Oracles** |
|---|
| *text_existence(txt)* |
| *text_invisible(txt)* |
| **Widget-dependent Test Oracles** |
| *widget_exists(w)* |
| *widget_invisible(w)* |
| *assert_equal(w, attr, val)* |

TABLE I: Oracle types supported by MAPIT.

corresponding MAPIT events also contain the widget element $w$, which is mapped by *UI Widget Mapper* as discussed above.

Transferring the *widget_invisible(w)* oracle is more challenging than *widget_exists(w)*. The reason is that the widget $w$ does not exist in the current state of the source app and, therefore, it is not possible to extract the needed data for the widget mapping process from the source UI state. To overcome this challenge we used a heuristic-based solution. Namely, we hypothesized that in most test cases in which the invisibility of a widget $w$ is asserted, $w$ is visible in some other app state that is visited during the execution of the scenario under test. Thus, for each *widget_invisible* oracle in the source test, the existence of its associated widget $w$ is checked in all states of the source app that are visited during the migration process. If $w$ is found in any state $S$ on the source app, *UI Widget Mapper* will search for its equivalent widget $w'$ in the equivalent state $S'$ on the destination platform. After mapping the widget $w$, all remaining elements of the oracle event are mapped to the target platform. If $w$ is not found in any UI state of the source app that is visited during the migration process, its corresponding oracle event will be marked as "not mappable" on the target test.

The *assert_equal(w, attr, val)* oracle checks whether the value of the attribute $attr$ of widget $w$ is equal to the asserted value $val$. In this case, MAPIT captures the combination of $attr$ and $val$ parameters as oracle event's element $t$. Transferring this oracle type is challenged by the differences in the widget attributes maintained by iOS and Android. The only attribute that is directly mappable is `enabled`, since it exists on both iOS and Android. Some of the widget attributes have different names but equivalent meanings on the two platforms. This includes the attributes forming the previously discussed textual data and their corresponding mappings, and also a mapping between `visible` attribute on Android and `displayed` on iOS. Another group of attributes can be mapped using heuristics. For instance, the `selected` and `checked` attributes assess whether a widget (e.g., radio button, tab) is selected/checked on Android. Although these two attributes do not exist on iOS, developers usually denote a widget being selected/checked by assigning 1 to its `value` attribute or including the word "selected" in its `accessibility id`.

For other attributes, finding a mapping is not possible. MAPIT transfers any such *assert_equal(w, attr, val)* oracles to a `not-mappable` event on the destination platform. For example, the `clickable` attribute is exists on Android but no corresponding attribute for iOS widgets provides the same information.

*System Event Mapper*

For system events, MAPIT supports the action types `enter` and `back`. For these events, only the action type $a$ in the 4-tuple ($a$, $w$, $t$, $o$) has a value. If $a$ is `enter`, it will remain unchanged between the source and target events. The case is more complex if $a$ is `back`. The challenge is that `back` is a system event on Android, but no corresponding system event exists on iOS. Instead, iOS's equivalent would be a UI event with a widget that appears as a "back" button on the app screen. We discuss how MAPIT handles this event's mapping in both directions.

When mapping from iOS to Android, MAPIT only has to check whether the source event is indeed a `back` event. This is done by checking whether the source event's action type $a$ is `click`, and whether the textual attributes of its widget $w$ contain the keywords ``back'' or ``previous''.

The mapping from Android to iOS is more challenging since it requires relating a system event to a UI event, where the latter contains information not present in the former. To address this, MAPIT internally introduces a *virtual* `click` event and an associated back-button widget. This widget contains the bitmap of a typical back button, with coordinates normalized by device size, and relevant textual data to describe the button (e.g., the string ``back button'' as the value of the `content descriptor` attribute). The normalized coordinates are calculated based on the observation that, in most iOS apps, the back button is located in the bottom-left corner of the screen. Finally, the corresponding back button on iOS is identified by mapping this virtual widget using *UI Widget Mapper* as discussed previously.

*Test Generator*

Finally, the *Test Generator* component generates a UI test for the target platform based on the mapped events. The generated test is in the *Internal Test* format discussed in Section III-A.

Generating tests is particularly challenging when the UI widget $w'$ in a mapped event $(a', w', t', o')$ does not contain an attribute that can be used as a `widget locator` (e.g., `accessibility id` or `resource id`). This information is needed to identify a specific UI widget to trigger an event. In such cases, MAPIT needs to generate a `locator` that is understandable to the target device.

