Test Migration Between Mobile Apps with Similar Functionality

Farnaz Behrang and Alessandro Orso
Georgia Institute of Technology
Atlanta, USA
behrang@gatech.edu orso@cc.gatech.edu

***Abstract*—The use of mobile apps is increasingly widespread, and much effort is put into testing these apps to make sure they behave as intended. To reduce this effort, and thus the overall cost of mobile app testing, we propose APPTESTMIGRATOR, a technique for migrating test cases between apps in the same category (e.g., banking apps). The intuition behind APPTESTMIGRATOR is that many apps share similarities in their functionality, and these similarities often result in conceptually similar user interfaces (through which that functionality is accessed). APPTESTMIGRATOR leverages these commonalities between user interfaces to migrate existing tests written for an app to another similar app. Specifically, given (1) a test case for an app (source app) and (2) a second app (target app), APPTESTMIGRATOR attempts to automatically transform the sequence of events and oracles in the test for the source app to events and oracles for the target app. We implemented APPTESTMIGRATOR for Android mobile apps and evaluated it on a set of randomly selected apps from the Google Play Store in four different categories. Our initial results are promising, support our intuition that test migration is possible, and motivate further research in this direction.**

***Index Terms*—Test migration, GUI testing, mobile apps**

I. INTRODUCTION

Mobile apps are used on a daily basis to perform a number of tasks, such as reading the news, accessing social media, and shopping. It is therefore important to thoroughly test these apps to gain confidence that they behave as intended when used in the field. Manually developing test cases for an app tends to be extremely expensive, as it involves human effort to define test cases and check test results.

We believe that the cost of testing mobile apps can be reduced by considering similarities between apps and migrating test cases across similar apps. More precisely, *this work is motivated by the intuition that, although GUIs for different apps can differ dramatically, there are many cases in which apps share similarities that result in conceptually similar GUIs*. Typical examples of this situation are apps that belong to the same category, such as banking applications, which share much of their functionality and may provide GUIs that are inherently similar. It is worth noting that migrating test cases is different in nature from—and mostly orthogonal to—input generation. Whereas input generation aims to produce inputs based on some criteria (e.g., coverage goals), *test migration has the potential to generate test cases that exercise specific scenarios that developers considered to be of interest for the functionality under test*.

Based on our intuition, we defined APPTESTMIGRATOR (App Test Migrator), a technique for migrating test cases (i.e., test inputs and oracles) between apps that share part of their functionality. APPTESTMIGRATOR takes as input a source app, a test case for the source app (source test), and a target app, and produces as output the source test migrated to the target app (target test). To do so, it (1) records both the sequence of (GUI) events generated and the assertions checked by the source test, (2) migrates events and assertions to the target app using a similarity metric based on a combination of techniques, and (3) generates a target test case based on the migrated events and assertions.

There is a good amount of recent work, including our own, that shares a similar intuition and proposes approaches for exploiting commonalities in the functionality of different apps and improve GUI test generation (e.g., [4,5,13,21,28,29]). In fact, this paper builds on the vision and ideas that we proposed in earlier work [4] and extends GUITESTMIGRATOR, our approach for migrating test cases between student apps developed based on identical specifications [5]. Specifically, APPTESTMIGRATOR (1) extends our earlier work in a number of ways and (2) addresses several limitations that hinder the effectiveness of GUITESTMIGRATOR when migrating tests between apps that do not have identical specifications but simply share part of their functionality.

First, while evaluating GUITESTMIGRATOR for use in this context, we realized that its success rate was inversely correlated to the number of events in the tests (i.e., the higher the number of events, the higher the chances of "getting lost" during dynamic exploration, or crawling). To address this issue, APPTESTMIGRATOR migrates each event independently from other events while crawling a target app. *Second*, APPTESTMIGRATOR incorporates the use of static analysis to prune the search space by taking into account the possible future states of the app, which can further improve the effectiveness of the approach. *Third*, APPTESTMIGRATOR improves the way in which GUITESTMIGRATOR computes the similarity between app entities and between events across apps. For the former, APPTESTMIGRATOR builds an app ontology that captures the semantic similarity between words using word embeddings produced by Word2Vec [24] and leverages all available textual information (e.g., labels, content descriptions, hints). For the latter, APPTESTMIGRATOR uses a new approach that is not based on the similarity of actions

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00016

or types associated with events, but rather on the targets of the event actions. *Fourth*, unlike GUITESTMIGRATOR, APPTESTMIGRATOR can migrate oracles in addition to test inputs. Migrating oracles for app tests involves understanding how developers write such oracles in practice and defining a technique that can map them across apps accordingly. This is particularly challenging, as there are typically multiple ways to find a counterpart for an oracle in a target app, but it can make the approach considerably more useful in practice: tests that are successfully migrated together with their oracles can in fact also detect failures that do not result in crashes.

To evaluate APPTESTMIGRATOR, while also assessing its improvements over previous work, we implemented it in a tool that supports Android apps and test cases written using the Espresso testing framework [3]. We then used our tool to migrate test cases for apps in four categories: *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather*. More precisely, we evaluated APPTESTMIGRATOR on (1) 4 apps per category (16 apps overall) randomly selected from the Google Play Store and (2) 48, 38, 38, and 34 test cases for the 4 categories. We used both the test cases provided with the apps, when available, and additional test cases written by CS graduate students not involved in this research. For each category, we then considered each app as a source app and the remaining apps as target apps, which resulted in APPTESTMIGRATOR attempting to migrate over 150 test cases.

Overall, APPTESTMIGRATOR fully migrated 48% and partially migrated 34% of the tests considered. For 42% of the fully migrated tests, it also fully migrated their oracles. We also summarize these results in terms of individual events and oracles migrated. On average, APPTESTMIGRATOR was accurate for 78% of the events considered; that is, for 78% of the events it either successfully migrated them (54%) or did not migrate them when they had no counterpart in the target app (24%). For the remaining events (22%), APPTESTMIGRATOR either matched them to the wrong event (17%) or did not match them when a matching was possible (5%). As for oracle migration, APPTESTMIGRATOR was accurate for 76% of the oracles considered; that is, for 76% of the oracles it either successfully migrated them (47%) or did not migrate them when they had no counterpart in the target app (29%). For the remaining oracles (24%), APPTESTMIGRATOR either matched them incorrectly (13%) or did not match them when a matching was possible (11%). Note that we considered measuring APPTESTMIGRATOR's effectiveness also in terms of coverage and fault-detection ability of the migrated tests. We decided against it because we believe that a meaningful and fair computation of these metrics would be extremely difficult, as we discuss in Section V.

In addition to assessing APPTESTMIGRATOR's performance in isolation, we compared its effectiveness with that of GUITESTMIGRATOR, so as to assess whether our extensions resulted in actual improvements in practice. In addition to not supporting oracle migration at all, GUITESTMIGRATOR was outperformed by APPTESTMIGRATOR in terms of both full test migration (16 percentage points improvement) and accuracy in individual event migration (11 percentage points improvement). We also manually inspected all the cases in which APPTESTMIGRATOR outperformed GUITESTMIGRATOR to understand which of our extensions to GUITESTMIGRATOR played a role in improving the effectiveness of the approach. As a result of this analysis, we found that considering events independently while crawling a target app, using static analysis to prune the search space, and using a new approach to compute similarity between entities and events across app accounted for APPTESTMIGRATOR's improvements in 23%, 36%, and 41% of the cases, respectively.

We believe that these results are promising, as they show that a considerable number of test cases can be successfully migrated between apps that share only partial functionality and were developed in a totally independent way. The results support our intuition that test migration is possible and motivate further research in this direction. In particular, our technique should generally be applicable to any GUI-based software, including web apps. In addition, if the effectiveness of test migration were confirmed by further studies, this could support the compelling idea of a "test store" that operates in parallel with a traditional app store; when developers submit an app, the test store could analyze the app, look for similar ones, migrate tests from these apps, and return all the tests that were successfully migrated. We believe that this is a realistic scenario, as many developers already provide test cases in public repositories, including for apps also available in app stores (e.g., WordPress). The benefit of getting additional tests in return may further motivate developers to create and share tests. In general, migrated tests could be provided as a service or used as an extra check before an app is published.

