Poster: Automated Test Migration for Mobile Apps

Farnaz Behrang
College of Computing
Georgia Tech
Atlanta, GA
behrang@gatech.edu

Alessandro Orso
College of Computing
Georgia Tech
Atlanta, GA
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 cost of mobile app testing, we propose AppTestMigrator, a technique that allows for migrating test cases between apps with similar features. 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). Typical examples of this situation are apps in the same category, apps developed based on the same specification, and different versions of the same app. In all these cases, the burden of writing test cases can be reduced by migrating test cases written for an app to another, similar app. Given a test case for an app (source app) and a second app (target app), AppTestMigrator attempts to automatically transform the sequence of events in the test for the source app to events that can be consumed by the target app. We implemented AppTestMigrator for Android mobile apps and evaluated our approach on four randomly selected shopping list apps from the Google Play Store. Our initial results are promising and motivate further research in this direction.

ACM Reference Format:
Farnaz Behrang and Alessandro Orso. 2018. Poster: Automated Test Migration for Mobile Apps. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195019

1 MOTIVATION AND INTUITION

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. Unfortunately, developing test cases for an app, as for any software system in general, is extremely expensive.

We believe that the cost of testing mobile apps can be considerably reduced by considering similarities between apps and reusing test cases across similar apps. Specifically, this work is motivated by the intuition that, although different apps can differ dramatically, there are many cases in which groups of apps share much of their functionality and may as a consequence provide conceptually similar graphical user interfaces (GUIs). Examples of this situation are apps in the same category (e.g., banking apps). Other examples are apps developed based on the same, or at least similar, requirements. (In the educational context, in particular, it is typical for students to be asked to develop an app that satisfies a given set of requirements.) A final, somehow simpler example consists of different versions of the same app, where the underlying functionality is typically mostly unchanged, and so is the corresponding GUI.

Based on our intuition, we developed AppTestMigrator, a technique that can decrease the amount of effort involved in testing a given app by leveraging and adapting existing tests for other similar or related apps.

2 APPROACH

Figure 1 shows an overview of our approach. Given the test cases of the source app, the *Scenario Extractor* first extracts *source scenarios* from these test cases, where each source scenario is a sequence of (GUI) events together with an assertion that performs a check at the end of the sequence. In this context, we define a GUI event as a triple (a, t, i), where a is the action that corresponds to the event (e.g., *click*), t is the target of the action (e.g., a specific *button*), and i is the input value (e.g., data for an `EditText` box).

Given the extracted source scenarios and the source code of both source and target apps, the *Scenario Matcher* tries to compute *target scenarios*—sequences of events in the target app that match the source scenarios. To do so, for each source scenario ss and target app ta, AppTestMigrator processes the events in ss in the order in which they appear and tries to match them one at a time with corresponding events in ta, while dynamically crawling ta. The matching is performed mostly based on the textual attributes of the widgets that are the targets of the events (e.g., the label of a button). To improve this process, AppTestMigrator builds an ontology for both source and target apps using word embedding [2], which extends general purpose lexical databases (e.g., WordNet [3]) with specific domain knowledge and allows for a more effective matching. In addition, AppTestMigrator takes advantage of a statically computed state-flow graph to (1) prune the search space and (2) account for the possible future states of the app when selecting which events to trigger.

When the *Scenario Matcher* cannot find a direct match for an event, it searches for other states in ta in which a match may be possible by generating extra events. These extra events are selected randomly among all the possible events for the ta in its current state and are considered only if they modify the ta's state. This process continues until either a match is found, and thus AppTestMigrator can continue with the next event in ss, or no match can be found, and AppTestMigrator backtracks to earlier matches and tries to look for alternative matches. If AppTestMigrator is successful in matching all events in ss, it generates a *complete target scenario.*

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195019

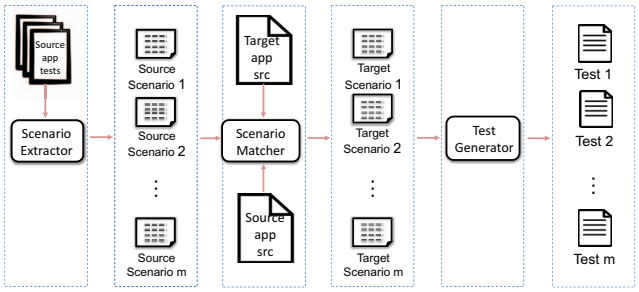

Figure 1: Overview of AppTestMigrator.

Otherwise, it produces a *partial target scenario*, that is, a target scenario that contains only a subsequence of the events in *ss*.