There are two options to locate the target widget: (1) using the widget's `coordinates` or (2) using the widget's `XPath`, which is the ancestral path from the root of the UI layout hierarchy [30]. We choose `XPath` because `coordinates` are tied to a specific device, which would result in brittle tests that are not executable on other devices. To automatically generate the `XPath` locator for a target widget, *Test Generator* leverages the UI layout hierarchy graph (recall Figure 5), which is stored as part of the target app's UITM (Figure 6). Specifically, *Test Generator* traverses this graph until the mapped event $(a', w', t', o')$ is reached, and retrieves the `XPath` associated with the target widget $w'$. The `XPath` is then stored as the mapped event's `locator` using *Internal Test*'s representation discussed in Section III-A and depicted in Figure 3-b.

## IV. Evaluation

Our evaluation focuses on two key aspects of MAPIT: (1) its accuracy in mapping events from the source to the target platform and (2) the usefulness of the tests it transfers. In this section, we first describe our empirical setup and then present the evaluation of MAPIT's two aspects.

### A. Evaluation Setup

MAPIT is not tied to any specific Android or iOS version or device, and is only practically constrained by the tools on which its implementation currently relies. Our evaluation was performed on an iPhone 7 running iOS 14.4, and a Pixel 4

emulator running Android 11.0 installed on a macOS laptop with 16GB RAM and 3.5GHz dual-core core i7 processor.

As discussed in the context of *UI Event Mapper* in Section III-B, MAPIT has two adjustable parameters: (1) textual similarity threshold and (2) proximity range. We empirically determined the best performing values for these parameters to be 0.5 and 0.1, respectively. Our results reported in this section were obtained using these values.

Recall that, unlike the lone existing approach for cross-platform test transfer [10], MAPIT does not require the apps' source code. This enables our evaluation to rely on widely-used apps, many of which are closed-source. Since MAPIT targets the bi-directional transfer of the *same* app across different platforms, we selected popular apps that are available on both Android and iOS. We first chose five different app categories: News, Shopping, ToDo List, Web Browser, and Mail Client. We chose these categories for two reasons: (1) they have a number of sibling apps on iOS's App Store and Android's Google Play, and (2) apps from these categories have been used for evaluating test migration techniques previously [8], [3]. In each app category, we selected five apps that are available on both platforms and are among the most downloaded apps, totaling 25 apps for each platform, as shown in Table II.

Table III shows the test scenarios we used to evaluate MAPIT. We selected the most common scenarios for each app category as identified by prior work [8], [3] and subsequently expanded the scenarios by further examining the subject apps. For each app, we evaluated four test scenarios. Note that there are more than four scenarios in the News, Shopping, and Mail Client categories because a given scenario may not be applicable to all subject apps. Within each scenario, we also identified a set of oracles, i.e., conditions that must hold true in the app at a given point.

To generate the test scripts for both Android and iOS platforms, we manually trigger the events, including both UI and system events, in each test scenario, and used Appium [14] to record the process. Appium automatically converts the recorded test scenarios to test scripts in the Robot framework format [15]. These test scripts serve as both source tests to be transferred from, and ground-truth tests to evaluate MAPIT's accuracy and usefulness. We manually added oracle events to the tests after they were translated to MAPIT's internal format during the *Source Data Extraction* phase (recall Section III-A).

We used MAPIT to transfer each test script from Android to iOS and vice-versa. This yielded 200 transfer cases in total (25 apps $\times$ 4 tests $\times$ 2 directions). Overall, our tests contain 828 UI events, 176 oracle events and 50 system events. This averages to slightly over 5 events per test. Overall, the choices we made in evaluating MAPIT (number of apps, tests, oracles, and test sizes) are at least comparable to, and in several instances significantly surpass, those reported in the emerging literature on mobile-app test transfer [3].

| **News** | **Shopping** | **ToDo List** | **Web Browser** | **Mail Client** |
|---|---|---|---|---|
| BBC | Wish | Google Tasks | Chrome | Gmail |
| CCN | Etsy | Microsoft To-Do | Firefox | Blue Mail |
| ABC News | ebay | Todoist | DuckDuckGo | Edison Mail |
| The Guardian | Poshmark | Any.do | Brave | Spark Mail |
| USA Today | AliExpress | My Tasks | Edge | Newton |

