

# AppTestMigrator: A Tool for Automated Test Migration for Android Apps\*

Farnaz Behrang  
behrang@gatech.edu  
Georgia Institute of Technology  
Atlanta, GA, USA

Alessandro Orso  
orso@cc.gatech.edu  
Georgia Institute of Technology  
Atlanta, GA, USA

## 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. In this demo, we present APPTESTMIGRATOR, a technique and tool for migrating test cases between apps with similar functionality. 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 attempts to automatically transform the sequence of events and oracles in a test case for an app (source app) to events and oracles for another app (target app). The results of our preliminary evaluation show the effectiveness of APPTESTMIGRATOR in migrating test cases between mobile apps with similar functionality.

**Video URL:** <https://youtu.be/WQnfEcwYqa4>

## CCS CONCEPTS

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

## KEYWORDS

GUI testing, test migration, mobile apps

## ACM Reference Format:

Farnaz Behrang and Alessandro Orso. 2020. AppTestMigrator: A Tool for Automated Test Migration for Android Apps. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion)*, May 23–29, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3377812.3382149>

## 1 INTRODUCTION

Most of the software systems we use every day provide a graphical user interface (GUI) through which users can access the different features provided by the underlying software. Although unit and integration testing may directly target this underlying software, testing of GUI-based applications is usually performed mainly through their GUIs. This is particularly true for web and mobile apps, where the GUI represents a fundamental part of the system. It is therefore important to thoroughly test these apps to gain confidence that they behave as intended when used in the field. However, manually

developing test cases for an app tends to be extremely expensive, as it involves considerable human effort.

The technique and tool described in this paper are motivated by the observation that, although GUIs for different apps can differ dramatically, there are many cases in which the apps share similarities that result in conceptually similar GUIs. A typical example of this situation is 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. In this case, we believe it is possible to decrease the amount of effort involved in GUI testing by leveraging and adapting, when performing GUI testing of a given app, existing GUI tests for other similar apps.

This paper summarizes our technique for migrating test cases between apps that share part of their functionality, which we defined in previous work [7] and presents APPTESTMIGRATOR, a tool that implements our technique. 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.

To evaluate APPTESTMIGRATOR, we used our tool to migrate test cases for apps in four categories: *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather*. Specifically, we selected four apps within each category and applied APPTESTMIGRATOR to these apps and 158 test cases by considering each app in a category as the source app and the remaining apps as target apps. We then manually inspected the migrated test cases. Overall, APPTESTMIGRATOR fully migrated 48% and partially migrated 34% of the tests considered. For 42% of the fully migrated tests, APPTESTMIGRATOR also fully migrated their oracles.

This paper is organized as follows. Section 2 describes APPTESTMIGRATOR and summarizes the technique behind it. Section 3 presents details of our tool's implementation. The evaluation of APPTESTMIGRATOR and related work are presented in Sections 4 and 5. Finally, we conclude in Section 6.

## 2 APPTTESTMIGRATOR TECHNIQUE

Fig. 1 shows an overview of APPTESTMIGRATOR's workflow and architecture. As the figure shows, APPTESTMIGRATOR takes as input the *source app*, the *source tests* (a set of tests for the source app), the *target app*, and a set of user guides for various mobile apps in different categories. (A user guide contains a set of instructions that users must perform to cover different user scenarios in a given app). As output, APPTESTMIGRATOR produces the *target tests* (the source

\*This demo describes the implementation of a technique we presented at ASE 2019 [7].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICSE '20 Companion, May 23–29, 2020, Seoul, Republic of Korea

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-7122-3/20/05.