This paper makes the following contributions:

- A new technique for migrating test cases, including oracles, between apps that share part of their functionality.
- An implementation of the technique for the Android platform that is publicly available, together with our experimental data and infrastructure [6].
- An empirical evaluation that provides initial, yet strong evidence of the effectiveness and potential usefulness of our technique and that shows its improvements over the state of the art.

II. BACKGROUND AND TERMINOLOGY

Mobile apps are largely tested through their GUIs; GUI-based testing aims to bring the app into a particular state through a sequence of GUI events, such as clicking on a button or submitting text to a form, and uses oracles to check the outcome of the test (e.g., the existence or the specific value of a property for a given GUI element). Since oracles in this context are assertion-based, in the paper we use terms *oracle* and *assertion* interchangeably. A *GUI element* (or simply *element*) is a graphical widget on a screen of the app. A *GUI state* s is a set of triples (w, p, v), where w is a GUI element, p is a property of w, and v is the value of p. A *GUI event* (or simply *event*) e is a triple (a, t, i), where a is the action

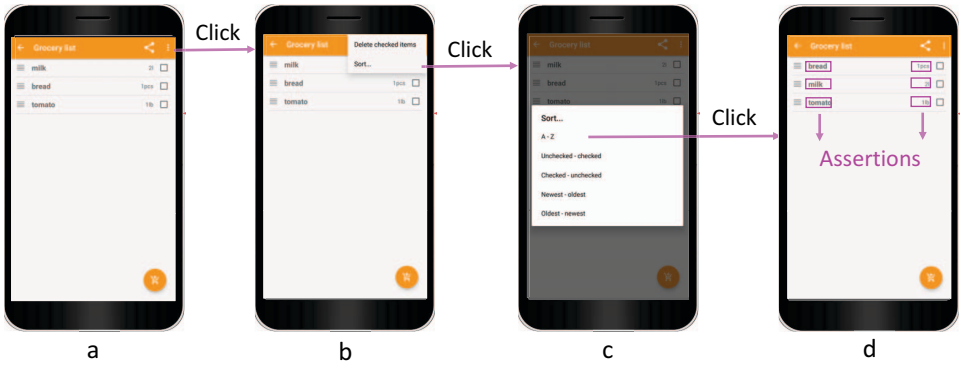

Fig. 1: Sequence of events of a test case that sorts items in a list in the source app.

Fig. 2: Sequence of events migrated by APPTESTMIGRATOR from the source app (Fig. 1) to the target app.

that corresponds to the event (e.g., click), t is the target of the action (e.g., button), and i is the (optional) input value (e.g., data for a text input box). An *assertion* as, is a function $F : (e, c) \rightarrow \{True, False\}$, where e is a GUI element, and c is a condition to be checked for that element. The function returns *True* if e satisfies condition c, and *False* otherwise.

III. MOTIVATING EXAMPLE

In this section, we present a motivating example that consists of two shopping list apps, a source app and a target app, and a test case for the source app (source test). The goal of APPTESTMIGRATOR is to migrate the source test from the source app to the target app. Fig. 1 shows the sequence of events and assertions that the test case triggers to check the "sorting items in list" functionality in the source app. (The sequence of events to add items to the list are not shown due to space limitations.) The test clicks on "More options" (a→b), then "Sort..." (b→c), and finally "A - Z" (c→d). It then checks whether the items are displayed in the sorted order by using six assertions. Items that are checked by assertions are highlighted with rectangles on screen *d*.

Fig. 2 shows the sequence of events and assertions generated by APPTESTMIGRATOR for the target app after migration. The test clicks on "More options" (a→b), then "Settings" (b→c), and finally on "Default style ordering of items" (c→d). Next, it selects "Ascending Alphabetical Order" (d→e). It then clicks the "Navigate up" button (e→f) and checks whether the items are displayed in the sorted order through six assertions. Also here, items that are checked by assertions are highlighted with rectangles on screen *f*.

As the example shows, although the apps belong to the same category, and provide similar functionality, their GUIs are quite different, which makes it difficult to migrate test cases from the source to the target app.

IV. APPROACH

Fig. 3 shows an overview of our approach, APPTESTMIGRATOR. As the figure shows, APPTESTMIGRATOR takes as input the *source app*, the *source tests* (a set of tests for the source app), and the *target app*, and produces as output the *target tests* (the source tests migrated to the target app). APPTESTMIGRATOR consists of five main modules: *Instrumenter*, *Test runner and recorder*, *Event migrator*, *Assertion migrator*, and *Test encoder*. First, *Instrumenter* instruments the source tests so that *Test runner and recorder* can record both the sequence of events generated and the assertions (i.e., oracles) checked by the tests. *Event migrator*, and subsequently *Assertion migrator*, then migrate the events and assertions from the source app to the target app. Finally, given the migrated events and assertions, *Test encoder* generates actual test cases for the target app. In the rest of this section, we discuss each step in detail.

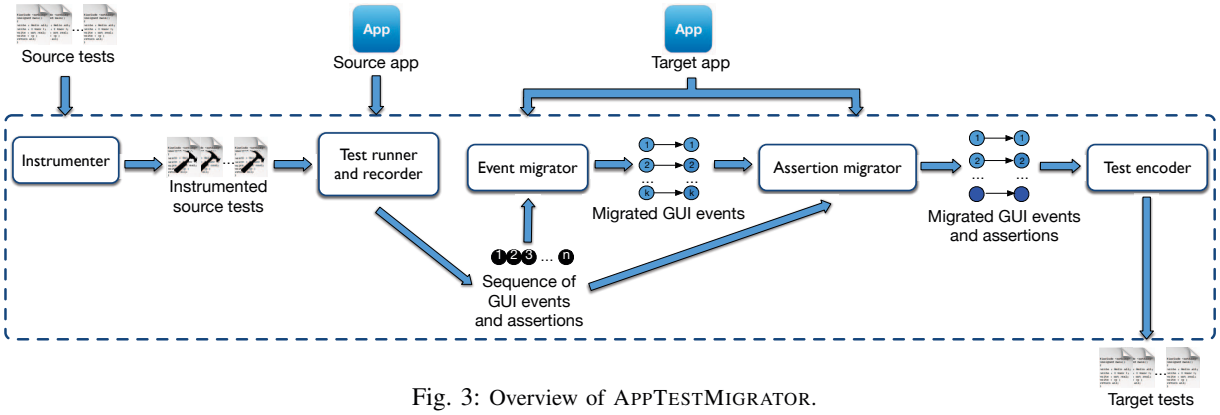

Fig. 3: Overview of APPTESTMIGRATOR.

A. Instrumenter

Testing frameworks provide APIs that let users write test cases by generating events and assertions. The *Instrumenter* module instruments these APIs to collect information about GUI interactions.

B. Test Runner and Recorder

The *Test runner and recorder* module runs the instrumented source tests to record the sequence of events they generate and the assertions they check. For each event, it logs the performed action, the element that is the target of the action, and the input value used by the action, if any.

C. Event Migrator

The *Event migrator* is one of the key modules of our technique. Given (1) a sequence of source events extracted from a source test and (2) the target app, this module tries to migrate the source events to the target app. Algorithm 1 (MIGRATEEVENTS) describes how this module operates.

The algorithm first computes a *Window Transition Graph (WTG)* [33] for the target app (line #5)—a statically-computed graph where nodes represent windows (i.e., activities, menus, and dialogs) and edges represent transitions between windows, triggered by callbacks executed in the UI thread [2]. The algorithm then launches the target app (line #6).