TABLE II: Subject apps used for MAPIT's evaluation.

| Category | Test Scenario |
|---|---|
| News | 1) Save or bookmark specific news article<br>2) Navigate to specific category of news<br>3) Search for specific news topic<br>4) Personalize newsfeed based on news topics<br>5) Change edition<br>6) Follow author |
| Shop | 1) Login to user account<br>2) Remove item from shopping cart<br>3) Navigate between product categories<br>4) Add item to shopping cart<br>5) Make wishlist<br>6) Filter products |
| ToDo | 1) Add ToDo task<br>2) Remove ToDo task<br>3) Edit ToDo task<br>4) Change due date of ToDo task |
| Web | 1) Access website by URL<br>2) Navigate to previous page<br>3) Navigate to new browser tab<br>4) Bookmark URL |
| Mail Client | 1) Compose email<br>2) Search email by keyword<br>3) Move emails across folders<br>4) Archive existing email<br>5) Reply to email |

TABLE III: The evaluated scenarios for each app category.

### B. Accuracy of Event Mappings

This part of our evaluation focuses on MAPIT's ability to correctly transfer a given app, oracle, or system event $e$ from a source test containing $e$ to the corresponding event $e'$ in the target test. Specifically, this reflects the accuracy of *Event Mapper*, MAPIT's core component (recall Section III-B).

Measuring *Event Mapper*'s accuracy requires that we isolate its impact from MAPIT's remaining components. To explain how we accomplish that, consider the following scenario. $S_i$ is a state in the source app and $S'_j$ its equivalent state in the target app. Event $e_{i+1}$ is an event in the extracted *UITM* representing a source test that takes the source app from $S_i$ to $S_{i+1}$ (recall Figure 6) . We evaluate whether *Event Mapper* is successful in finding the correct mapping for $e_{i+1}$ that will advance the target app from state $S'_j$ to $S'_{j+1}$.

To this end, we manually inspect each pair of sibling apps and detect their equivalent states for each test scenario based on the functionality they provide. We feed those states alongside the source event to *Event Mapper*. Manually detecting equivalent states in sibling apps was straightforward in practice: in more than 95% of the cases in our subject apps, there existed one-to-one mappings between the source and target states, and they occurred in the same order (i.e., $i = j$). This is consistent with our guiding hypothesis that sibling apps will have highly similar functionalities by design.

The correctness of each source event's mapping is determined by manually comparing the transferred test and the ground-

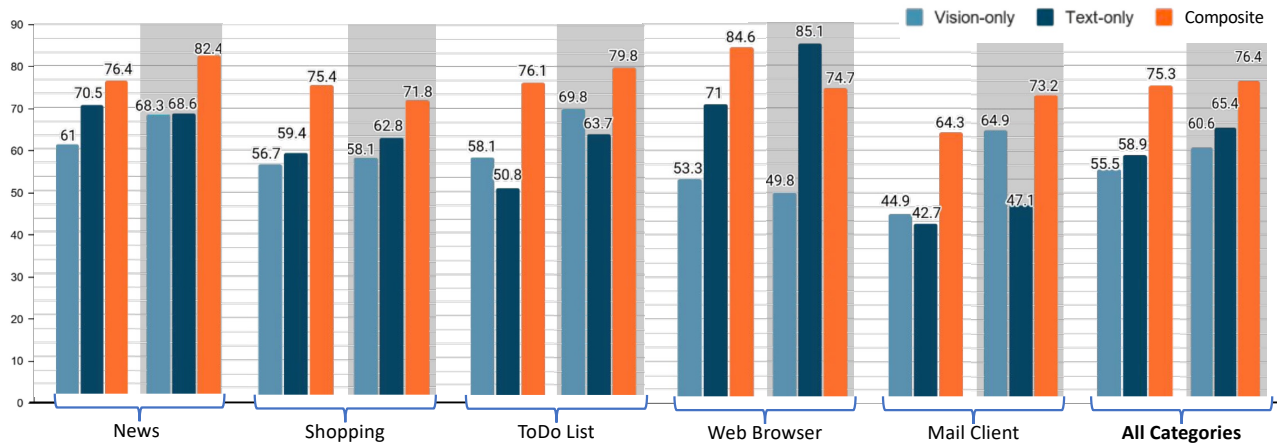


Fig. 8: The accuracy of MAPIT's event mapping across different app categories. Within each category, there are two clusters of results: the left (unhighlighted) cluster represent the mapping from Android to iOS and the right (highlighted) from iOS to Android. Within each cluster, the results are divided by mapping strategy: vision-only (left), text-only (middle), and composite (right).

truth test. Note that there can be multiple correct mappings for a given source event. For example, the correct mapping of the click event on widget $a3$ in Figure 1 can be a click on either $b3$ or $b4$ since they both result in the same action.