<https://doi.org/10.1145/3377812.3382149>

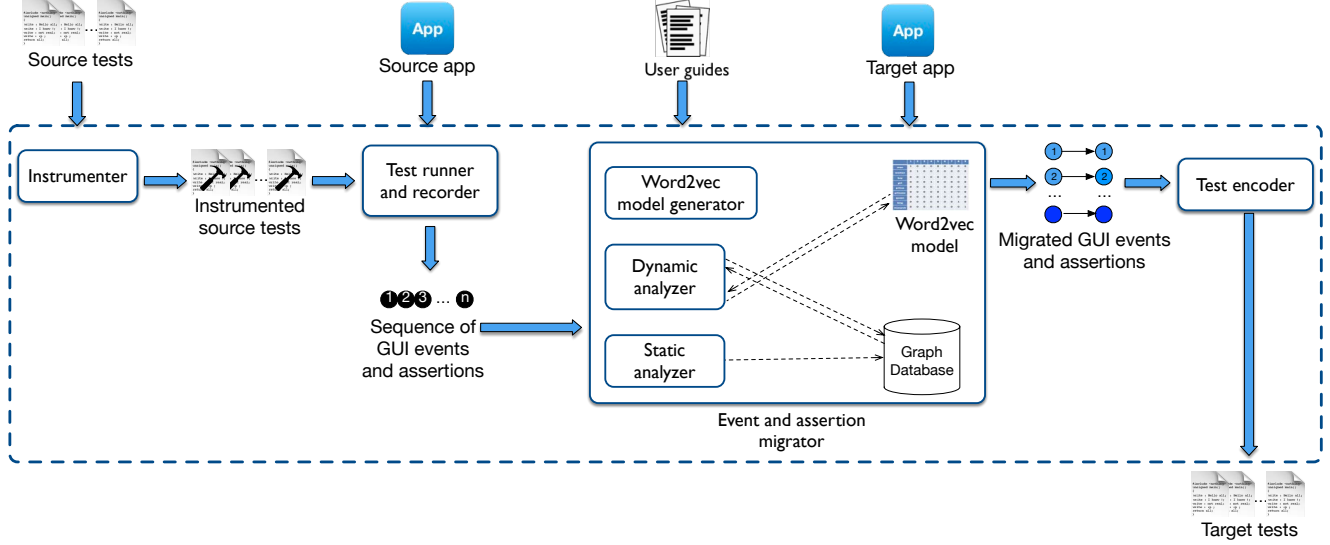


Figure 1: Overview and Architecture of APPTeSTMigrator.

tests migrated to the target app). APPTeSTMigrator consists of four main modules: *Instrumenter*, *Test runner and recorder*, *Event and 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 and assertion migrator* then migrates 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 of APPTeSTMigrator’s modules in more detail.

**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.

**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.

**Event and Assertion Migrator.** Given a sequence of source events extracted from a source test and a target app, this module tries to migrate the source events and assertions to the target app. To do so, it processes the events and assertions in the order in which they appear and tries to match them one at a time with corresponding events in target app *TA*, while dynamically crawling *TA*. The matching is performed based on the textual attributes of the widgets that are the targets of the events (e.g., the label of a button).

To allow more effective matching, APPTeSTMigrator creates an ontology for mobile apps based on word embeddings it generates using the Word2Vec [13] methodology. 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 *Event and Assertion Migrator* cannot find a direct match for an event or assertion, 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 *TA* in its current state, and are considered only if they modify *TA*’s state. This process continues until either a match is found, and thus APPTeSTMigrator can continue with the next event, or no match can be found, and APPTeSTMigrator backtracks to earlier matches and tries to look for alternative matches.

**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. If successful in matching all events and assertions, APPTeSTMigrator generates a *complete target test*. Otherwise, it produces a *partial target test*, that is, a test that contains only a subsequence of the events and assertions.

### 3 APPTeSTMigrator Implementation

Our implementation of APPTeSTMigrator supports Android apps and tests written using the Espresso testing framework [3]. We chose Android because it is one of the major platforms in the mobile app market. 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. Note, however, that our general approach is not specific to Android and Espresso and could be ported to other mobile platforms and testing frameworks. In the rest of this section, we discuss the tool implementation for each of the modules in figure 1.

#### 3.1 Instrumenter and Test Runner and Recorder

We modified the Espresso framework and instrumented the APIs relevant to GUI interactions such that, when the tests are executed,

APPTMIGRATOR can collect information about GUI events and assertions. For GUI events, APPTMIGRATOR collects the action that corresponds to the event (e.g., click), the target of the action (e.g., a button), and input values (e.g., data for a text-input box), if any. The targets of the actions are GUI elements that are represented by a set of properties (e.g., text) and values that are associated with the properties. More specifically, we instrumented the following APIs in the Espresso testing framework: *perform*, *check*, *matches*, *doesNotExist*, *withId*, *withText*, *withContentDescription*, *withHint*, *withSpinnerText*, *isDisplayed*, *isCompletelyDisplayed*, *isEnabled*, *isClickable*, *isChecked*, *withParent*, *withChild*, *hasDescendant*, *isDescendantOfA*, and *hasSibling*.

### 3.2 Word2Vec Model Generator

Given a set of randomly selected user guides in PDF format for mobile apps in different categories, the *Word2Vec model generator* uses PDF2TEXT [1] to convert PDF documents into plain text files. It then preprocesses the raw text by tokenizing it and removing stop-words and special-characters using NLTK [10], which is an open-source Python library for Natural Language Processing. Given the preprocessed text corpus, the model generator learns a word embedding—a model that maps each word in the vocabulary to a numeric vector. Intuitively, the distance between two vectors that represents two words represent the semantic distance between those two words. In other words, the closer two words are in the vector space corresponding to the word embedding, the more likely they are to have similar meaning. To generate the Word2Vec model, our current implementation of the *Word2Vec Model Generator* uses Genism [19], an open-source vector space and topic modeling toolkit implemented in Python. In order to serve the pre-trained Word2Vec model as a service, we built a REST API using Flask [4].