For each source event, the algorithm tries to find a match in the target app (target event) within a predefined time limit (lines #7-74). To do so, it gets the GUI state and finds all the elements in the GUI state with which users can interact: it first traverses the GUI state and checks the value of the attributes of the elements and of their ancestors; it then identifies *actionable elements*, that is, elements that are clickable, long-clickable, or checkable (lines #14–15).

After identifying the actionable elements, the algorithm computes a similarity score between the source event and these elements (call to method *computeSimilarityScore* on line #18, discussed later in this section.) The actionable element with the highest similarity score is considered a match for the source event, if the score is above a given threshold. If the algorithm finds multiple matches, it selects one of them randomly and records the others as alternative matches (lines #19–26). We discuss how the algorithm might use these alternative matches later in this section. If the algorithm can find at least a match, it adds to the target events the actionable element matched (lines #28–31). It then triggers the matched actionable element and continues by trying to find a match for the next source event (lines #32–33).

Conversely, if the algorithm does not find any matches in the current GUI state, it leverages the WTG it statically computed for the target app to find a matching element elsewhere in the app. Specifically, it finds all the actionable elements in the WTG and tries to match them against the current source event. Also in this case, the actionable element with the highest similarity score (above a given threshold) is considered a match for the source event (lines #35–43). If one such match is found, the algorithm tries to find the shortest path $sPath$ in the WTG from the current GUI state to the GUI state where the matched actionable element exists (lines #44–46). If $sPath$ exists, the algorithm considers the first actionable element of the path as the next event to trigger (lines #47–49). Note that the algorithm only triggers the first actionable element of the path, rather than triggering them all, because the WTG is computed statically and may contain infeasible paths; roughly speaking, triggering the first event is likely to point the search in the right direction even when the whole path $sPath$ is not feasible. After triggering this element, APPTESTMIGRATOR again tries to find the next match dynamically.

If the algorithm does not find any match, either dynamically or statically, it semi-randomly[1] selects one of the actionable elements that was not been previously selected (lines #51–54). If the number of consecutive random events triggered reaches a predefined limit, the algorithm backtracks to the GUI state where the first random event was triggered (lines #55–58).

If the algorithm times out without finding a match for the current source event ($currSrcEv$), it checks whether an alternative match exists for the last matched source event ($lastMatchedEv$). If so, it (1) backtracks by invalidating the last target event and replacing it with one of the alternative matches (lines #64–70), (2) relaunches the target app (line

[1]Intuitively, and similar to GUITESTMIGRATOR [5], APPTESTMIGRATOR would select a random event while prioritizing "promising" events (e.g., a dialog button if the previous event resulted in opening a dialog window).

#71), (3) triggers all the previously migrated events, including the selected alternative match (line #72), and (4) tries to find a match for $currSrcEv$ in this new GUI state. (Relaunching the target app and replaying events is necessary because an app execution cannot be easily backtracked.) Conversely, if there are no possible alternative matches for $lastMatchedEv$, the algorithm skips $currSrcEv$ and tries to match the source event that follows $currSrcEv$. Also in this case, the algorithm relaunches the app and replays the previously matched events up to (and including) the one(s) matching $lastMatchedEv$. It does so to undo the possible triggering of target events that were statically or randomly identified while looking for a match for $currSrcEv$.

After processing all source events, the algorithm returns the sequence of successfully migrated target events (line #75).

Computing the Similarity Score Between Two Events: To match a source event ev_s to a target event ev_t, APPTESTMIGRATOR computes their similarity score by comparing the GUI elements targeted by the events (e.g., the button in the case of a button click). We refer to these elements as el_s (source element) and el_t (target element). Our technique does so, instead of assessing the similarity of the actual events, because it is possible to provide the same functionality by performing different actions on a target element. For instance, it is possible to select an element by clicking, long clicking, or checking such element. Similarly, APPTESTMIGRATOR does not require matching elements to be of the same type (unless the element is of a type that accepts inputs), as different elements might provide the same functionality (e.g., *Button*, *ImageButton*, and *TextView*, in Android).

Given ev_s and ev_t, our technique extracts all the textual information associated with el_s and el_t. First, if an element has a label, or any label exists within a predefined distance from the element, the technique extracts it. Second, it checks whether the developer defined any content description or hint for the element and, if so, it retrieves the values of these attributes as well. Third, it considers the ID of the element, as developers tend to assign meaningful ids to elements. Finally, if the element is an image, the technique retrieves the image filename, which is also often meaningful.

After extracting these pieces of textual information associated with el_s and el_t, the technique first preprocesses them. In particular, it tokenizes each piece of textual information and applies lemmatization [19] to the resulting tokens. The results of this preprocessing are two sets of tokens, one for el_s (set_s) and one for el_t (set_t). The technique then considers all possible pairs of tokens (tok_s, tok_t), such that $tok_s \in set_s$ and $tok_t \in set_t$, and computes two distance scores for each such pair based on (1) edit distance [31] and (2) semantic similarity. To compute this latter, we created an ontology for mobile apps based on word embeddings that we generated using the Word2Vec [24] methodology. Specifically, to build the ontology, we generated a Word2Vec model using 500 randomly selected user manuals for mobile apps in different categories. The user manuals contain sets of instructions that correspond to different user scenarios for a given app. For

Algorithm 1 Algorithm for migrating events.

```
    Input: srcEvents, targetApp
    Output: targetEvents
 1: procedure MIGRATEEVENTS
 2:     MR ← MAX_CONSECUTIVE_RANDOM_EVENTS
 3:     MT ← MATCH_THRESHOLD
 4:     targetEvents ← List< tEvent >
 5:     wtg ← computeWTG(targetApp)
 6:     launchTargetApp(targetApp)
 7:     for index ← 1 to srcEvents.size() do
 8:         currSrcEv ← srcEvents.get(index)
 9:         matched ← False
10:         numRandEvents ← 0
11:         eventsTriggered ← List< tEvent >
12:         while !timeout() do
13:             nextEvent ← null
14:             state ← getGUIState()
15:             actionables ← findActionables(state)
16:             mScore ← 0
17:             for each actionable in actionables do
18:                 score ← computeSimilarityScore(currSrcEv, actionable)
19:                 if score > MT ∧ score >= mScore then
20:                     if nextEvent != null then
21:                         nextEvent ← actionable
22:                     else
23:                         nextEvent.setAlternatives(actionable)
24:                     end if
25:                     mScore ← score
26:                 end if
27:             end for
28:             if nextEvent != null then
29:                 matched ← True
30:                 eventsTriggered.add(nextEvent)
31:                 targetEvents.addAll(eventsTriggered)
32:                 triggerEvent(nextEvent)
33:                 break
34:             else
35:                 sActionables ← findStaticActionables(wtg)
36:                 mScore ← 0
37:                 for EACH sA IN sActionables do
38:                     score ← COMPUTESIMILARITYSCORE(currSrcEv, sA)
39:                     if score > MT ∧ score > mScore then
40:                         staticNextEvent ← sA
41:                         mScore ← score
42:                     end if
43:                 end for
44:                 if staticNextEvent != null then
45:                     staticEventState ← getState(staticNextEvent)
46:                     sPath ← findShortestPath(state, staticEventState)
47:                     if sPath != null then
48:                         nextEvent ← sPath.getFirstEvent()
49:                     end if
50:                 end if
51:                 if nextEvent == null then
52:                     if numRandEvents != MR then
53:                         nextEvent ← pickNextEventSemiRandomly()
54:                         numRandEvents ← numRandEvents + 1
55:                     else
56:                         nextEvent ← back
57:                         numRandEvents ← 0
58:                     end if
59:                 end if
60:                 triggerEvent(nextEvent)
61:                 eventsTriggered.add(nextEvent)
62:             end if
63:         end while
64:         if !matched then
65:             lastMatchedEv ← getLastMatchedSourceEvent(targetEvents)
66:             if alternativeEventExists(lastMatchedEv) then
67:                 alternativeEvent ← getAlternative(lastMatchedEv)
68:                 targetEvents.replaceLastEvent(alternativeEvent)
69:                 index ← index - 1
70:             end if
71:             launchTargetApp(targetApp)
72:             triggerPreviouslyMigratedEvents(targetEvents)
73:         end if
74:     end for
75:     return targetEvents
76: end procedure
```

each word, the model produces a feature vector, and the semantic distance between two words is determined by the cosine similarity between their vectors. Building this ontology allows us to improve over our previous work [5], where we used general-purpose lexical databases such as WordNet [25].

