Test Transfer Across Mobile Apps Through Semantic Mapping

Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek
School of Information and Computer Sciences
University of California, Irvine, USA
{junwel1, jabbarvr, malek}@uci.edu

***Abstract*—GUI-based testing has been primarily used to examine the functionality and usability of mobile apps. Despite the numerous GUI-based test input generation techniques proposed in the literature, these techniques are still limited by (1) lack of context-aware text inputs; (2) failing to generate expressive tests; and (3) absence of test oracles. To address these limitations, we propose CRAFTDROID, a framework that leverages information retrieval, along with static and dynamic analysis techniques, to extract the human knowledge from an existing test suite for one app and transfer the test cases and oracles to be used for testing other apps with the similar functionalities. Evaluation of CRAFTDROID on real-world commercial Android apps corroborates its effectiveness by achieving 73% precision and 90% recall on average for transferring both the GUI events and oracles. In addition, 75% of the attempted transfers successfully generated valid and feature-based tests for popular features among apps in the same category.**

***Index Terms*—Test transfer, test migration, GUI testing, natural language processing, semantic similarity**

I. INTRODUCTION

GUI testing is the primary way of examining the functionality and usability of mobile apps. To reduce the cost of manual GUI testing, many automated test input generation techniques have been proposed in the literature over the past years [1]–[19]. These techniques follow different exploration strategies, such as random, model-based, stochastic, or search-based, for generating inputs in order to achieve a pre-defined testing goal, e.g., maximizing code coverage or finding more crashes. Despite all these efforts to automate the GUI test input generation, several studies indicate that they are not widely adopted in practice and majority of the mobile app's testing is still manual [20]–[22]. There are three main reasons that limit the viability of these techniques:

(1) Lack of context-aware text inputs. Most of the state-of-the-art techniques use random input values or rely on the manual configurations for text inputs. However, *contextual* text inputs are critical to thoroughly test majority of the apps, e.g., city names for a navigation app, correct URLs for a browser app, and valid username/password for a mail client app. Without such meaningful inputs, exploration of the App Under Test (AUT) may get stuck at the very beginning and GUI states deep in the testing flow may never been exercised.

(2) Failing to generate expressive tests. Majority of the automated testing techniques aim to maximize the code coverage or reveal as many crashes as possible. The generated tests by such techniques are typically feature-irrelevant or unrepresentative of the canonical usages of apps [9], [23]. This lack of expressiveness makes debugging cumbersome, as such tests do not include the reproduction steps that can be organized by use cases or features [22].

(3) Absence of test oracles. Despite a few efforts for automatic generation of test oracles for mobile apps [15], [24], majority of the existing test generation tools are unable to identify failures other than crashes or run-time exceptions. Without automated test oracles, such tests cannot thoroughly verify correct behavior of the AUT.

To address these limitations, we propose CRAFTDROID, a framework to reuse an existing test suite for one app to test other similar apps. CRAFTDROID is inspired by recent work from Behrang and Orso [25] and Rau et al. [26], which provided initial evidence of the feasibility of test transfer for mobile apps and web applications, respectively. Like their works, our proposed technique transfers available test cases corresponding to a specific feature or use-case scenario of one app to other apps with similar functionality. However, unlike their work, CRAFTDROID is also able to transfer the test oracles, if they exist. To enable context-awareness for text inputs, CRAFTDROID relies on information retrieval techniques to extract the human knowledge from an existing test suite and reuse it for other apps. Since test transfer is across apps with similar functionalities/features, the generated tests using CRAFTDROID are inherently feature-relevant and expressive. As CRAFTDROID not only transfers test inputs, but also oracles (assertions), it is able to thoroughly verify correct behavior of the AUT.

Two insights from the prior literature [27], [28] form the foundation of our work. First, apps within the same category share similar functionalities. For example, shopping apps should implement user registration and authentication to provide personalized services. As another example, web browsers should implement common features such as browsing, adding/removing tabs, or bookmarking URLs, despite different strategies they take for enabling privacy. Second, GUI interfaces for the same functionality are usually semantically similar, even if they belong to different apps with different looks and styles. By semantic similarity, we mean the conceptual relation between the textual information, e.g., the text, adjacent labels, or variable names, which can be retrieved from actionable GUI widgets such as buttons, input fields, or checkboxes. For instance, a button to start the registration process on an app can appear with text "Join", "Sign Up", or "Create Account". Even if the texts are syntactically different, they are semantically related. As another example, a "Confirm and Pay" button on a shopping app for checkout can be a

”Place Order” button on another shopping app.

Extensive evaluation of CRAFTDROID on real-world commercial and open-source Android apps collected from various categories on Google Play, including popular apps such as Wish, Yelp, and Firefox Focus, confirms effectiveness of the proposed approach. In fact, 75% of the attempted transfers by CRAFTDROID successfully generated valid and feature-based test cases, with 73% precision and 90% recall on average for the transferred GUI events and oracles. This paper makes the following contributions:

- A novel technique for transferring both test inputs and oracles across mobile apps through semantic mapping of actionable GUI widgets.
- An implementation of the proposed framework for Android apps, which is publicly available [29].
- Empirical evaluation on real-world apps demonstrating the utility of CRAFTDROID to successfully transfer tests across mobile apps.

The remainder of this paper is organized as follows. Section II introduces a motivating example that is used to describe our research. Section III provides an overview of our framework and Sections IV-VI describe the details of the proposed technique. Section VII presents the evaluation results. The paper concludes with a discussion of the related research and avenues for future work.

II. MOTIVATING EXAMPLE

To illustrate how CRAFTDROID works, consider Rainbow Shops [30], a shopping app for women clothing, and Yelp [31], a local-search app for services and restaurants. Figures 2 and 3 show the registration process on Rainbow Shops and Yelp, respectively. To register a new account on Rainbow Shops, user starts by clicking on the “Join” button (Figure 2-a), which directs the user to a registration form (Figure 2-b). By filling the required fields of registration form and clicking on the “Create Account” button, Rainbow Shops creates an account for the user and moves to the profile page (Figure 2-c), which shows information about user, such as her username, i.e., Sealbot.

To initiate the registration process on Yelp, the testing flow starts by clicking on the profile tab, denoted by “Me” in Figure 3-a. Then, the user should navigate through several screens to provide required registration information (Figures 3-d to 3-e). Finally, by clicking on the “Sign Up” button (Figure 3-f), the registration process is complete and Yelp moves to the profile page, where user can see her username, i.e., Sealbot (Figure 3-g).

While the overall registration process in these two apps follows the same steps—clicking on a button to start registration, filling the registration form, and submitting information—a direct copy of the test steps from Rainbow Shops to Yelp is not possible due to the following reasons: (1) The mapping of test steps between the two apps is not one-to-one. For example, to reach the registration form, Rainbow Shops requires only one click (Figure 2-a), while it takes three clicks in Yelp to reach the registration form (Figures 3-a, 3-b, and 3-c). As another example, a user provides personal information using two forms in Yelp compared to the one form in Rainbow Shops. (2) The mapping of GUI widgets is challenging, especially if they are syntactically different but semantically similar. For example, the clicked buttons in these two test flows are different in terms of their label, i.e., “Join” in Rainbow Shops and “Sign Up” in Yelp.

