APIMigrator: An API-Usage Migration Tool for Android Apps*

Mattia Fazzini
University of Minnesota
Minneapolis, MN, USA
mfazzini@umn.edu

Qi Xin
Georgia Institute of Technology
Atlanta, GA, USA
qxin6@gatech.edu

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

ABSTRACT

To provide their functionality, mobile apps interact extensively with the application programming interface (API) of the underlying operating system. Given that this API evolves frequently, app developers are periodically required to migrate API usages in their apps to ensure that the apps behave as expected when running on the new API. To help developers with this tedious, error-prone, and time-consuming task, we defined a technique for automated API migration and implemented it in a tool called APIMigrator that supports Android apps. APIMigrator (1) automatically migrates API usages within an app by leveraging how developers of other apps migrated corresponding API usages and (2) validates the migrations through differential testing. We evaluated APIMigrator on a benchmark of 15 real-world apps and obtained promising results. Overall, our tool was able to migrate 85% of the API usages considered and validate 68% of these migrations. We provide a demo video of the tool at https://youtu.be/v0VfpKi_IDc.

CCS CONCEPTS

• **Software and its engineering → Software evolution**.

KEYWORDS

Mobile apps, API-usage migration, API analysis

ACM Reference Format:
Mattia Fazzini, Qi Xin, and Alessandro Orso. 2020. APIMigrator: An API-Usage Migration Tool for Android Apps. In *IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3387905.3388608

1 INTRODUCTION

To build mobile apps, developers rely heavily on the application programming interface (API) provided by the underlying operating system (OS). This API allows developers to build feature-rich apps without worrying about complex and low-level implementation details encapsulated in the OS.

To provide important improvements and new functionality, APIs change over time. The API of the Android OS is an egregious case of this phenomenon, as it has evolved at an average rate of 115 API changes per month [17]. Considering that methods in newer versions of the API are routinely deprecated, eliminated, added, and changed, Android developers must regularly migrate their apps to the newer API to take advantage of new functionality and ensure that their apps behave as expected on the new API. Additionally, due to the extensive fragmentation of the Android ecosystem [8, 14], developers must also perform backward-compatible migrations, so that their apps can adequately function also on older APIs.

Although API changes are usually reported in the API documentation, in a great majority of cases the documentation does not provide relevant examples on how to perform required migrations, and developers must understand how to do so on their own [12]. Additionally, given the extensive use of the API within apps, the required changes can be widespread. As a result, the task of migrating apps to support the new version of the API is typically tedious, time-consuming, and error-prone [12, 15].

To address this challenging maintenance task, we devised a technique [3] that migrates API usages in an app (i.e., calls to the API), and implemented the technique in a tool called APIMigrator that supports Android apps. In this demo paper, we summarize the technique and present APIMigrator. The high-level intuition behind our technique is that it is possible to automatically migrate API usages by leveraging the information contained in the codebase of other apps. Our technique takes as inputs a target app and summary information about the changes in the API (i.e., a mapping describing the changes in the calls to the API, which can be extracted from the API documentation). Given these inputs, the techniquefi nds the API usages that require migration, searches for corresponding migration examples, transforms the examples into generic migration patches, uses the patches to migrate the target app, and validates the migrations through differential testing. The outputs of the technique are the evolved target app and a report documenting the migrations performed.

Although other techniques and tools that perform migrations based on examples exist (e.g., [1, 18, 20]), they mostly target repetitive changes within the same codebase or require examples to be provided as inputs. Our technique differs from these approaches as it automatically identifies examples across different codebases, prioritizes the examples by comparing them, and uses differential testing to perform a sanity check and provide more confidence in the migrations performed.

To assess the usefulness of APIMigrator, we performed an empirical evaluation on a benchmark of 15 real-world apps that contain 20 API usages occurring at 41 different locations. We believe our results are promising: APIMigrator successfully migrated 85% of the API usages considered and validated 68% of the migrations. APIMigrator is publicly available at https://doi.org/10.5281/zenodo.3668385.

*This demo illustrates the implementation of a technique presented at ISSTA 2019 [3].

MOBILESoft '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7959-5/20/05…$15.00
https://doi.org/10.1145/3387905.3388608

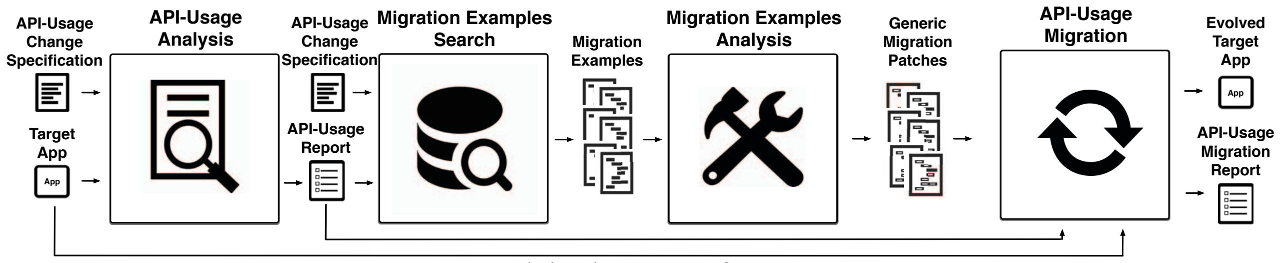
API-Usage Change Specification
Target App
App
API-Usage Analysis
API-Usage Change Specification
API-Usage Report
Migration Examples Search
Migration Examples
Migration Examples Analysis
Generic Migration Patches
API-Usage Migration
Evolved Target App
App
API-Usage Migration Report

Figure 1: High-level overview of APIMigrator.