After computing the scores based on edit and semantic distance between two tokens, our technique considers the maximum of these two values as the similarity score for these tokens. It then matches each source token with the target token that has the highest similarity score (above a given threshold) and computes the overall similarity score by averaging the scores of all the matched tokens in set_s and set_t.

D. Applying Algorithm 1 to the Example

We now show how algorithm 1 can migrate the source test (Fig. 1) to the target app (Fig. 2) in our motivating example.

APPTESTMIGRATOR first matches the clicks on the 3-Dots ("More options") menus in the source and target apps (screen *a* in both figures). The next event in the source test is a click on the element with label "Sort..." (screen *b* in Fig. 1). Since a corresponding element does not exist in the target app, APPTESTMIGRATOR does not find any matches. Therefore, it uses the WTG of the target app to find a match in another GUI state of the app and compute the shortest path from the current GUI state to the GUI state where the match exists. Due to space limits, we do not show the WTG and the statically matched element here. For the sake of the example, it suffices to say that the first actionable element of the computed shortest path is menu entry "Settings". Therefore, our technique would trigger that menu entry (screen *b* in Fig. 2).

APPTESTMIGRATOR then looks for a match for the next source event (click on menu entry "Sort...") by comparing it against the actionable elements of the current GUI state (screen *c* in Fig. 2) and finds two possible matches: "Default style ordering of checked items" and "Default alphabetical style ordering of items". (The technique finds these matches by considering the semantic relation between the words "sort" and "order" when computing the similarity scores between the tokens associated with the source and target elements.) Since the similarity scores for these two possible matches are the same, the technique triggers one of the two actionable elements (e.g., "Default style ordering of checked items") randomly and records the other as an alternative match. APPTESTMIGRATOR then tries to find a match for the next source event (click on menu entry "A - Z"), but it does not find any match, either statically or dynamically, within the time limit. Since the previous source event (click on "Sort...") has an alternative match, APPTESTMIGRATOR invalidates the previous match, "Default style ordering of checked items", and triggers the alternative match, "Default alphabetical style ordering of items" (screen *c* in Fig. 2). It then tries again to find a match for "A - Z", is able to match it to the *CheckedTextView* element with label "Ascending Alphabetical Order" (due to the semantic similarity of the terms "Ascending" and "A - Z"), and triggers the corresponding actionable element (screen *d* in Fig. 2). Note that our specialized ontology for mobile apps (described above) is what allows our technique to identify this semantic similarity, which would have not been possible using a general-purpose lexical database.

TABLE I: Assertions statistics.

Category	Property (if applicable)	Total	Percentage
UI-based			
	Displayed	430	48.6%
	Text	224	25.3%
	Clickable	22	2.5%
	EffectiveVisibility	11	1.2%
	Hint	10	1.1%
	SpinnerText	8	0.9%
	Checked	6	0.7%
	CompletelyDisplayed	4	0.5%
	Enabled	3	0.3%
	ContentDescription	1	0.1%
		719	81.2%
Hierarchy-based			
	Child	20	2.2%
	isDescendantOfA	14	1.6%
	Parent	13	1.5%
	hasDescendant	13	1.5%
		60	6.8%
DoesNotExist		50	5.7%
Intent-based		44	5%
Others		11	1.2%

E. Assertion Migrator

Assertions are essential parts of test cases. To better understand how assertions are written for GUI-based test cases, we manually inspected a large number of test cases for Android apps on GitHub. We specifically focused on test cases written using the Espresso testing framework [3], which provides various APIs for writing assertions. As we also explain in Section V-A, we chose Espresso because it is one of the major test automation frameworks.

We randomly selected 500 test cases, which contained 884 assertions and classified these assertions. Table I shows the categories of assertions that we identified, together with the total number and the percentage of assertions for each category. In total, we identified five main categories, where the most common category is *UI-based*, which accounts for 81.2% of the considered assertions. Assertions in this category check for properties of elements at a specific point of the execution. The list of such properties is also shown in Table I. The second most common category is *Hierarchy-based* (6.8% of the assertions). Assertions in this category check the relationships between two elements. We show different examples, such as child-parent (*Child*) and parent-child (*Parent*) relationships, in Table I. The third and fourth most common categories are *DoesNotExist* and *Intent-based*. *DoesNotExist* assertions check that a specific element does not exist in a given GUI state, whereas *Intent-based* assertions check specific properties of intents—message objects used within Android to request an action from another component, either within the same app or in another app. The rest of the assertions (1.2%) are specific to their corresponding apps and cannot be easily generalized.

In the rest of this section, we discuss how the *Assertion migrator* module migrates assertions that belong to categories *UI-based*, *Hierarchy-based*, and *DoesNotExist*, which account

for more than 93% of the assertions we observed. Note that the properties shown in Table I for each category are only a subset of the properties supported by our technique. Other examples include *Focus* and *Sibling* for the *UI-based* and *Hierarchy-based* categories, respectively. We decided not to consider the *Intent-based* category for now, as these assertions are used to test the control flow of different components of the apps, rather than their GUIs, and such control flow might be quite different even among similar apps.

Algorithm 2 describes our technique for migrating assertions. An assertion consists of (1) a condition $cond$ and (2) an element el on which $cond$ is checked. Therefore, to migrate an assertion, the algorithm must migrate both el and $cond$. For each source assertion as_s, consisting of a source element el_s and a source condition $cond_s$, the algorithm operates as follow. First, it launches the target app and triggers the migrated events therein (lines #3–4), so that it reaches the GUI state in which it can start migrating the assertion.

To migrate el_s, the algorithm compares all the elements of the current GUI state with el_s and identifies the target element el_t with the highest similarity score (line #10). To do so, function *findMatch* uses an approach analogous to the one we discussed in Section IV-C. If a suitable element cannot be found, the algorithm selects an event randomly, triggers it, and tries to find a match in the new GUI state (lines #11–14). The algorithm keeps exploring randomly until it either finds a match (line #16) or reaches a given time limit (line #8). If it is unable to find a match, either directly or through random exploration, the algorithm skips the current assertion (lines #19–20). Otherwise, it continues and tries to migrate $cond_s$ (lines #22–42) based on its category:

UI-based: If $cond_s$ checks a *UI-based* property $prop$, the algorithm gets the expected value val for $prop$ from el_t (line #25). Given el_t, $prop$, and val, the algorithm then generates an assertion for the target app (line #26).

Hierarchy-based: If $cond_s$ checks a *Hierarchy-based* property, the algorithm first tries to match the element associated with $cond_s$ with an element in the current GUI state, using again function *findMatch* (lines #28–29). For condition *Parent(parentEl)*, for instance, *parentEl* would be the element in the source app that must be matched in the target app. If *findMatch* is able to find a match, the algorithm generates an assertion for the target app using the matched element, the property that needs to be checked, and el_t (lines #30–32).

DoesNotExist: If $cond_s$ belongs to category *DoesNotExist*, the algorithm must identify and migrate in the source app an element, $srcEl$, that (1) has the properties specified in the assertion (e.g., a specific label), (2) does not exist in the last GUI state reached by the source test, where the assertion is checked, and (3) exists in a previous GUI state. Note that the last condition is included because, in most if not all cases, this kind of tests are used to check that a previously existing element has been successfully removed. To identify $srcEl$, function *findElement* (line #34) examines the GUI states reached by the source test from the beginning of the execution to the point where the assertion is checked.