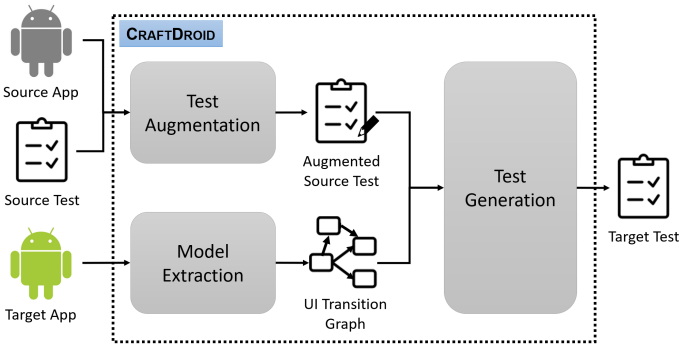

Fig. 1: Overview of CRAFTDROID

Despite these challenges, CRAFTDROID is able to transfer a test case that verifies the registration process in Rainbow Shops to Yelp by semantically mapping their GUI widgets. In the following sections, we describe the details of how CRAFTDROID identifies the matches and transfers GUI/oracle events from Rainbow Shops to Yelp.

III. APPROACH OVERVIEW

Figure 1 provides an overview of CRAFTDROID consisting of three major components: (1) *Test Augmentation* component that augments test cases available for an existing app, i.e., *source app*, with the information extracted from its GUI widgets that are exercised during test execution, (2) *Model Extraction* component that uses program analysis techniques to retrieve the GUI widgets and identify transitions between Activity components of a target app, and (3) *Test Generation* component that leverages Natural Language Processing (NLP) techniques to compute similarity between GUI widgets of the source and target apps to transfer tests.

More specifically, CRAFTDROID takes an existing mobile app and its test case as input. It then instruments, executes, and augments the source test with textual information retrieved from the GUI widgets it exercised during its execution. Afterwards, CRAFTDROID statically analyzes the target app to extract its *UI Transition Graph (UITG)*. Finally, CRAFTDROID uses UITG of the target app to search for widgets that are similar to those found in the source app to generate a new test. It leverages NLP techniques, such as word embedding, to compute the similarity between GUI widgets in the source and target apps. Regarding the transfer of oracle, CRAFTDROID is able to deal with several types of oracles that are commonly used in practice, including negative ones such as nonexistence check of text. We will describe these three components in more detail in the following sections.

IV. TEST AUGMENTATION

Algorithm 1 shows how *Test Augmentation* component works. It takes the source app, $srcApp$, with an existing test case, t, as input, and generates an augmented test case t', which contains textual meta-data related to the GUI widgets

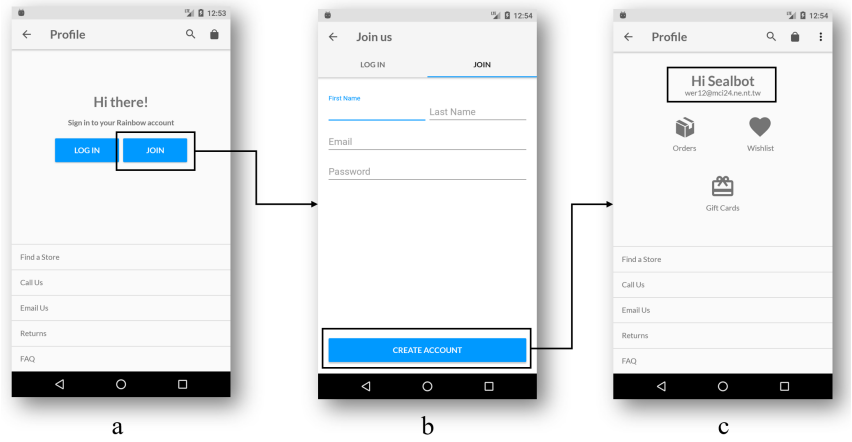

Fig. 2: Excerpted registration process on Rainbow Shops

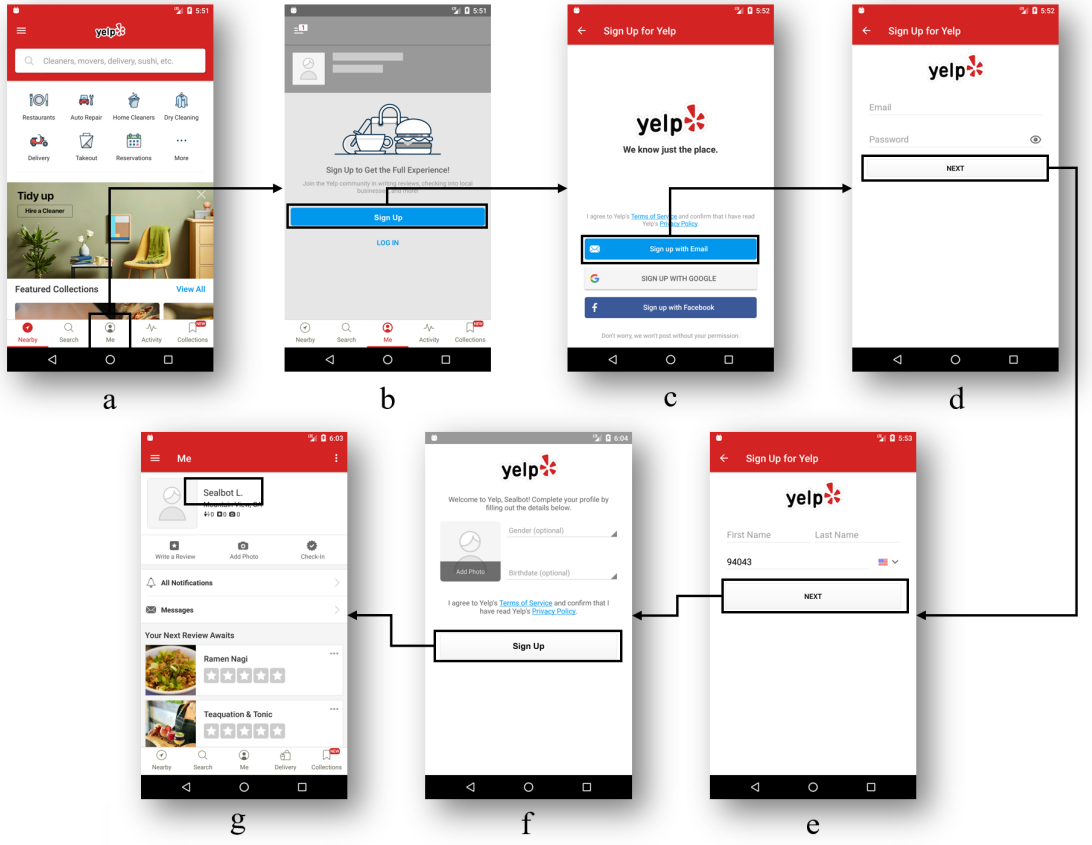

Fig. 3: Excerpted registration process on Yelp

that are exercised by t. We formulate test case t as a set of *GUI events* $\{(w_1, a_1), (w_2, a_2), ...\}$, where w_i is a GUI widget, e.g., Button, and a_i is the action performed on w_i, e.g., *click*. An action a_i can be a single operation such as *click* or an operation with arguments such as *swipe* that contains starting and ending coordinates. If a test comes with an oracle, CRAFTDROID identifies it as a special type of event (w_i, a_i), where a_i is an assertion, e.g., *assertEqual*. If the assertion is widget-specific, e.g., existence check of a widget, w_i denotes the widget to be checked. On the other hand, if the assertion is widget-irrelevant, e.g., existence of certain text on the screen, w_i is set to be empty.