The set of identified (complete and partial) target scenarios is the input to the *Test Generator*, which simply encodes them as test scripts for the target app for which the scenarios were created.

3 PRELIMINARY EVALUATION

To evaluate AppTestMigrator, we implemented a prototype that supports Android mobile apps and test cases written using the Espresso testing framework [1]. As benchmarks for our evaluation, we randomly selected four shopping list apps from the Google Play Store. We randomly selected one of the apps to be the source app; the other three apps were the target apps for the approach. We then asked a graduate student majoring in computer science (and not involved in this research) to write test cases for the source app. The student wrote 11 test cases that achieved, overall, 67% statement coverage. Finally, we applied AppTestMigrator to these source and target apps and test cases.

AppTestMigrator first extracted 14 source scenarios from the 11 test cases for the source app, with the number of events in the scenarios ranging from 2 to 14 (and a mean of 6). It then migrated the source scenarios to the target apps. Table 1 shows our results for the three target apps, listed in the first column. Columns two (completely migrated) and three (partially migrated) show the percentage of test scenarios for which AppTestMigrator was able to generate complete and partial target scenarios, respectively. The fourth column (correctly matched) indicates instead the percentage of individual events in the source scenarios that AppTestMigrator correctly matched to events in the target app. As the table shows, AppTestMigrator was able, on average, to completely migrate 43% of the source scenarios and partially migrate 52% of the source scenarios. This corresponds to 54% of the events in the scenarios being correctly matched by AppTestMigrator.

To get a better understanding of the performance of AppTestMigrator, we analyzed the events that were not successfully matched and classified them in one of three categories: (1) *unmatched (!exist)* events are events in the source scenario that AppTestMigrator could not match to a corresponding event in the target app because they actually do not have a counterpart in that app (i.e., true negatives); (2) *unmatched (exist)* events, conversely, represent events in the source scenario that have a counterpart in the target app, but that AppTestMigrator was nevertheless unable to map (i.e., false negatives); finally, *incorrectly matched* events are events in the source scenario that were mapped by AppTestMigrator to the wrong events in the target app (i.e., false positives).

target app	completely migrated	partially migrated	correctly matched	unmatched (!exist)	unmatched (exist)	incorrectly matched
App1	43%	57%	59%	20%	6%	15%
App2	50%	50%	64%	21%	3%	12%
App3	36%	50%	40%	26%	7%	27%
Avg.	43%	52%	54%	23%	5%	18%

Table 1: Evaluation results.

Table 1 also shows the results for this further classification (Columns 5–7). As the table shows, among the unmatched events (46%), 23% were true negatives, 5% false negatives, and 18% false positives. Our manual investigation of these events showed that the (few) false negatives were mostly caused by the use of radically different terminology for the same action in source and target apps. Our word embedding technique allows AppTestMigrator to match many, but not all of these cases. As for the false positives, they occurred mainly because of the use of the same term in different contexts. Finally, the majority of the true negatives involved operations on app settings, which tend to be considerably different for the different apps (at least in the cases we considered).

Overall, we believe that our results, albeit still preliminary, are promising. Being able to automatically and completely migrate almost half of the test cases from an app to another has the potential to dramatically reduce the cost of app testing. Moreover, we believe that there is room for improving our technique and making it even more effective.

4 CONCLUSION AND FUTURE WORK

We presented AppTestMigrator, a technique that aims to decrease the cost of app testing by leveraging the similarities between GUIs of different yet related apps and performing test migration—reusing and adapting test cases among such apps. AppTestMigrator can be used in all those situations in which different apps share similarities that result in conceptually similar GUIs.

We implemented a prototype of AppTestMigrator that supports Android apps and tests written using the Espresso framework, and we have used our prototype to evaluate our approach on four randomly selected shopping list apps from the Google Play Store. We believe that our results, although still preliminary, show clear evidence of the potential usefulness of our approach.

In future work, we will first perform additional experiments to validate our findings and guide future research. Second, we will extend the technique so that it can migrate test oracles (i.e., assertions), in addition to events. Finally, we will investigate large scale applications of our approach, in the context of a database of apps and tests for these apps. Developers could submit their apps to the database, which would analyze them, look for similar apps, migrate the tests from these apps, and return all the tests that were successfully migrated. If successful, this could result in a sort of test store that operates in parallel to the app store.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation under grants CCF-1161821 and 1548856.

REFERENCES

[1] Espresso 2018. (2018). https://developer.android.com/training/testing/espresso/.
[2] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013).
[3] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (Nov. 1995), 39–41.