Algorithm 2 Algorithm for migrating assertions.

```
    Input: targetEvents, srcAssertions, targetApp
    Output: targetAssertions
 1: procedure MIGRATEASSERTIONS
 2:     for each as_s in srcAssertions do
 3:         launchTargetApp(targetApp)
 4:         triggerMigratedEvents(targetEvents)
 5:         mAssertion ← null
 6:         mElement ← null
 7:         el_s ← as_s.getElement()
 8:         while !timeout() do
 9:             state ← getGUIState()
10:             el_t ← findMatch(state, el_s)
11:             if el_t == null then
12:                 nextEvent ← pickNextEventRandomly()
13:                 triggerNextEvent(nextEvent)
14:                 continue
15:             else
16:                 break
17:             end if
18:         end while
19:         if el_t == null then
20:             continue
21:         else
22:             cond_s ← as_s.getCondition()
23:             prop ← as_s.getConditionProperty()
24:             if cond_s is UI-based then
25:                 val ← el_t.getPropertyValue(prop)
26:                 mAssertion ← Assertion(el_t, prop, val)
27:             else if cond_s is Hierarchy-based then
28:                 cElement ← assertion.getConditionElement()
29:                 cmElement ← findMatch(getGUIState(), cElement)
30:                 if cmElement != null then
31:                     mAssertion ← Assertion(el_t, prop, cmElement)
32:                 end if
33:             else if cond_s is DoesNotExist then
34:                 srcEl, srcEv ← findElement(as_s)
35:                 trgEl ← srcEv.getTargetElement()
36:                 trgSt ← targetEvents.get(trgEl).getState()
37:                 lastTrgSt ← targetEvents.getLast().getState()
38:                 ne ← findDoesNotExist(el, trgSt, lastTrgSt)
39:                 if ne != null then
40:                     mAssertion ← Assertion(cond_s, ne)
41:                 end if
42:             end if
43:         end if
44:         if mAssertion != null then
45:             targetAssertions.add(mAssertion)
46:         end if
47:     end for
48:     return targetAssertions
49: end procedure
```

While doing so, function *findElement* also identifies the last GUI state in which $srcEl$ existed and the source event, $srcEv$, that led to that state. The algorithm then identifies the target event that was mapped to $srcEv$ when migrated, $trgEl$, and the GUI state, $trgSt$, resulting from triggering $trgEl$ (lines #35–36). The algorithm also identifies the GUI state, $lastTrgSt$, resulting from triggering the last target element; that is, the state in the target app in which the migrated element should not exist (line #37). To generate the actual assertion, method *findDoesNotExist* (line #38) checks whether there is an element ne in $trgSt$ that does not exist in $lastTrgSt$ and matches $srcEl$; if so, the algorithm generates a corresponding assertion (lines #39–41).

Finally, the algorithm adds all the assertions that it is able to migrate to the list of migrated assertions (lines #44–46) and return them (line #48).

F. Applying Algorithm 2 to the Example

In this section, we illustrate how algorithm 2 can migrate the assertions in the source test (Fig. 1) to the target app (Fig. 2)

in our motivating example.

APPTESTMIGRATOR first triggers the migrated source events in the target app, thus reaching Screen *e* in Fig. 2. The technique must then identify in this screen the elements checked by the assertions, shown on Screen *d* in Fig. 1. Because it is unable to do so, the technique randomly selects an actionable element in the target app. Assume that at some point it selects the "Navigate up" button (e→f), which takes the target app to screen *f* in Fig. 2. The technique again tries to find matches for the elements in the new GUI state, and this time it is successful. It then checks the property of the conditions in the assertions, which are all "Displayed" (i.e., all *UI-based*). Therefore, APPTESTMIGRATOR generates the assertions using the matched elements and using "Displayed" as the condition to check.

G. Test Encoder

Once APPTESTMIGRATOR has processed all source events and assertions, the *Test encoder* generates actual test cases for the target app based on the migrated events and assertions.

V. EMPIRICAL EVALUATION

To evaluate our approach, we implemented APPTESTMIGRATOR and investigated the following research questions:

1) **RQ1:** How accurate is APPTESTMIGRATOR in migrating events from source to target apps?
2) **RQ2:** Is APPTESTMIGRATOR more effective than GUITESTMIGRATOR in migrating test events?
3) **RQ3:** How accurate is APPTESTMIGRATOR in migrating oracles from source to target apps?

A. Implementation

Our implementation supports Android apps, as Android is one of the major platforms in the mobile app market. APPTESTMIGRATOR requires as input tests for the source app, and our current implementation supports tests written using the Espresso testing framework [3]. We chose Espresso for several reasons, including the fact that it is widely used and is actively maintained by Google, provides easy access to a more complete GUI state, is integrated with Android Studio's Espresso Test Recorder, and supports asynchronous tasks. We modified Espresso to collect relevant dynamic information during test execution. Note, however, that our general approach is not specific to Android and Espresso and could be ported to other mobile platforms and testing frameworks. For our static analysis, we leverage gator [33], a static analysis tool that creates a model of the GUI-related behavior of an Android app. We used Neo4j [26], a graph database management system, to interact with the static model. To generate our Word2Vec model, we used Genism [30], an open-source vector space and topic modeling toolkit implemented in Python.

B. Evaluation Setup

To evaluate APPTESTMIGRATOR, we first identified app categories in the Google Play Store containing at least four apps with over 1,000 installs and source code available.[2] This resulted in many categories, including address book, diet tracking, expense tracking, food ordering, mail clients, music, news, note taking, online shopping, shopping list, to-dos management, and weather apps. We then excluded those categories that are too broad to provide a standard set of features (e.g., games and fitness). Among the resulting categories, we randomly selected four: *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather* apps. Finally, for each of these four categories, we selected the four apps either with the most test cases or randomly, in case there were not enough apps with test cases in a category.

TABLE II: Description of our benchmark apps and tests.

ID	Name	LOC	Installs	#Test cases	#Events	#Oracles	Coverage
S1	Shopping List	6.7K	10,000+	11	61	20	67%
S2	Shopping List	8.2K	100,000+	11	44	12	60%
S3	Shopping List	18.2K	5,000+	10	44	16	54%
S4	OI Shopping List	24.9K	1,000,000+	16	71	20	51%
N1	Note Now	7.7K	1,000+	8	35	16	53%
N2	Swiftnotes	5.7K	5,000+	11	56	24	68%
N3	Writeily Pro	9.3K	5,000+	9	49	12	57%
N4	Pocket Note	13.4K	1,000+	10	50	13	50%
E1	EasyBudget	14.1K	50,000+	10	53	11	52%
E2	Expenses	5.2K	1,000+	8	43	10	90%
E3	Daily Budget	7.1K	10,000+	10	49	14	69%
E4	Open Money Tracker	14.5K	1,000+	10	70	27	56%
W1	Forecastie	8.6K	10,000+	9	44	13	53%
W2	Good Weather	11.6K	5,000+	9	40	19	59%
W3	World Weather	18.5K	1,000+	8	60	6	59%
W4	Geometric Weather	37.5K	10,000+	8	28	8	30%

In total, we found two apps with GUI test cases: one in the *Shopping List* category (six test cases), and one in the *Note Taking* category (four test cases). We then asked eight CS students not involved in this research but familiar with (when not expert in) testing to write more test cases for all the apps using BARISTA [10]—a test record and replay tool that allows users to generate tests in a visual and intuitive way. We first introduced the participants to the BARISTA tool and gave them some time to become familiar with it and ask us questions about it. We then asked each participant to write test cases for two randomly selected apps.