Algorithm 1 starts by initializing variables (Line 1) and launching the source app (Line 2). For each GUI or oracle event (w_i, a_i) in t, *Test Augmentation* component dynamically analyzes current screen to retrieve required textual information of w_i, such as the *resource-id*, *text*, and *content-desc*. To that end, it uses adb tool [32] to dump current screen, i.e., an XML file of current UI hierarchy (Line 4), and parses the XML file (Line 5). Algorithm 1 updates w_i with textual information to

Algorithm 1 Test Augmentation

Input:
A source app $srcApp$;
A test case $t = \{(w_1, a_1), (w_2, a_2), ...\}$ for $srcApp$
Output:
An augmented test case $t' = \{(w'_1, a_1), (w'_2, a_2), ...\}$

1: $t' = \varnothing$;
2: $launchApp(srcApp)$
3: **for each** $(w_i, a_i) \in t$ **do**
4: $screen = dumpCurrentState()$
5: $info = getExtraInfo(w_i, screen)$
6: $w'_i = augment(w_i, info)$
7: $t' = t' \cup (w'_i, a_i)$
8: $execute(w_i, a_i)$
9: **end for**
10: **return** t'

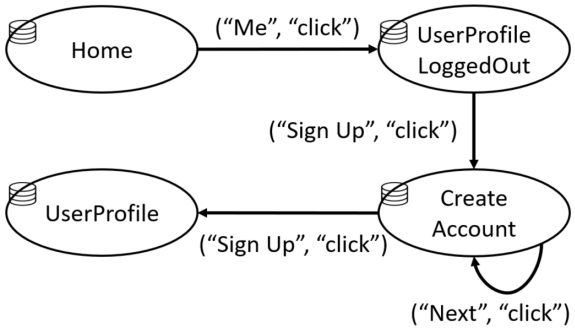

Fig. 4: Excerpted UI Transition Graph for Yelp

produce augmented widget w'_i and adds it to the augmented test (Line 6-7). Finally, it executes w_i (Line 8) to move to the next widget.

V. Model Extraction

Model Extraction component statically analyzes the target app, $targetApp$, to generate a model called *UI Transition Graph* (UITG). This model represents how Activities/Fragments of an app interact with each other through invoking GUI widgets' event handlers. UITG will later be used by *Test Generation* component to search for a match for a given widget of source app in the target app.

At a high level, UITG represents Activity components comprising the target app as nodes and GUI events as transitions among the nodes. Each node of UITG in turn contains a list of widgets that can be rendered directly through the Activity, or indirectly through Fragments comprising the Activity. It is important to consider Fragments, since an Activity may consist of several Fragments, which can be called from different Activities. Figure 4 shows a partial *UITG* for Yelp. As demonstrated by this UITG, clicking on the "Me" widget transfers users from the *Home* Activity to the *UserProfileLoggedOut Activity*.

Model Extraction constructs the UITG in two steps:

(1) Extracting Activities, Fragments, and their corresponding GUI widgets. *Model Extraction* parses *Manifest* file of the target app to collect a list of Activity components. For each identified Activity, it then extracts all of the GUI widgets, e.g., Button, EditText, and TextView, that it renders during execution of the app. These widgets are either implemented by the Activity itself or inside Fragments within the Activity.

To extract widget list, *Model Extraction* analyzes XML-based meta-data (*Resource* files)—for statically defined GUI widgets—as well as the source code—for dynamically defined ones. More specifically, to get the list of statically defined GUI widgets, *Model Extraction* first refers to the source code of each Activity/Fragment and looks for specific methods, such as *setContentView()* and *findViewById()*, to identify resource files corresponding to widgets. It then adds all the widgets identified in the resource file to the widget list of the Activity. To get the list of dynamically defined GUI widgets, *Model Extraction* analyzes the source code of Activity/Fragment components to identify initialization of GUI widget elements in them and adds the corresponding widgets to the widget list. During extraction of widgets, *Model Extraction* also retrieves and stores their corresponding textual information.

(2) Identifying transitions between Activities. *Model Extraction* starts from the launcher Activity, which is specifically identified in the *Manifest* file. For each Activity, it analyzes the event handlers of all the GUI widgets in the Activity's widget list, e.g., *onClick()* for a button or *onCheckedChanged()* for a check box. If the event handler of a widget invokes specific methods that result in transition to another Activity (e.g., *startActivity()*) or Fragment (e.g., *beginTransaction()*), *Model Extraction* includes a transition between the two Activity Components. We identify two types of transitions in UITG:

1) *Inter-component transition*. The method call results in a transfer of control between two distinct Activities. For example, when user clicks on the "Me" tab in Figure 3-a, the *onClick()* handler of this widget initiates an Intent message and invokes *startActivity()* method to transfer the control to *UserProfileLoggedOut* Activity.
2) *Intra-component transition*. The method call to a GUI event handler results in a transition back to the same Activity. Such transitions happen when an Activity consists of multiple Fragments and performing an action on one Fragment results in transition to another Fragment within the same Activity. For instance, Figures 3-d and 3-e represent two Fragments related to the *CreateAccount Activity*. Clicking on the "Next" button on the first Fragment moves the control to the second Fragment. Thereby, this transition causes a loop in the UITG, as shown in Figure 4.

After generation of *UITG*, *Model Extraction* component combines the widgets collected for all nodes into an associative array, denoted as map. This construct maps all the GUI widgets in $targetApp$ to the corresponding Activity/Fragment that can render them during execution of app.