### 3.3 Static Analyzer

To perform static analysis of the apps, the *Static Analyzer* leverages gator [22], a static analysis tool that creates a model of the GUI-related behavior of an Android app. Specifically, the *Static Analyzer* leverages gator to compute a *Window Transition Graph* (WTG) for the target app—a statically-computed graph in which 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 *Static Analyzer* extracts the WTG for the target app as a JSON file and uses Neo4j [15], a graph database management system, to store it. Using Neo4j allows for an easier interaction between the *Dynamic Analyzer* and the static model.

### 3.4 Dynamic Analyzer

The *Dynamic Analyzer* leverages the Espresso testing framework to explore target apps dynamically. During dynamic exploration, the *Dynamic Analyzer* extracts, for each screen, the GUI state of the app, represented as a set of GUI elements. To compute the similarity score between two GUI elements, the *Dynamic Analyzer* uses Stanford’s CoreNLP library [11], which allows it to preprocess the textual information associated with the GUI elements, namely, tokenization and lemmatization.

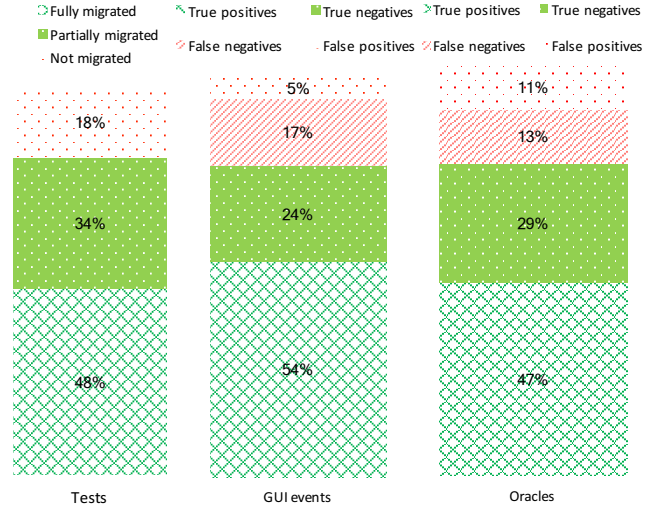


Figure 2: Results of migrating test cases using APPTMIGRATOR.

In order to retrieve data from the graph database, the *Dynamic Analyzer* queries graph data using Cypher [14], which is Neo4j’s graph query language. Cypher is a declarative, SQL-inspired language for describing visual patterns in graphs and allows the user to state what to select, insert, update, or delete from the graph data. Finally, the *Dynamic Analyzer* uses OkHttp [20], an HTTP client library, to perform REST calls to both the Word2Vec model and the graph database.

### 3.5 Test Encoder

To generate actual test cases for the target app, the *Test Encoder* identifies the most suitable type of selector for every migrated GUI element. To do so, it checks whether the element has properties “resource ID” and “Text” and whether it is possible to select the element using one or both of them. If so, the *Test Encoder* creates a corresponding selector. Otherwise, it generates a selector of type XPath [21], that is, a path expression that identifies a specific element in the GUI state. It is worth noting that we extended the Espresso APIs to include the notion of XPath selector. It is also worth noting that, since selectors that use “resource ID” or “Text” are more readable compared to an XPath selector, they are considered first when generating the selectors.

## 4 EVALUATION

In this section, we summarize our evaluation of APPTMIGRATOR. For further details, please refer to the paper that presents the APPTMIGRATOR technique in detail [7]. To evaluate APPTMIGRATOR, we identified four app categories in the Google Play Store: *Shopping List*, *Note Taking*, *Expense Tracking*, and *Weather* apps. We then selected four apps within each category. For each app, we considered the test cases contained in the apps and also asked students not involved in this research but familiar with testing to write more test cases for all the apps (158 test cases in total). We applied APPTMIGRATOR to these apps and test cases by considering each app in a category as the source app and the remaining apps as target apps.