Table II shows the list of the apps and tests we used. For each app, the table shows its ID, name, size (LOC), number of installations, and number of test cases, in addition to the total number of events, the total number of oracles, and the statement coverage for the test cases considered. Note that, although we considered only open-source apps, these apps are available in the Google Play Store and have been installed at least 1,000 times, as mentioned above, which should provide some confidence in their quality and popularity.

We applied APPTESTMIGRATOR to these apps and test cases by considering each app in a category as the source app and the remaining apps as target apps. We also compared APPTESTMIGRATOR with its most closely related technique, GUITESTMIGRATOR [5] (GTM, for brevity, in the rest of this section). To do so, while reducing the cost of the manual check of the results, we randomly selected half of the possible combinations of source and target apps and ran GTM on them.

[2]Because the current implementation of APPTESTMIGRATOR supports test cases written in Espresso, the app source code is required to build and run test cases. However, our technique could be directly applied to binary apps by targeting a different testing framework.

TABLE III: Results of migrating test cases using APPTESTMIGRATOR (ATM) (both events and oracles) and GTM (events only).

Source app	Target app	Completely migrated		Partially migrated		Correctly matched			Unmatched (!exist)			Unmatched (exist)			Incorrectly matched		
						ATM		GTM	ATM		GTM	ATM		GTM	ATM		
		ATM	GTM	ATM	GTM	Events	Oracles	Events	Events	Oracles	Events	Events	Oracles	Events	Events	Oracles	Events
S1	S2	54%	27%	46%	36%	63%	65%	31%	18%	15%	17%	4%	0%	35%	15%	20%	17%
S1	S3	55%	-	45%	-	65%	65%	-	20%	25%	-	3%	5%	-	12%	5%	-
S1	S4	36%	-	46%	-	30%	30%	-	35%	35%	-	8%	30%	-	27%	5%	-
S2	S1	45%	-	45%	-	68%	84%	-	22%	8%	-	0%	0%	-	10%	8%	-
S2	S3	55%	36%	36%	27%	71%	83%	48%	20%	17%	15%	2%	0%	22%	7%	0%	15%
S2	S4	55%	36%	45%	27%	76%	75%	42%	10%	25%	18%	2%	0%	17%	12%	0%	23%
S3	S1	60%	30%	40%	30%	68%	88%	40%	20%	6%	23%	5%	6%	23%	7%	0%	14%
S3	S2	40%	30%	50%	50%	64%	75%	45%	14%	0%	14%	9%	19%	23%	13%	6%	18%
S3	S4	30%	-	20%	-	28%	50%	-	36%	0%	-	18%	44%	-	18%	6%	-
S4	S1	44%	-	31%	-	49%	50%	-	28%	40%	-	7%	0%	-	16%	10%	-
S4	S2	50%	-	40%	-	47%	45%	-	21%	25%	-	18%	25%	-	14%	5%	-
S4	S3	44%	44%	25%	31%	45%	55%	44%	32%	25%	33%	3%	20%	7%	20%	0%	16%
Avg.	-	47%	34%	39%	34%	56%	64%	42%	23%	18%	20%	7%	12%	21%	14%	6%	17%
N1	N2	50%	-	50%	-	69%	81%	-	17%	19%	-	0%	0%	-	14%	0%	-
N1	N3	63%	50%	37%	37%	74%	50%	57%	9%	44%	9%	0%	0%	11%	17%	6%	23%
N1	N4	62%	38%	25%	37%	51%	50%	44%	11%	38%	8%	6%	0%	23%	32%	12%	25%
N2	N1	82%	-	18%	-	88%	100%	-	9%	0%	-	0%	0%	-	3%	0%	-
N2	N3	64%	-	36%	-	73%	79%	-	7%	13%	-	7%	4%	-	13%	4%	-
N2	N4	73%	55%	18%	45%	77%	88%	52%	9%	0%	9%	2%	4%	27%	12%	8%	12%
N3	N1	67%	-	11%	-	64%	58%	-	18%	8%	-	0%	0%	-	18%	34%	-
N3	N2	33%	0%	33%	0%	55%	42%	14%	23%	42%	23%	4%	8%	40%	18%	8%	23%
N3	N4	33%	33%	44%	44%	52%	17%	52%	23%	8%	23%	2%	17%	2%	23%	58%	23%
N4	N1	50%	40%	30%	30%	53%	84%	49%	30%	8%	32%	0%	0%	11%	17%	8%	8%
N4	N2	10%	-	30%	-	32%	0%	-	36%	8%	-	15%	84%	-	17%	8%	-
N4	N3	40%	-	20%	-	41%	54%	-	38%	16%	-	4%	15%	-	17%	15%	-
Avg.	-	52%	36%	29%	32%	61%	59%	45%	19%	17%	17%	3%	11%	19%	17%	13%	19%
E1	E2	50%	30%	50%	40%	58%	36%	43%	19%	36%	15%	0%	0%	17%	23%	28%	25%
E1	E3	50%	-	50%	-	47%	27%	-	34%	9%	-	4%	36%	-	15%	28%	-
E1	E4	50%	30%	30%	30%	47%	55%	45%	26%	0%	32%	0%	9%	10%	27%	36%	13%
E2	E1	62%	38%	38%	50%	74%	70%	55%	11%	0%	16%	2%	10%	7%	13%	20%	22%
E2	E3	75%	50%	0%	37%	38%	70%	45%	40%	0%	40%	0%	0%	13%	22%	30%	2%
E2	E4	62%	-	25%	-	69%	50%	-	18%	20%	-	0%	0%	-	13%	30%	-
E3	E1	70%	-	0%	-	53%	43%	-	31%	29%	-	8%	7%	-	8%	21%	-
E3	E2	60%	-	10%	-	47%	36%	-	47%	43%	-	0%	0%	-	6%	21%	-
E3	E4	50%	30%	20%	30%	67%	21%	41%	13%	36%	16%	10%	7%	31%	10%	36%	12%
E4	E1	40%	-	0%	-	37%	11%	-	28%	52%	-	16%	26%	-	19%	11%	-
E4	E2	30%	10%	30%	40%	43%	33%	31%	33%	30%	33%	4%	26%	22%	20%	11%	14%
E4	E3	50%	-	30%	-	36%	44%	-	33%	30%	-	13%	26%	-	18%	0%	-
Avg.	-	54%	31%	24%	38%	51%	41%	43%	28%	24%	25%	5%	12%	17%	16%	23%	15%
W1	W2	44%	-	44%	-	55%	15%	-	9%	46%	-	7%	31%	-	29%	8%	-
W1	W3	44%	33%	44%	44%	68%	15%	43%	20%	62%	25%	0%	0%	18%	12%	23%	14%
W1	W4	22%	-	67%	-	36%	0%	-	34%	85%	-	5%	15%	-	25%	0%	-
W2	W1	44%	22%	56%	67%	63%	11%	35%	25%	68%	50%	2%	0%	7%	10%	21%	8%
W2	W3	44%	-	44%	-	52%	16%	-	32%	37%	-	3%	5%	-	13%	42%	-
W2	W4	33%	0%	22%	67%	30%	16%	18%	35%	74%	45%	15%	10%	20%	20%	0%	17%
W3	W1	50%	-	37%	-	47%	33%	-	43%	50%	-	0%	17%	-	10%	0%	-
W3	W2	38%	-	50%	-	30%	17%	-	17%	50%	-	2%	0%	-	51%	33%	-
W3	W4	25%	25%	50%	50%	20%	17%	30%	37%	66%	43%	5%	17%	15%	38%	0%	12%
W4	W1	50%	-	37%	-	64%	50%	-	22%	50%	-	0%	0%	-	14%	0%	-
W4	W2	37%	37%	50%	50%	61%	50%	61%	14%	25%	14%	7%	12%	7%	18%	13%	18%
W4	W3	50%	50%	37%	37%	71%	38%	71%	11%	62%	11%	0%	0%	0%	18%	0%	18%
Avg.	-	40%	28%	45%	52%	50%	23%	43%	25%	56%	31%	4%	9%	11%	21%	12%	15%

We then compared the accuracy of migrating events by GTM with that of APPTESTMIGRATOR. Because GTM does not support the migration of oracles, we could not compare that part of our approach.