VI. Test Generation

Algorithm 2 demonstrates how the *Test Generation* component of CraftDroid works. This component takes the $targetApp$, its corresponding *UITG* and widget *map*, and an augmented test for the $srcApp$, t', as input and generates a

Algorithm 2 Test Generation

Input:
$targetApp$,
$UITG$ of $targetApp$,
map widgets on each Activity/Frag. in the $targetApp$,
$t' = \{(w'_1, a_1), (w'_2, a_2), ...\}$ from $srcApp$,
Output:
$t_{new} = \{(w_{n_1}, a_{n_1}), (w_{n_2}, a_{n_2}), ...\}$ for $targetApp$

```
1:  while true do
2:      t_new = ∅
3:      for each (w'_i, a_i) ∈ t' do
4:          candidates = getCandidates(w'_i, map, UITG)
5:          for each w_n ∈ candidates do
6:              leadingEvents =
                    getLeadingEvents(w_n, UITG, map, t_new)
7:              if leadingEvents ≠ null then
8:                  a_n = generateAction(w'_i, a_i, w_n)
9:                  t_new = t_new ∪ leadingEvents ∪ (w_n, a_n)
10:                 break
11:             end if
12:         end for
13:     end for
14:     if Δ fitness(t_new) ≤ threshold or timeout
15:         break
16:     end if
17: end while
18: return t_new

19: function GETLEADINGEVENTS(w_n, UITG, map, t_new)
20:     execute(t_new)
21:     srcAct = getCurrentActivity()
22:     destAct = getActivity(w_n, map)
23:     paths = getPaths(srcAct, destAct, UITG)
24:     sort(paths)
25:     for each path ∈ paths do
26:         isValid = validate(w_n, path, map)
27:         if isValid = true then
28:             return path
29:         end if
30:     end for
31:     return null
32: end function
```

new test case t_{new} for $targetApp$ by transferring the GUI and oracle events of t'.

To that end, it iterates over every GUI or oracle event (w_i, a_i) in t' and collects a list of candidate widgets in $targetApp$, $candidates$, which are ranked based on their similarity to w_i (Line 4, details in Section VI-A). For each GUI widget w_n in $candidates$, Algorithm 2 checks to see if it is reachable, and if so, identifies a sequence of events—leading events—that should be executed to reach w_n (Line 6, details in Section VI-B).

For a reachable candidate w_n, Algorithm 2 identifies the appropriate action a_n (Line 8, details in Section VI-C), adds (w_n, a_n) along with the leading events to t_{new} (Line 9), and moves to the next w'_i to find its match (Line 10).

Once all the widgets in t' are checked for a match in $targetApp$, Algorithm 2 checks the termination criteria (Line 14, details in Section VI-D). If termination criteria are met, it terminates (Line 15). Otherwise, it repeats the whole process of transfer. The reason for repeating the test generation process is that *Test Generation* component relies on *UITG* to identify reachability of the candidate widgets. Since *UITG* is derived through static analysis, it is an over approximation of the app's runtime behavior. In addition, static analysis is not able to realize dynamically generated contents such as pop-up dialogues or buttons in Android's WebView. To overcome these limitations, CRAFTDROID executes $targetApp$ to determine reachability and updates *UITG* based on runtime information. Thereby, *Test Generation* repeats transfer with an updated *UITG* to increase the chance of successful transfer.

In the remainder of this section, we describe the key components of *Test Generation* in more detail.

A. Computing Similarity Score

In the `getCandidates` function (Line 4), *Test Generation* considers two factors to compute the similarity between widgets: (1) their corresponding textual information, and (2) their location in *UITG*. More specifically, to determine the similarity of a candidate widget w_n to source widget w'_i, *Test Generation* first computes $score_t$—a measure of how similar are the textual information of w_n to that of w'_i. It then normalizes $score_t$ based on how close w_n is to the current Activity by leveraging *UITG* to compute the final similarity value.

1) Computing textual similarity score, $score_t$: CRAFTDROID collects the textual information of a GUI widget from multiple sources, such as widget's attributes, the name of Activity/Fragment that renders it, and its immediate parent and siblings. CRAFTDROID follows a two step process to measure the textual similarity. It first retrieves raw textual data from different sources and processes them. It then utilizes the processed data to measure the similarity in a weighted scheme among all sources.

Text Processing. *Test Generation* processes the collected textual information by *Test Augmentation* and *Model Extraction* through applying a series of common practices in NLP, including tokenization and stopword removal. The result of this step is a set of word lists for every textual information. For example, textual information for the button *Sign Up* in Figure 3-b can have three word lists: (1) ["Sign","Up"] from its label, (2) ["sign", "up", "button"] from its resource-id of *sign_up_button*, and (3) ["user", "profile", "logged", "out"] from its Activity name of *UserProfileLoggedOut*.

Computing Textual Similarity. To determine $score_t$ between two GUI widgets w_n and w'_i, *Test Generation* computes the similarity score for each information source and then calculates a weighted sum of the individual scores. Since the previous step produces a set of word lists for each GUI widget, the problem of determining the textual similarity between two GUI widgets is dual to the problem of computing the similarity score between word lists.

CRAFTDROID leverages Word2Vec [33]—a model that captures the linguistic contexts of words—to compute the simi-

larity score between two word lists. That is, it first computes the cosine similarity for all possible word pairs in the word lists. Next, it identifies the best match among pairs based on two criteria: (1) the pair has the highest cosine similarity, and (2) every word is only matched once.

For instance, consider the *Create Account* button in Figure 2-b and *Sign Up* button in Figure 3-b from the motivating example. The two word lists corresponding to these buttons are ["Create", "Account"] and ["Sign", "Up"]. To compute the similarity score between them, the pairwise cosine similarity is calculated as follows:

$$\begin{array}{c} \\ Sign \\ Up \end{array} \begin{array}{c} \begin{array}{cc} Create & Account \end{array} \\ \begin{bmatrix} \mathbf{0.405} & 0.168 \\ 0.201 & \mathbf{0.158} \end{bmatrix} \end{array}$$

In this example, the word pairs that match the mentioned criteria are ("Create", "Sign") and ("Account", "Up") with cosine similarity of 0.405 and 0.158, respectively. Thereby, the final similarity score between these two word lists is calculated as $(0.405{+}0.158)/2 = 0.282$, which is the score for the *text* of these two buttons. Similarly, the two word lists corresponding to the *resource-id* of these two widgets are ["button", "sign", "up"] and ["sign", "up", "button"]. The cosine similarity for these lists are as follows:

$$\begin{array}{c} \\ sign \\ up \\ button \end{array} \begin{array}{c} \begin{array}{ccc} button & sign & up \end{array} \\ \begin{bmatrix} 0.117 & \mathbf{1.0} & 0.149 \\ 0.048 & 0.149 & \mathbf{1.0} \\ \mathbf{1.0} & 0.117 & 0.048 \end{bmatrix} \end{array}$$

Based on these values, the score for *resource-id* is calculated as $(1.0 + 1.0 + 1.0)/3 = 1.0$. If only these two information sources are considered to compute the similarity score, the final textual similarity between these two buttons is $(0.282 + 1.0)/2 = 0.641$.

2) Computing final similarity score: To compute the final similarity score between w_n and w_i, *Test Generation* normalizes $score_t$ based on the distance of w_n from current screen. *Test Generation* consults *UITG* to get the shortest distance d, i.e., number of GUI events, from the current screen to the Activity to which w_n belongs. It computes the final similarity as follows:

$$similarity(w_n, w_i') = \begin{cases} score_t, & \text{if } d = 0 \\ \dfrac{score_t}{1 + \log_2 d}, & \text{otherwise} \end{cases}$$

This adjustment assigns a higher priority to candidate GUI widgets that are closer to the current screen. This is because intuitively, the steps or events to test the same functionality should not be significantly different even in different apps. For example, consider the *Join* button from Figure 2-a in Rainbow Shops. The most semantically similar widget in Yelp app to this button is the *Sign Up* button, which appears in multiple UIs, e.g., Figures 3-b, 3-c, and 3-f. To identify which one of these buttons is the best match for *Join*, CRAFTDROID starts from the launcher Activity of Yelp, *Home* Activity, and finds the closest node in its *UITG* (Figure 4) that contains a *Sign Up* button, *UserProfileLoggedOut* Activity, which is shown in Figure 3-b.