MAPIT's test transfer approach assumes that there exists one-to-one mappings between UI states of the sibling apps. However, in certain, rare cases the numbers of events that represent the same test scenario will differ between the two platforms. For example, Etsy's login on iOS requires the user to choose the account type first and then navigate to the main login page, whereas on Android the user chooses the account type on the login page itself. In the 100 test scenarios used in our evaluation, we encountered only 8 such cases; this prevalence (8%) is consistent with previously reported results [10]. Such differences do not impact MAPIT's event-mapping accuracy. Instead, mismatched UI states affect the *usefulness* of transferred tests and are taken into account in Section IV-C below.

Recall from Section III that MAPIT introduces a combination of visual and textual techniques for mapping events between platforms. We thus also evaluate the benefits of this composite mapping. Note that the sole previously existing cross-platform test migration technique, TestMig, only employs textual mapping [10]. Adding visual information was important in our case since we target closed-source apps whose textual information is limited due to the unavailability of app code. This was also the reason we were unable to directly compare MAPIT with TestMig: in addition to only supporting uni-directional transfer from iOS to Android, TestMig requires access to app source code.

Figure 8 shows the accuracy of our bi-directional event mapping in each app category, based on vision-only, text-only, and composite mappings. With one exception, MAPIT's composite mapper outperforms the other two strategies. The composite mapper was able to accurately map events in over $\frac{3}{4}$ of all cases (see "All Categories" in Figure 8). No notable differences in accuracy emerged when mapping UI, oracle, or system events. Furthermore, these results are independent of the platform: the overall results are separated by a single percentage point between the Android-to-iOS and iOS-to-Android mappings.

The lone exception to the above trends is the iOS-to-Android mapping of Web Browser apps. Our subsequent analysis uncovered a likely reason and a possible remedy. Namely, in MAPIT's analysis of some of the browser apps, a textually mapped widget would have been the correct widget to select, but its textual similarity with the source widget was below the threshold discussed in Section III-B. In those cases, the visually mapped widget was chosen per the strategy adopted by MAPIT. However, this was a flawed strategy because of the sibling apps' layout differences. This suggests that the thresholds, which we set across all apps, may need to be further tuned for different app categories and possibly based on other criteria.

### C. Usefulness of Transferred Tests

To assess how useful the tests transferred by MAPIT are, we leverage a metric introduced by recent work [3], which measurs the reduction in the manual effort required to accomplish a test-transfer task. Specifically, the manual effort required after using MAPIT is quantified as the number of steps needed to rectify the incorrectly-mapped events in the transferred test, while the effort without using MAPIT amounts to the number of steps needed to write the entire ground-truth test from scratch. A step can be an event's insertion, deletion, or substitution [3].

To perform this evaluation, we provide MAPIT with the binaries of sibling apps as well as a test script on the source platform, and compare the transferred test to the ground truth on the target platform. Recall from Section IV-B that there may exist multiple correct mappings for a source event. For this reason, we manually inspect each transferred test to verify the correctly mapped events.

Figure 9, shows the average effort reduction across the different subject-app categories. Overall, MAPIT reduces more than half the manual effort required to write UI tests for a new platform. Furthermore, the average reductions are similar in the two transfer directions (Android-to-iOS and iOS-to-Android), indicating that MAPIT's usefulness is independent of the platform. In our evaluation, MAPIT was able to achieve 100% reduction—eliminating all manual effort—in 58 of 200 test cases (29%).

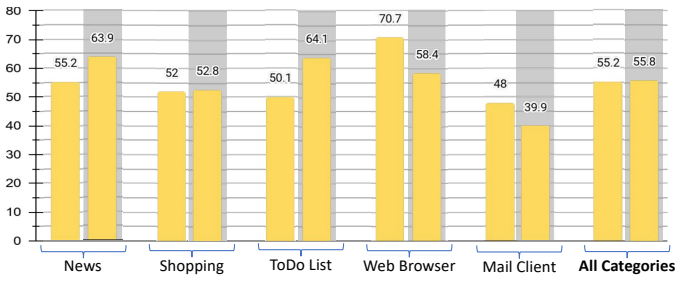


Fig. 9: Effort reduction afforded by MAPIT. Result pairs within each app category correspond to the mappings from Android to iOS (left) and from iOS to Android (right, highlighted).