C. Results

Table III shows the results obtained through manual inspection of the migrated test cases by APPTESTMIGRATOR (both events and oracles) and GTM (events only). The results for the combinations of source and target apps not considered (see Section V-B) are indicated with a dash, "–". The first and second columns of the table show the ID of the source and target apps, respectively. Columns 3 (*completely migrated*) and 4 (*partially migrated*) show the percentage of test cases for which the technique generated complete and partial target test cases, respectively. Completely migrated tests are those for which all events are successfully migrated. Partially migrated tests are those that are not completely migrated but for which at least one event is successfully migrated.

The fifth column (*correctly matched*) indicates the percentage of individual events in the source tests that were correctly matched to events in the target app. To get a better understanding of the performance of the techniques, we manually inspected the events (or oracles) that were not successfully matched and classified them in one of three categories: (1) *unmatched (!exist)* events (oracles) are events (oracles) in the source test that could not be matched to a corresponding event (oracle) in the target app because they actually do not have a counterpart in that app (i.e., true negatives); (2) *unmatched (exist)* events (oracles), conversely, represent events (oracles) in the source test that have a counterpart in the target app, but were nevertheless not migrated (i.e., false negatives); finally, *incorrectly matched* events (oracles) are events (oracles) that

TABLE IV: Benchmarks assertions statistics.

Category	Property (if applicable)	Total	Percentage
UI-based			
	Text	110	46%
	Displayed	84	35%
	Enabled	16	7%
	CompletelyDisplayed	8	4%
	Checked	5	2%
	Focus	4	1%
Hierarchy-based			
	Child	4	1%
DoesNotExist		4	1%
Others		6	3%

were mapped to the wrong events (oracles) in the target app (i.e., false positives). Table III shows the results with respect to this further classification, in Columns 6 to 8.

Note that, initially, we considered measuring APPTESTMIGRATOR's effectiveness also in terms of coverage and fault-detection ability of the migrated tests. We ultimately decided against it because we believe that these metrics make little sense in the context of test migration; the degree of coverage and fault-revealing ability of the migrated tests not only depends on the number and variety of the source tests, but also on the specifics of the code on which they run, which makes a meaningful computation of these metrics extremely difficult. A migrated test that exercises an important feature that only represents 1% of the source code, for instance, can be very useful but is not going to affect much the overall coverage or fault revealing ability of the migrated test suite. In other words, we believe that these metrics provide little to no information on the effectiveness of the technique itself, which is what we are interested in assessing.

D. RQ1: Accuracy in Migrating Events

As Table III shows, on average, APPTESTMIGRATOR *completely migrated* 47%, 52%, 54%, and 40% of the tests considered, and *partially migrated* 39%, 29%, 24%, and 45% of the tests considered, for the *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather* categories, respectively. This corresponds to correctly matching 56%, 61%, 51%, and 50% of the events in the four categories. Among the unmatched events, 23%, 19%, 28%, and 25% were true negatives, 7%, 3%, 5%, and 4% were false negatives, and 14%, 17%, 16%, and 21% were false positives.

In summary, on average, APPTESTMIGRATOR completely migrated 48% and partially migrated 34% of the tests considered, while correctly matching 54% of the events. Of the unmatched 46% events, 24% were true negatives, 5% were false negatives, and 17% were false positives.

E. RQ2: Comparison with GTM

As Table III shows, on average, GTM completely migrated 34%, 36%, 31%, and 28% of the tests considered, and partially migrated 34%, 32%, 38%, and 52% of the tests considered, for the *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather* categories, respectively. This corresponds to correctly matching 42%, 45%, 43%, and 43% of the events in the four categories. Among the unmatched events, 20%, 17%, 25%, and 31% were true negatives, 21%, 19%, 17%, and 11% were false negatives, and 17%, 19%, 15%, and 15% were false positives.

In summary, on average, GTM completely migrated 16% fewer tests than APPTESTMIGRATOR (32% versus 48%) and partially migrated 5% more tests than APPTESTMIGRATOR (39% versus 35%). Also on average, GTM correctly matched 11% less event than APPTESTMIGRATOR (43% versus 54%). Finally, the percentages of true negatives, false negatives, and false positives for GTM are 23%, 17%, and 16%, which are 1% less, 12% more, and 1% less than the corresponding results for APPTESTMIGRATOR.

To better understand which aspects of APPTESTMIGRATOR allowed it to outperform GTM in most cases, we manually inspected the cases labeled as *correctly matched* for APPTESTMIGRATOR but not for GTM. We found that the new approach for computing similarity scores, the use of static analysis, and the new crawling algorithm helped APPTESTMIGRATOR to successfully migrate the events for which GTM failed in 41%, 36%, and 23% of the cases, respectively.

F. RQ3: Accuracy in Migrating Oracles

Table IV shows the properties that are checked by the assertions in the tests we considered in our evaluation. As the table shows, the most checked property is *Text* (46%), followed by *Displayed* (35%), *Enabled* (7%), *CompletelyDisplayed* (4%), *Checked* (2%), *Focus* (1%), *Child* (1%), and *DoesNotExist* (1%). The assertions in category *Others* check whether the elements in an *AdapterView* [1] have specific names using custom APIs that are not currently supported by our implementation.

As the results in Table III show, APPTESTMIGRATOR correctly matched 64%, 59%, 41%, and 23% of the assertions in the *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather* categories, respectively. Among the unmatched assertions, 18%, 17%, 24%, and 56% were true negatives, 12%, 11%, 12%, and 9% were false negatives, and 6%, 13%, 23%, and 12% were false positives.

In summary, on average, APPTESTMIGRATOR correctly matched 47% of the assertions in the test cases. Of the remaining 53%, 29% were true negatives (assertions that APPTESTMIGRATOR could not match and that in fact had no counterpart in the target app), 11% were false negatives (assertions that APPTESTMIGRATOR could not match but had a counterpart in the target app), and 13% were false positives (assertions without a counterpart in the target app that APPTESTMIGRATOR matched incorrectly). Whereas fault negatives simply make a migrated test potentially less useful, as developers would have to manually check its results, false positives may result in erroneous assertions and, ultimately, in erroneous test outcomes. This issue could be addressed by using the technique as a recommender system, as it is typical for this kind of automated approaches: APPTESTMIGRATOR would propose to the developers the migrated test cases, so that they would have a chance to check them before use.

VI. Threats To Validity

The primary threat to the external validity of our results concerns whether they will generalize to other apps, tests, and categories. To mitigate this issue, we used randomly selected real-world apps from four different categories. A first threat to internal validity is that the students who wrote some of the tests used in our evaluation were not familiar with the apps under test. However, it is not uncommon for testers to test software they did not develop themselves. A second threat to internal validity consists of possible mistakes in the manual inspection of the test migration results. Despite the many differences in the GUI design of apps within the same category, however, the checks were time consuming but ultimately straightforward, due to the similarity in the functionality behind the GUIs.

VII. Related Work

A. GUI Test Migration

AppTestMigrator builds on the idea and vision we proposed in earlier work [4] and extends GUITestMigrator [5], our technique for migrating test cases between student apps developed based on identical specifications. As we discussed in the Introduction, AppTestMigrator addresses several limitations of GUITestMigrator that limit its effectiveness when migrating tests between apps that share only part of their functionality. It does so by using a new crawling algorithm, a more powerful approach for computing similarity scores, and static analysis for better exploring the search space. Furthermore, unlike GUITestMigrator, AppTestMigrator can also migrate test oracles.