B. Reachability Check

The function `getLeadingEvents` in Algorithm 2 checks the reachability of w_n, a candidate widget in $targetApp$ that can be matched to w_i'. If reachable, the function returns the GUI events leading to the Activity holding w_n. To that end, first t_{new}—series of GUI events successfully transferred so far—is executed and the last activity *srcAct* executed by t_{new} is identified (Line 21). The widget map is then used to pinpoint the Activity *destAct* that holds w_n (Line 22). Next, all the potential paths in *UITG* from $srcAct$ to $destAct$ are explored to derive sequences of GUI events—leading events—that execute each path (Line 23).

The identified paths are sorted based on their length (Line 24). This way, shorter paths have a higher chance of being selected, thereby making the length of final transferred test shorter, which is generally desirable for debugging purposes. The function `validate` then verifies whether w_n is reachable by executing actions corresponding to each $path$ on $targetApp$ (Line 26). The first path that verifies reachability of $destAct$ from $srcAct$ is returned as output (Lines 27-28). If no path is found or could be verified, `null` is returned (Line 31), indicating that w_n is not reachable.

Finally, it is worth mentioning that in addition to verifying the reachability of each path, function `validate` (1) updates *UITG* by removing invalid paths, i.e., unreachable paths, (2) updates the widget map by adding new GUI widgets that are encountered at runtime (i.e., those that are loaded dynamically), and (3) determines the correct screen for asserting negative oracles (details in Section VI-C).

C. Actions for the Transferred GUI and Oracle Events

Once a widget match w_n is found, Algorithm 2 determines the proper action a_n for it to successfully transfer (w_i', a_i) (Line 8). Based on the type of event, i.e., GUI or oracle event, Algorithm 2 identifies a_n as follows:

GUI event. Even when the type of matched GUI widgets in $srcApp$ and $targetApp$ are the same, their corresponding action might be different. For example, removing an item in a to-do list app can be performed by a swipe, while the same task in another to-do list app might be performed by a long click. To overcome this challenge, CRAFTDROID considers a series of possible actions for w_n and finds the one that properly works on w_n in $targetApp$. To that end, it first analyzes the source code of $targetApp$ to find a specific event listener, such as *onSwiped()* or *setOnLongClickListener()*, registered for the matched widget w_n, and returns a_n as the action corresponding to such an event listener. If no specific action can be identified, it reuses the same action in $srcApp$, i.e., assign $a_n = a_i$.

Oracle event. For oracle events (w_i', a_i) in $srcApp$, where a_i is an assertion, CRAFTDROID generates a_n for $targetApp$ based on whether a_i is widget-specific, e.g., existence check of a widget, or widget-irrelevant, e.g., existence check of text.

Table I lists the types of oracle events supported by the current version of CRAFTDROID. For widget-specific assertions, *Test Generation* modifies the assertion so that it matches the target widget, w_n. For example, when a_i checks if the *resource-id* of w_i' matches a specific value, the generated a_n

TABLE I: Types of oracle supported by CRAFTDROID. (w'_i, a_i): the source oracle event. (w_n, a_n): the transferred target oracle event.

a_i	a_n	Widget-specific?
$assertEqual(VALUE_i, attr(w'_i))$	$assertEqual(VALUE_n, attr(w_n))$	Y
$elementPresense(w'_i)$	$elementPresense(w_n)$	Y
$elementInvisible(w'_i)$	$elementInvisible(w_n)$	Y
$textPresense(STRING)$	$textPresense(STRING)$	N
$textInvisible(STRING)$	$textInvisible(STRING)$	N

should also check if the *resource-id* of w_n matches a specific value (First row in Table I). On the other hand, if a_i is widget-irrelevant, it can be directly transferred to $targetApp$.

Transferring negative oracle events, e.g., nonexistence of text, is challenging, as they can make the transferred test pass, regardless of the successful transfer of tests. For example, consider testing the functionality of removing a task from to-do list. To ensure that an item has been successfully deleted, the oracle could be a negative assertion of *textInvisible* to check non-existence of item's text. Suppose that we have a source app that removes a task without confirmation, while target app requires one additional step to get confirmation of removal from user before removing the task. An unsuccessful transfer of test that does not consider user confirmation in target app can still pass, since the negative assertion will be checked at the confirmation step, where the text of item is not visible, yet item is not deleted. Thereby, the main challenge of transferring negative oracles is to identify the correct screen for them to be executed.

A heuristic that allowed us to overcome this challenges is as follows: a negative oracle is likely to be asserted on the proper screen when its negation (i.e., positive oracle) is also asserted on that same screen, albeit with different content displayed on the screen. To find the correct screen for a negative oracle, CRAFTDROID uses *anchor widget*—an actionable widget that appears in the screen where both a negative oracle and negation of the negative oracle (i.e., positive oracle) should be asserted. The anchor widget serves as a reference to the correct screen. To identify an anchor widget, CRAFTDROID first negates the assertion of negative oracle and then searches for a screen where that assertion can be verified. Any actionable widget in that screen can be considered as the anchor widget. To that end, CRAFTDROID analyzes the source test, t'_i, before transfer and determines the negate of negative oracle, if one exists. During test transfer, it examines the negated assertion on all screens and selects an actionable widget in a screen that the negated assertion passes as an anchor widget[1].

In the example of to-do list apps, CRAFTDROID negates the negative oracle of text non-existence to existence, i.e., checks if the text of an item exists in the current screen. The anchor widget in this example could be an *Add* widget that is used to add items to a list. This is because existence of the text of a to-do item should be checked when that item is being added. Thereby, a widget for adding always exists in the screen that list items exist. Later for transfer of oracle event, CRAFTDROID leverages *UITG* to first navigate back to the screen, where the *Add* exists, and then transfers the oracle.

[1]CRAFTDROID uses anchor widget instead of Activity names, since Activity might have multiple Fragments. Thereby, just getting back to the Activity does not guarantee the screen is correct.

D. Termination Criteria

Algorithm 2 iteratively improves the quality of test transfer through updating *UITG* and the widget map. It terminates once the fitness of a transferred test cannot be improved any further, or a timeout value is reached. The fitness of a transferred test is the average of similarity values (Section VI-A) computed for its corresponding events.

VII. EVALUATION

We investigate the following research questions in our experimental evaluation of CRAFTDROID:

RQ1. How effective is CRAFTDROID in terms of the number of successful transfers compared to total attempted transfers? What are the precision and recall for attempted GUI and oracle transfers?

RQ2. What are the main reasons yielding transfer failure?

RQ3. How efficient is CRAFTDROID in terms of the running time to transfer tests from one app to another?

RQ4. What are the factors impacting the efficiency of CRAFTDROID?