If we consider these results in tandem with those from Figure 8, it is interesting to note that, in a number of instances, MAPIT achieved high accuracy but relatively low reduction (i.e., usefulness). Initially, this seemed counter-intuitive since, in principle, accurate event mappings should result in high-quality transferred tests. However, a more detailed analysis uncovered that the incorrectly-mapped events in these cases are rare, but they start appearing relatively early in a transferred test. In turn, this leads a target app into an incorrect state early during the test migration phase and causes it to "get lost" so that all subsequent source events are also mapped incorrectly.

In a great majority of cases, these subsequent events would have been mapped correctly if the app were in the correct state (as can be confirmed by MAPIT's complete accuracy data [13]). In fact, 74 of the 200 test cases (37%) would only have one incorrectly-mapped event if a correct app state were reached. This strongly suggests that minor human effort has the potential to improve MAPIT's usefulness significantly. For example, slightly "nudging" MAPIT in certain cases—by manually providing a correct mapping of a single event or by guiding the target app once to a correct state—would combine these 74 cases with the 58 fully transferred test cases to raise MAPIT's reduction in effort to nearly zero in $\frac{2}{3}$ of cases.

## V. Related Work

TestMig [10] is the only existing approach for *migrating tests* across platforms. However, as discussed earlier, TestMig only transfers tests from iOS to Android and requires the source code of both source and target apps. Other approaches in the mobile-app domain have focused on migrating UI tests between different Android apps within the same category. Behrang et al. [5], [6] and Lin et al. [8] rely on static analysis for extracting the GUI models of the app, which are not available for closed-source iOS apps. Hu et al. [7] proposed an ML-based approach for generating UI tests for an app using a library of existing tests. This work generates regression tests for a specific app rather than enabling test migration. Zhao et al. [3] proposed a framework for automatically evaluating the previous approaches, but did not address test migration. Beyond the mobile-app arena, Rau et al. [31] proposed an approach for efficiently generating UI tests by learning from the existing tests of other apps, but their work targets web applications. Similarly, Yeh et al. [32] proposed an early image-based platform-independent testing tool for testing desktop and web applications. Finally, Mariani et al. [33] proposed an approach that automatically exploits the common functionalities of Java applications to generate UI tests.

Another related body of work focuses on *remote app execution*. Yu et al. [34] proposed a record-and-replay technique for executing scenarios across mobile platforms. Their approach is based on image feature matching and UI layout characterization. Another recent approach [35], [36] enabled remote interaction with iOS devices and dynamic extraction of partial app UI models. These approaches do not generate a test case from an existing UI test, but replay on a target device a specific scenario that was recorded on a source device. Cross-platform test migration is different from them in at least three important ways. (1) Actual test cases must be human readable and modifiable. (2) Migrated test cases include oracle events, which may in fact benefit record-and-replay but are not considered by it. (3) Migrated test cases are not device coordinate-dependent and can be directly reused across devices on the same platform.

There is an emerging body of work that *extracts UI models* from Android apps and uses them to guide testing. This may be done statically [37], dynamically [2], [38], or by a combined strategy [39], [40]. Our approach is platform-independent and is, in principle, closer to dynamic approaches that do not assume the existence of app code.

## VI. Discussion and Conclusion

Our work has demonstrated that it is viable to flexibly transfer both individual app events and entire UI tests across mobile platforms. This will serve as a foundation for a range of follow-on activities in this area. Several of those will, naturally, focus on improving MAPIT's accuracy and usefulness, and on addressing its current shortcomings. This may involve relaxing some of our assumptions, such as taking advantage of code when it is available. It may also involve leveraging MAPIT's modular architecture to introduce platform- or technology-specific components when appropriate. Future work will also require overcoming specific challenges that have not been our focus to date for practical reasons.