Concurrently to our development of AppTestMigrator, Lin, Jabbarvand, and Malek developed CraftDroid [17], which also aims to migrate tests, including oracles, across mobile apps that share part of their functionality. Since AppTestMigrator and CraftDroid have similar goals, both build on and extend some of the ideas and vision proposed in our and others' earlier work [4,5,29], and rely on similar techniques (e.g., Word2Vec models to match GUI elements, combination of static and dynamic analysis), we plan to perform an empirical comparison between the two approaches in future work.

Rau, Hotzkow, and Zeller propose a technique for generating more effective GUI tests by transferring tests across web applications [28,29]. Besides targeting mobile instead of web apps, our technique is different from theirs in several ways. First, their matching approach exploits only textual labels, while our technique takes into account additional features and can also handle icons without any associated text. Second, their dynamic crawling algorithm is different from ours and does not use any static-analysis information. Finally, unlike AppTestMigrator, their approach does not migrate oracles.

TestMig [27] migrates GUI tests between iOS and Android apps. The goal of their work is different, as AppTestMigrator migrates GUI test cases between similar apps, whereas TestMig migrates GUI test cases for apps meant to have the same functionality across different platforms (iOS to Android).

B. GUI Test Repair

There has been much research on GUI test repair. These techniques focus on repairing GUI test scripts on different versions of the same software [7,8,11,12,14–16,18,22,23,32,34]. AppTestMigrator differs from these techniques, as our goal is to migrate test cases between apps, rather than repairing tests during software evolution. In fact, these techniques could not be readily applied in our context, whereas our approach could be used for GUI test repair as well.

C. GUI Test Generation

Recently, several researchers have defined techniques that exploit commonalities among apps to generate more effective GUI tests. Polariz [20] generates test scripts from crowd-based tests by extracting cross-app reusable event sequences. Augusto [21] generates semantic UI tests that target popular functionality. AppFlow [13] leverages machine learning to automatically recognize common screens and widgets and synthesize reusable GUI Tests. Ermuth and Pradel [9] propose a UI-level test generation approach that exploits execution traces of human users to automatically create complex sequences of events. None of the above approaches exploits commonalities in functionality to migrate test cases between apps.

VIII. Conclusion and Future Work

We presented AppTestMigrator, a technique for migrating test cases between mobile apps that share part of their functionality. We implemented a prototype of AppTestMigrator for Android apps and test cases written using the Espresso framework, and used it to migrate tests between 16 randomly selected apps in four different app categories. Overall, AppTestMigrator fully migrated 48% of the tests considered and partially migrated 34% of them. For 42% of the fully migrated tests, it also fully migrated their oracles. In addition, AppTestMigrator improves on the state of the art in terms of both effectiveness (higher migration success) and functionality (ability to migrate oracles).

In future work, we will evaluate AppTestMigrator on additional benchmarks to confirm our initial results. We will also investigate ways to generalize the approach to other GUI-based software, such as web apps. As we discussed in the Introduction, we will also explore the idea of creating a "test store" that, when developers submit an app, analyzes the app, looks for similar ones in the store, tries to migrate tests from these apps, and returns all the tests that were successfully migrated. Such a test store could operate in parallel with a traditional app store and automatically provide test cases that developers could run on their apps before submitting them to the app store.

Acknowledgments

We thank the students who helped with our empirical evaluation for their time. This work was partially supported by the National Science Foundation under grants CCF-1161821 and 1548856.

REFERENCES

[1] Android Open Source Project, "AdapterView," 2019, https://developer.android.com/reference/android/widget/AdapterView.

[2] ——, "Communicate with the UI thread," 2019, https://developer.android.com/training/multiple-threads/communicate-ui.

[3] ——, "Espresso," 2019, https://developer.android.com/training/testing/espresso/.

[4] 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.

[5] ——, "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 '18. New York, NY, USA: ACM, 2018, pp. 164–175.

[6] ——, "App Test Migrator," 2019, https://sites.google.com/view/apptestmigrator/.

[7] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "WATER: Web Application TEst Repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ser. ETSE '11. New York, NY, USA: ACM, 2011, pp. 24–29.

[8] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezzè, "Automated GUI Refactoring and Test Script Repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ser. ETSE '11. New York, NY, USA: ACM, 2011, pp. 38–41.

[9] 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 '16. New York, NY, USA: ACM, 2016, pp. 82–93.

[10] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '17. New York, NY, USA: IEEE, 2017, pp. 149–160.

[11] C. Fu, M. Grechanik, and Q. Xie, "Inferring Types of References to GUI Objects in Test Scripts," in *Proceedings of the International Conference on Software Testing Verification and Validation*, ser. ICST '13. New York, NY, USA: IEEE, 2009, pp. 1–10.

[12] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and Evolving GUI-directed Test Scripts," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 408–418.

[13] G. Hu, L. Zhu, and J. Yang, "Appflow: Using Machine Learning to Synthesize Robust, Reusable UI Tests," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '18. New York, NY, USA: ACM, 2018, pp. 269–282.

[14] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing GUI Test Suites Using a Genetic Algorithm," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 245–254.

[15] P. Li and E. Wohlstadter, "View-based Maintenance of Graphical User Interfaces," in *Proceedings of the 7th International Conference on Aspect-oriented Software Development*, ser. AOSD '08. New York, NY, USA: ACM, 2008, pp. 156–167.

[16] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "ATOM: Automatic Maintenance of GUI Test Scripts for Evolving Mobile Applications," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '10. New York, NY, USA: IEEE, March 2017, pp. 161–171.

[17] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test Transfer Across Mobile Apps Through Semantic Mapping," in *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '19. New York, NY, USA: ACM, 2019, p. To appear.

[18] A. D. Lucia, R. Francese, G. Scanniello, G. Tortora, and N. Vitiello, "A Strategy and an Eclipse Based Environment for the Migration of Legacy Systems to Multi-tier Web-based Architectures," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, ser. ICSME '06. New York, NY, USA: IEEE, 2006, pp. 438–447.

[19] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Association for Computational Linguistics (ACL) System Demonstrations*, 2014, pp. 55–60.

[20] K. Mao, M. Harman, and Y. Jia, "Crowd Intelligence Enhances Automated Mobile Testing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17. New York, NY, USA: ACM, 2017, pp. 16–26.

[21] 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.

[22] A. M. Memon, "Automatically Repairing Event Sequence-based GUI Test Suites for Regression Testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 1–36, Nov. 2008.

[23] A. M. Memon and M. L. Soffa, "Regression Testing of GUIs," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 118–127, Sep. 2003.

[24] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *CoRR*, vol. abs/1301.3781, pp. 1–10, 2013.

[25] G. A. Miller, "Wordnet: A Lexical Database for English," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[26] Neo4j, Inc., "Neo4j," 2019, https://neo4j.com.

[27] 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.

[28] A. Rau, J. Hotzkow, and A. Zeller, "Efficient GUI Test Generation by Learning from Tests of Other Apps," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 370–371.

[29] ——, "Transferring Tests Across Web Applications," in *Web Engineering*, T. Mikkonen, R. Klamma, and J. Hernández, Eds. Cham: Springer International Publishing, 2018, pp. 50–64.

[30] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, 2010, pp. 45–50.

[31] E. S. Ristad and P. N. Yianilos, "Learning String-Edit Distance," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 5, pp. 522–532, May 1998.

[32] S. Staiger, "Static Analysis of Programs with Graphical User Interface," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, ser. CSMR '07. New York, NY, USA: IEEE, 2007, pp. 252–264.

[33] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static Window Transition Graphs for Android," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. New York: IEEE, 2015, pp. 658–668.

[34] S. Zhang, H. Lü, and M. D. Ernst, "Automatically Repairing Broken Workflows for Evolving GUI Applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '13. New York, NY, USA: ACM, 2013, pp. 45–55.