A. Experimental Setup

We implemented CRAFTDROID with Python and Java for test cases written using Appium [34], which is an open source and cross-platform testing framework. Existing test cases for the subject apps are written using Appium's Python client and the augmented/generated test cases are stored in JSON format. The *Model Extraction* component is built on top of Soot, a static analysis framework for Java [35]. For our experiments, we used a Nexus 5X Emulators running Android 6.0 (API 23) installed on a Windows laptop with 2.8 GHz Intel Core i7 CPU and 32 GB RAM.

Subject apps. We evaluated the proposed technique using both open-source and commercial Android apps. CRAFTDROID is able to transfer tests for similar functionalities implemented differently on separate apps. Thereby, we performed test transfers among apps within the same category and for each category, identified main functionalities to be tested. To that end, we selected five categories that have large number of apps on Google Play, namely *Browser*, *To-Do List*, *Shopping*, *Mail Client*, and *Tip Calculator*. These five categories are often used in prior research that either studied common functionalities across mobile/web apps [26], [28], [36], [37] or proposed Android GUI testing solutions [11], [12], [38], [39]. Table II shows the list of 25 subjects and their categories.

For each category, we identified two main functionalities and the corresponding test steps. The test steps for each

TABLE II: Subject apps.

Category	App (version)	Source
a1-Browser	a11-Lightning (4.5.1)	F-Droid
	a12-Browser for Android (6.0)	Google Play
	a13-Privacy Browser (2.10)	F-Droid
	a14-FOSS Browser (5.8)	F-Droid
	a15-Firefox Focus (6.0)	Google Play
a2-To Do List	a21-Minimal (1.2)	F-Droid
	a22-Clear List (1.5.6)	F-Droid
	a23-To-Do List (2.1)	F-Droid
	a24-Simply Do (0.9.1)	F-Droid
	a25-Shopping List (0.10.1)	F-Droid
a3-Shopping	a31-Geek (2.3.7)	Google Play
	a32-Wish (4.22.6)	Google Play
	a33-Rainbow Shops (1.2.9)	Google Play
	a34-Etsy (5.6.0)	Google Play
	a35-Yelp (10.21.1)	Google Play
a4-Mail Client	a41-K-9 (5.403)	Google Play
	a42-Email mail box fast mail (1.12.20)	Google Play
	a43-Mail.Ru (7.5.0)	Google Play
	a44-myMail (7.5.0)	Google Play
	a45-Email App for Any Mail (6.6.0)	Google Play
a5-Tip Calculator	a51-Tip Calculator (1.1)	Google Play
	a52-Tip Calc (1.11)	Google Play
	a53-Simple Tip Calculator (1.2)	Google Play
	a54-Tip Calculator Plus (2.0)	Google Play
	a55-Free Tip Calculator (1.0.0.9)	Google Play

TABLE III: Test cases for the proposed functionalities.

Functionality	#Test Cases	Avg# Total Events	Avg# Oracle Events
b11-Access website by URL	5	3.4	1
b12-Back button	5	7.4	3
b21-Add task	5	4	1
b22-Remove task	5	6.8	2
b31-Registration	5	14.2	5
b32-Login with valid credentials	5	9	4
b41-Search email by keywords	5	5	3
b42-Send email with valid data	5	8	3
b51-Calculate total bill with tip	5	3.8	1
b52-Split bill	5	4.8	1
Total	50	6.6	2.4

functionality, which are listed in Table IV include at least one oracle step. The oracle steps are implemented as assertion and wait-until statements.

Test cases. To construct tests suites, we first collected tests for each subject app[2], if there were any, and then augmented the test suites with the test cases corresponding to the steps described in Table IV. The number of events for tests among different categories varies from 3 to 19, with an average of 6.6 events, including 2.4 oracle events [3].

Attempted transfers. For each test case validating a functionality of an app, CRAFTDROID transfers the test case to the other four apps under the same category. Thereby, the number of attempted transfers for each category are 5 (test cases) $\times$ 4 (transfers) $=$ 20, making the total number of attempted transfers for evaluating CRAFTDROID to be 200. After each transfer, we manually examined the test and its execution to identify *false positive*, *false negative*, and *true positive* cases as follows: *false positive* occurs when the target widget of manual transfer is different from w_n identified by CRAFTDROID; *false negative* occurs when CRAFTDROID fails to find a target widget, while manual transfer can; and *true positive* occurs when the target widget from manual transfer matches w_n identified by CRAFTDROID. Based on these metrics, we measured the *Precision* as the number of generated target events that are correct. Additionally, *Recall* measures how many of the source events are correctly transferred. Our experimental data is publicly available [29].

[2]Test suites for *Geek*, *Wish*, and *Etsy* apps are from [37]

[3]The number of actual GUI and oracle events in the test cases may be more than the number of steps shown in Table IV, since Table IV only provides general instructions for testing the functionalities

B. RQ1: Effectiveness

Table V demonstrates the effectiveness of CRAFTDROID in terms of successful transfers for each functionality listed in Table IV. These results demonstrate that on average, 74.5% of the attempted transfers by CRAFTDROID are successful, with an overall 73% precision and 90% recall considering all the transferred GUI and oracle events. Thereby, CRAFTDROID is substantially effective in identifying correct GUI widgets and successfully transferring tests across mobile apps.

The results shown in Table V also confirm that finding a match for all the widgets in source test is not necessary to successfully transfer a test. As an instance for such cases, consider the functionality *b11*, where its corresponding precision for transferring GUI events is 79%, while it successfully transfers all tests (success rate = 100%). That is, transfer of events from source app to target app in *b11* has been accompanied by false positives, i.e., the target widget is identified incorrectly. However, these false positives are not harmful, since different apps might implement common functionalities in different ways. For example, while one app may require the user to confirm the provided password during registration and before an account is created, this confirmation may not be required in another app, thereby, can be skipped.

While false positive may be acceptable, false negative is not, as it prevents the examination of the desired functionality. In other words, high recall is more important than high precision in test transfer, as false negatives typically have more adverse affect compared to false positives. CRAFTDROID's high recall of 90% for transferring GUI and oracle events makes it suitable for test transfer.

Another important observation from the results in Table V is that the success rate varies significantly among different categories of apps, ranging from 100% success rate for the apps under *Browser* and *Mail Client* categories to 40% for Shopping apps. Even within the same category, the success rate varies for different functionalities. In the next research question, we investigate the attributes that impact the success rate of test transfer.

C. RQ2: Factors Impacting Effectiveness

To identify the factors that impact effectiveness of a test transfer, we manually investigated all of the attempted transfers, including both successful and failed ones. We identified the following reasons for transfer failure:

Length of test. Intuitively, transfer of a long test is more challenging compared to a shorter one, since more GUI and oracle events should be transferred. Thereby, more false positives and false negatives might be generated. To identify how the length of tests impact effectiveness of a test transfer, we calculated the Pearson correlation coefficient [40] between the average length of tests, i.e., number of total events, and the effectiveness metrics in our experiments. Table VI represents

TABLE IV: Identified main functionalities for subject apps.