Figure 2 shows the results of the study, which we validated through manual inspection of all the migrated test cases. The bar on the left-hand side of the figure shows the percentage of test cases for which the technique generated fully and partially migrated test cases, along with the percentage of test cases that APPTESTMIGRATOR was not able to migrate. The bars in the middle and on the right-hand side of the figure show the percentage of individual events (resp., oracles) in the source tests that APPTESTMIGRATOR correctly matched (true positives), could not match to a corresponding event (resp., oracle) in the target app because they did not actually have a counterpart in that app (true negatives), did not migrate to an existing corresponding event (resp., oracle) in the target app (false negatives), and mapped to the wrong events (resp., oracles) in the target app (false positives).

Overall, APPTESTMIGRATOR fully migrated 48% and partially migrated 34% of the tests considered. For 42% of the fully migrated tests, APPTESTMIGRATOR also fully migrated their oracles. In terms of individual events and oracles, on average, the true positives, true negatives, false positives, and false negatives were 54%, 24%, 17%, and 5% for the individual events, and 47%, 29%, 13%, and 11% for the individual oracles.

## 5 RELATED WORK

APPTESTMIGRATOR builds on the vision and ideas that we proposed in earlier work [5, 6] and implements the technique we presented in [7]. CRAFTDROID [9] was developed concurrently to [7] and also aims to migrate tests, including oracles, across mobile apps that share part of their functionality.

Rau, Hotzkow, and Zeller proposed a technique for generating more effective GUI tests by transferring tests across web applications [17, 18]. Besides targeting mobile instead of web apps, our technique is different from theirs in terms of event-matching approach and underlying analysis. Also, unlike APPTESTMIGRATOR, their approach does not migrate oracles.

TestMig [16] is a technique for migrating 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).

Recently, several approaches have also been proposed that share similar intuitions with APPTESTMIGRATOR and propose approaches for exploiting commonalities in the functionality of different apps to improve GUI test generation (e.g., [8, 12]).

## 6 CONCLUSION

We presented APPTESTMIGRATOR, a tool for migrating test cases between mobile apps that share part of their functionality. We used APPTESTMIGRATOR to migrate test cases between 16 randomly selected apps in 4 different app categories. Our evaluation shows that APPTESTMIGRATOR can be effective in migrating both test inputs and oracles. Our demo illustrates APPTESTMIGRATOR by showing its performance on an example.

## 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, DARPA, under contracts FA8650-15-C-7556 and FA8650-16-C-7620, and gifts from Google, IBM Research, and Microsoft Research.

## REFERENCES

- [1] AKS-Labs. 2020. PDF2TEXT. <https://www.pdf2txt.com/>.
- [2] Android Open Source Project. 2020. Communicate with the UI thread. <https://developer.android.com/training/multiple-threads/communicate-ui>.
- [3] Android Open Source Project. 2020. Espresso. <https://developer.android.com/training/testing/espresso/>.
- [4] Armin Ronacher. 2020. Flask. <https://palletsprojects.com/p/flask/>.
- [5] Farnaz Behrang and Alessandro Orso. 2018. Automated Test Migration for Mobile Apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE '18)*. ACM, New York, NY, USA, 384–385.
- [6] Farnaz Behrang and Alessandro Orso. 2018. 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 (ISSTA '18)*. ACM, New York, NY, USA, 164–175.
- [7] Farnaz Behrang and Alessandro Orso. 2019. Test Migration Between Mobile Apps with Similar Functionality. In *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering (ASE '19)*. ACM, New York, NY, USA, 54–65.
- [8] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. 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 (ESEC/FSE '18)*. ACM, New York, NY, USA, 269–282.
- [9] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering (ASE '19)*. ACM, New York, NY, USA, 42–53.
- [10] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1 (ETMTNLP '02)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 63–70.
- [11] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. Association for Computational Linguistics, Stroudsburg, PA, USA, 55–60.
- [12] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. 2018. Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 280–290.
- [13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013), 1–10.
- [14] Neo4j, Inc. 2020. Cypher Query Language. <https://neo4j.com/developer/cypher-query-language/>.
- [15] Neo4j, Inc. 2020. Neo4j. <https://neo4j.com>.
- [16] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: Migrating GUI Test Cases from iOS to Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. ACM, New York, NY, USA, 284–295.
- [17] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. 2018. Efficient GUI Test Generation by Learning from Tests of Other Apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE '18)*. ACM, New York, NY, USA, 370–371.
- [18] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. 2018. Transferring Tests Across Web Applications. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 50–64.
- [19] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50.
- [20] Square, Inc. 2020. OkHttp. <https://square.github.io/okhttp/>.
- [21] W3C. 2020. XML Path Language. <https://www.w3.org/TR/xpath-30/>.
- [22] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. 2015. Static Window Transition Graphs for Android. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE, New York, 658–668.