One such challenge is presented by sibling apps whose events are not related one-to-one. Even though both our evaluation and prior work indicate that such cases occur relatively infrequently, addressing them would further improve MAPIT's utility. Another challenge is the inability to extract data (e.g., MAPIT's *UITM* discussed in Section III-A) from certain commercial apps. Our analysis of this problem identified four potential reasons: (1) some apps may use obfuscation and make certain UI elements inaccessible; (2) hybrid apps that combine web and native code may not be analyzable by "one size fits all" tools such as Appium; (3) certain UI states may not allow data extraction for security reasons (e.g., no screenshots may be taken on the login screen); and (4) some tests' execution requires currently unrecognized types of action (e.g., scrolling). Some of these (e.g., adding support for scrolling) are straightforward extensions to MAPIT, but others (e.g., overcoming obfuscation or security-driven restrictions) present compelling research challenges.

## REFERENCES


[1] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 738–748.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE software*, vol. 32, no. 5, pp. 53–59, 2014.

[3] Y. Zhao, J. Chen, A. Sejfia, M. Schmitt Laser, J. Zhang, F. Sarro, M. Harman, and N. Medvidovic, "Fruiter: a framework for evaluating ui test reuse," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1190–1201.

[4] M. L. Vasquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" *arXiv preprint arXiv:1801.06268*, 2018.

[5] 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*, 2018, pp. 164–175.

[6] ——, "Test migration between mobile apps with similar functionality," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 54–65.

[7] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable ui tests," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 269–282.

[8] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 42–53.

[9] A. Rau, J. Hotzkow, and A. Zeller, "Transferring tests across web applications," in *International Conference on Web Engineering*. Springer, 2018, pp. 50–64.

[10] X. Qin, H. Zhong, and X. Wang, "Testmig: Migrating gui test cases from ios to android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 284–295.

[11] "Soot," 2021. [Online]. Available: http://soot-oss.github.io/soot/

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

[13] [Online]. Available: https://figshare.com/s/a71c59d0c9c6e745dfc7

[14] "Appium," 2021. [Online]. Available: https://appium.io/

[15] "Robot framework," 2021. [Online]. Available: https://https://robotframework.org/

[16] "Android debug bridge," 2021. [Online]. Available: https://developer.android.com/studio/command-line/adb/

[17] "Web driver agent," 2021. [Online]. Available: https://github.com/facebookarchive/WebDriverAgent

[18] "Page source," 2021. [Online]. Available: https://www.google.com/search?q=page+source+appium&oq=page+source+appium&aqs=chrome.0.69i59j0i22i30.1929j0j7&sourceid=chrome&ie=UTF-8

[19] "Tesseract," 2021. [Online]. Available: https://github.com/tesseract-ocr/tesseract

[20] "Euclidean distance," 2021. [Online]. Available: https://www.oxfordreference.com/view/10.1093/oi/authority.20110803095800175

[21] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International Conference on Computer Vision*, 2011, pp. 2564–2571.

[22] O. Chum, J. Philbin, A. Zisserman *et al.*, "Near duplicate image detection: Min-hash and tf-idf weighting." in *Bmvc*, vol. 810, 2008, pp. 812–815.

[23] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International conference on computer vision*. Ieee, 2011, pp. 2564–2571.

[24] "Hamming distance," 2021. [Online]. Available: https://www.oxfordreference.com/view/10.1093/oi/authority.20110803095918607

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

[26] "Nltk stop words," 2021. [Online]. Available: https://www.nltk.org/book/ch02.html

[27] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*, 2013.

[28] A. Aizawa, "An information-theoretic perspective of tf–idf measures," *Information Processing & Management*, vol. 39, no. 1, pp. 45–65, 2003.

[29] "Cosine similarity," 2021. [Online]. Available: https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/cosdist.htm

[30] "Xpath," 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/XPath

[31] 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)*. IEEE, 2018, pp. 370–371.

[32] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using gui screenshots for search and automation," in *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, 2009, pp. 183–192.

[33] L. Mariani, M. Pezzè, and D. Zuddas, "Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 280–290.

[34] S. Yu, C. Fang, Y. Yun, Y. Feng, and Z. Chen, "Layout and image recognition driving cross-platform automated mobile testing," *arXiv preprint arXiv:2008.05182*, 2020.

[35] N. Lukić, S. Talebipour, and N. Medvidović, "Airmochi–a tool for remotely controlling ios devices," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1273–1277.

[36] ——, "Remote control of ios devices via accessibility features," in *Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2020, pp. 35–40.

[37] 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*, 2014, pp. 143–153.

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

[39] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining android app usages for generating actionable gui-based execution scenarios," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 111–122.

[40] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.