Category	Functionality	Test Steps
a1-Browser	b11-Access website by URL	1. Locate the address/search bar 2. Input a valid URL and press Enter 3. Specific content about the URL should appear
	b12-Back button	1. Locate the address/search bar 2. Input valid URL1 and press Enter 3. Specific content about URL1 should appear 4. Input valid URL2 and press Enter 5. Specific content about URL2 should appear 6. Click the back button 7. Specific content about URL1 should appear
a2-To Do List	b21-Add task	1. Click the add task button 2. Fill the task title 3. Click the add/confirm button 4. The task title should appear in the task list
	b22-Remove task	1. Add a new task 2. Click/long-click/swipe the task in the task list to remove the task 3. Click the confirm button if exists 4. The task should not appear in the task list
a3-Shopping	b31-Registration	1. Click the register/signup button 2. Fill out necessary personal data 3. Click the submit/signup button to confirm registration 4. Personal data should appear in the profile page
	b32-Login with valid credentials	1. Click the login/signin button 2. Fill out valid credentials 3. Click the submit/signin button to login 4. Personal data should appear in the profile page
a4-Mail Client	b41-Search email by keywords	1. Start the inbox activity 2. Click the search button 3. Input keywords for search and press Enter 4. Specific email related to the keywords should appear
	b42-Send email with valid data	1. Start the inbox activitiy 2. Click compose button 3. Input an unique ID for the subject 4. Input a valid email address for the recipient 5. Click send button 6. The unique ID should appear in the inbox
a5-Tip Calculator	b51-Calculate total bill with tip	1. Start the tip calculation activity 2. Input bill amount and tip percentange 3. Total amount of bill should appear based on the values in step 2
	b52-Split bill	1. Start the tip calculation activity 2. Input bill amount and tip percentage 3. Input number of people 4. Total amount of bill per person should appear based on the values in step 2 and 3

TABLE V: Effectiveness and Efficiency Evaluation of CRAFTDROID.

Functionality	GUI Event		Oracle Event		#Successful Transfer	Avg. Transfer Time (sec)
	Precision	Recall	Precision	Recall		
b11	79%	100%	100%	100%	20/20 (100%)	1,144
b12	85%	100%	100%	100%	20/20 (100%)	4,986
b21	78%	100%	85%	100%	17/20 (85%)	1,051
b22	69%	100%	85%	80%	11/20 (55%)	10,611
b31	44%	90%	34%	67%	8/20 (40%)	14,974
b32	53%	82%	56%	61%	10/20 (50%)	8,644
b41	100%	100%	100%	100%	20/20 (100%)	349
b42	85%	80%	89%	89%	14/20 (70%)	2,611
b51	82%	100%	100%	80%	16/20 (80%)	2,581
b52	80%	100%	100%	65%	13/20 (65%)	6,762
Total	70%	94%	79%	85%	149/200 (74.5%)	5,371

TABLE VI: Pearson correlation coefficient between average test length and effectiveness.

	GUI Event		Oracle Event		#Successful Transfer
	Precision	Recall	Precision	Recall	
Avg. Test Length	-0.74	-0.60	-0.87	-0.51	-0.71

the computed correlation coefficients. These results indicate a strong and negative correlation between the length of tests and success of a test transfer.

Complexity of app. Complexity of subject apps, in terms of their interface and functionality, also impacts the effectiveness of CRAFTDROID. Some categories of apps have standard or de-facto design guidelines, such as arrangement of GUI widgets to follow, which makes the transfer of test cases across such apps easier. For example, the design guideline for browser apps is to have a simple main screen that only contains a search bar and few actionable GUI widgets. This relatively simple design for the browser apps makes the transfer of the GUI events across them easier, since there are fewer candidate widgets on a screen to be analyzed for proper mapping.

On the other hand, apps without uniform design guidelines, such as Shopping apps, are flexible to determine the number of functionalities contained on a screen and the number of required steps for a functionality. This flexibility makes the search for finding correct matches more complicated. As demonstrated by the results in Table V, while CRAFTDROID successfully transfers all the tests under the Browser category, the success rate of transfer among Shopping apps is not as high as other categories.

D. RQ3: Efficiency

Table V shows the average running time of CRAFTDROID to transfer a test from one app to another when executed sequentially. On average, a test transfer takes less than 1.5 hours, ranging from 6 minutes to 4.2 hours among different functionalities. Performance evaluation of different components of CRAFTDROID shows that validating reachability of a candidate widget is the most time-consuming part of test transfer. That is, the function `getLeadingEvents` in Algorithm 2 dominates the execution time, as it frequently restarts the target app to validate the potential paths for a candidate widget.

Fortunately, this function can be easily parallelized. Multiple devices or emulators can be used to drastically cut down the execution time by performing the reachability check in parallel. For example, in our experiments, CRAFTDROID verifies 6.6 paths on average to transfer a source event. As a result, by using 6 emulators, we can speed up the transfer approximately 6 times to reduce the average running time to 15 minutes. We believe this is a reasonable amount of time to produce a feature-based test, consisting of both inputs and oracles.

E. RQ4: Factors Impacting Efficiency

By analyzing efficiency of test transfer among different apps and functionalities, we identified three factors that impact the efficiency of CRAFTDROID: (1) length of tests, (2) transfer success, and (3) size of target app. Intuitively, the longer is a test, it takes more time to transfer its events. In fact, the average test length and average transfer time are strongly and positively correlated, as the Pearson correlation coefficient between them is 0.81 in our experiments.

In addition, we observed that unsuccessful transfers take more time compared to successful ones. That is, an unsuccessful transfer often needs to examine and validate more candidate widgets during transfer. In our experiments, the average running time of the 149 successful transfers is $3,577$ seconds, while this number for the remaining 51 unsuccessful transfers is $10,613$ seconds, meaning unsuccessful transfers are $3x$ slower. Finally, the size of *UITG* is positively correlated to the size of app—with correlation coefficient = 0.5. Since CRAFTDROID heavily relies on the *UITG* to search and validate the correct widget, it requires more time to transfer a test for a larger target app.

F. Threats to Validity

The major external threat to validity of our results is the generalization to other mobile apps and test cases. To mitigate this threat, we collected 25 commercial and open-source apps from Google Play and F-Droid under various categories. The main internal threat to validity of the proposed approach is the possible mistakes involved in our implementation and experiments. We manually inspected all of our results to increase our confidence in their correctness. The experimental data is also publicly available for external inspection. In terms of the construct validity, CRAFTDROID assumes that the source test is transferable, i.e., the source and the target apps share similar functionalities, which is not always true. However, CRAFTDROID is not designed to generate test cases for every possible or app-specific features. It aims at reducing the manual effort of implementing tests for common or popular functionalities across apps. Our evaluation shows that this assumption does hold for apps under different categories.

VIII. RELATED WORK

1) Common functionalities across GUI-based apps: Several prior works have discussed common functionalities across desktop software [28], application-agnostic features across mobile apps [15], and common GUI patterns used in web app testing [19], [41]. Augusto [28] studies common functionalities such as authentication and saving a file in desktop software and proposes an automated technique to generate GUI tests for them with pre-defined GUI structures and formal pre/post conditions. Zaeem et al. [15] introduce several application-agnostic UI interactions, which can serve as oracles for mobile testing. Ermuth and Pradel [19] propose that sequences of low-level UI events, which correspond to high level logical steps can be inferred from test traces and to be further used for test generation. Moreira et al. [41] develop a domain-specific language to assist modeling and testing of UI patterns. Similar to the above work, CRAFTDROID exploits the existence of commonality across GUI-based apps. However, unlike them, CRAFTDROID aims to generate feature-based tests for an app from existing tests for apps within the same category (similar features).

2) Semantic mapping of GUI widgets: In recent years, researchers have proposed approaches for transferring or reusing tests on different platforms. Rau et al. [26] proposed a technique for mapping of GUI widgets among *web applications*. Behrang and Orso [25] proposed an approach to transfer test cases by mapping the GUI widgets to support assessment of mobile app coding assignments. Hu et al. [37] presented a framework that leverages machine learning to synthesize reusable UI tests for mobile apps. Qin et al. [42] recently proposed to migrate GUI events for the different instances of the same app running on different operating systems. Unlike all prior work, CRAFTDROID is able to transfer test oracles. In fact, [26], [25], and [42] only discuss GUI element mapping, and [37] focuses on generating GUI tests from high-level, manually written test cases. Additionally, CRAFTDROID utilizes an unsupervised and data-driven model, i.e., Word2Vec, to compute the similarity score between GUI widgets, hence it is fully automated. On the other hand, [25] uses a lexical database such as *WordNet*, and [37] adopts supervised machine learning, both of which require human effort in database maintenance or data annotation, for this purpose. Finally, CRAFTDROID leverages both static and dynamic analyses to transfer tests, while [25] only adopts dynamic analysis.

Concurrent to our development of CRAFTDROID, Behrang and Orso developed APPTESTMIGRATOR [43] to migrate GUI tests, including oracles, for mobile apps with similar functionality. While at a high level both works adopt similar techniques, e.g., using Word2Vec models and combination of static and dynamic analyses, CRAFTDROID is different from APPTESTMIGRATOR in terms of several algorithmic details, such as the ways it leverages the statically extracted model of app and computes similarity between GUI widgets. Since both approaches have similar goals, an empirical comparison between them in future may provide more insights into their relative strengths and weaknesses.

IX. CONCLUSION AND FUTURE WORK

In this paper, we presented CRAFTDROID, a framework for transferring tests across mobile apps through semantic mapping of actionable GUI widgets. We evaluated CRAFTDROID using 25 real-world apps from 5 categories. Our experimental results show that 75% of the attempted transfers are successful, with 73% precision and 90% recall for the transferred GUI and oracle events. We also discussed the factors impacting the effectiveness and efficiency of CRAFTDROID, which can be used as a guideline by researchers to improve test transfer techniques.

For the future work, we are planning to conduct empirical study with more apps and incorporate techniques such as crowd sourcing to improve the effectiveness of CRAFTDROID. We share the vision of Behrang and Orso [27] toward the establishment of a centralized repository similar to App Store, but for test cases. This Test Store will be able to generate feature-based test cases for newly developed apps. The knowledge mined from existing tests and apps, which we use for test transfer, can also have applications beyond testing, such as suggesting missing features and improving GUI layouts/flows for new apps.

ACKNOWLEDGMENT

This work was supported in part by awards CCF-1618132 and CNS-1823262 from the National Science Foundation.

REFERENCES

[1] https://developer.android.com/studio/test/monkey.

[2] https://firebase.google.com/docs/test-lab/android/robo-ux-test.

[3] 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, Sept 2015.

[4] 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*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[5] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 623–640. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509552

[6] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 204–217. [Online]. Available: http://doi.acm.org/10.1145/2594368.2594390

[7] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "Qbe: Qlearning-based exploration of android applications," in *Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on*. IEEE, 2018, pp. 105–115.

[8] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 67–77. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483777

[9] 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 *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 111–122. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820534

[10] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[11] 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*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 599–609. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635896

[12] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 94–105. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931054

[13] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for android applications," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2015, pp. 461–471.

[14] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Fundamental Approaches to Software Engineering*, V. Cortellessa and D. Varró, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 250–265.

[15] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ser. ICST '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 183–192. [Online]. Available: http://dx.doi.org/10.1109/ICST.2014.31

[16] H. Zhang and A. Rountev, "Analysis and testing of notifications in android wear applications," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 347–357. [Online]. Available: https://doi.org/10.1109/ICSE.2017.39

[17] 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*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 245–256. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106298

[18] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 643–653. [Online]. Available: https://doi.org/10.1109/ICSE.2017.65

[19] M. Ermuth and M. Pradel, "Monkey see, monkey do: Effective generation of gui tests with inferred macro events," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 82–93. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931053

[20] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*, 2013, pp. 15–24. [Online]. Available: https://doi.org/10.1109/ESEM.2013.9

[21] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1–10.

[22] M. Linares-Vsquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *2017 IEEE Interna-*

tional Conference on Software Maintenance and Evolution (ICSME), Sept 2017, pp. 613–622.

[23] P. Tonella, R. Tiella, and C. D. Nguyen, "Interpolated n-grams for model based testing," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 562–572. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568242

[24] R. Jabbarvand and S. Malek, "μdroid: an energy-aware mutation testing framework for android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 208–219.

[25] 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*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 164–175. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213854

[26] A. Rau, J. Hotzkow, and A. Zeller, "Transferring tests across web applications," in *Web Engineering*, T. Mikkonen, R. Klamma, and J. Hernández, Eds. Cham: Springer International Publishing, 2018, pp. 50–64.

[27] F. Behrang and A. Orso, "Automated test migration for mobile apps," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 384–385. [Online]. Available: http://doi.acm.org/10.1145/3183440.3195019

[28] 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*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 280–290. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180162

[29] https://github.com/seal-hub/CraftDroid.

[30] https://play.google.com/store/apps/details?id=com.rainbowshops.

[31] https://play.google.com/store/apps/details?id=com.yelp.android.

[32] https://developer.android.com/studio/command-line/adb.html.

[33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119. [Online]. Available: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

[34] https://github.com/appium/appium.

[35] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[36] A. Rosenfeld, O. Kardashov, and O. Zang, "Automation of android applications functional testing using machine learning activities classification," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '18. New York, NY, USA: ACM, 2018, pp. 122–132. [Online]. Available: http://doi.acm.org/10.1145/3197231.3197241

[37] G. Hu, L. Zhu, and J. Yang, "Appflow: Using machine learning to synthesize robust, reusable ui tests," in *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ser. ESEC/FSE 2018, 2018.

[38] 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)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 429–440. [Online]. Available: https://doi.org/10.1109/ASE.2015.89

[39] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 559–570.

[40] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[41] R. M. Moreira, A. C. Paiva, and A. Memon, "A pattern-based approach for gui modeling and testing," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 288–297.

[42] 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*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, pp. 284–295. [Online]. Available: http://doi.acm.org/10.1145/3293882.3330575

[43] F. Behrang and A. Orso, "Test migration between mobile apps with similar functionality," in *Proceedings of the The 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19, 2019. To appear.