2 TERMINOLOGY

Before describing the technique and APIMigrator, we introduce some necessary terminology. Given two consecutive API versions, *old API* = $[m_1, ..., m_k]$ and *new API* = $[m'_1, ..., m'_l]$, we define an *API usage* as any call of one or more methods of either the old or the new API versions. Intuitively, an API-usage change (*AU change*) between the old and the new versions of the API can typically be described as an API-usage change mapping (*AU cm*) between one or more methods in the old version and one or more methods in the new version: $[m_1, ..., m_p] \rightarrow [m'_1, ..., m'_q]$.

To illustrate, we present an example of an *AU change* taken from the Android API [6]. In this example, method `getAllNetworkInfo()` was deprecated in version 23 of the API in favor of the two new methods `getAllNetworks()` and `getNetworkInfo(Network)`. The corresponding *AU cm* would be from method `getAllNetworkInfo()` to methods `getAllNetworks()` and `getNetworkInfo(Network)`.

Given an AU change, we define an *old API usage* (resp., *new API usage*) for that AU change as a sequence of one or more method invocations that match the left-hand side (resp., right hand side) of the *AU cm*. Considering an app and an AU change, we use the term *migration* to indicate the operation of migrating an old API usage in the app to a new API usage. Finally, we use the term *migration example* to refer to existing migrations (e.g., in an app code base).

3 TECHNIQUE OVERVIEW

In this section, we summarize our technique. Full details can be found in [3]. Figure 1 provides a high-level overview of our technique. As shown in thefi gure, the approach takes as inputs a target app and a specification of the AU changes, where the latter consists of a set of AU changes. Given these inputs, the technique operates in four phases and generates as outputs (i) an evolved target app and (ii) an API-usage migration report that describes the changes in the app. We now summarize the four phases of the approach.

API-Usage Analysis. Given the target app and the specification of the AU changes, thefi rst phase of the technique statically analyzes the source code of the app to identify API usages that should be changed and stores this information in the API-usage report. Specifically, the approach (i) identifies old API usages in the app, (ii) checks whether the corresponding API calls can execute on the new version of the API, and (iii), if so, stores the information of the API usages and the location of their API calls in the API-usage report, as these API usages require migration.

Migration Example Search. After identifying which API usages require migration, the approach tries tofi nd migration examples by analyzing how other developers migrated corresponding API usages in their apps. To this end, the technique analyzes various app repositories (repos, in short) that are publicly available in a code-hosting infrastructure (e.g., GitHub [5]).

For each API usage that requires migration, the approach quickly searches for code bases that could contain migration examples by performing a textual search in thefi les that are present in the code-hosting infrastructure considered. The search is based on a set of keywords extracted from the new API usage. Specifically, these keywords are (i) the name, (ii) the parameter types, and (iii) the declaring class of each API call in the new API usage.

At this point, the technique considers the history of eachfile (returned by the search) in the corresponding repo to identify migration examples. Specifically, the approach compares each version of thefi le (new version) with its previous version (old version) tofind a method in the code of the app that performed a migration from the old API usage to the new API usage. The technique analyzes the differences between the twofi les and uses this information to search for a method that (i) does not contain the new API usage in the old version, (ii) uses the old API usage in the old version, (iii) uses the new and the old API usages in the new version. Furthermore, the new and old API usages must be in different branches of a condition that checks the API version on which the app is executing. This latter check allows for identifying only backward-compatible migrations. Note that, although the technique tries tofi nd multiple migration examples to produce a more general solution, it can be applied also when a single example is found.

Migration Examples Analysis. In this phase, the approach generalizes migration examples into *generic migration patches* that can be used to migrate the target app. When more than one patch is available for the same AU change, the technique also prioritizes the patches based on how closely related they are to the "common core" shared across examples. In this way, the approach is able to favor patches that best capture the essence of the migration.

a) Generic Migration Patch Generation. For each migration example, the technique considers the method containing the migration and initiates the generalization task by identifying a list of edit operations that transforms the API usages in the old version of the code (old method) to the API usages in the new version of the code (new method). Specifically, the technique leverages an approach [4, 18] that generates an ordered list of tree operations (i.e., Delete, Insert, Move, and Update) for transforming the statements in the abstract syntax tree (AST) of the old method into the AST statements of the new method. Because developers of other apps might include, between two versions, additional modifications unrelated to the AU change, the technique performs an intra-procedural dependency analysis to select only the relevant edits.

The edits extracted so far are dependent on the specific example from which they were computed. To use these edits for migrating the methods of the target app, the technique transforms the edits into generic edits and identifies the context in which the edits can be used. A *generic edit* consists of three parts: (i) the original edit, (ii) the position in the AST of the statement affected by the edit, and (iii) the *abstraction* of the statement affected by the edit, where the abstraction is the statement itself with variables replaced by their type. The context of the edits is defined by *context variables*, that is, variables that are used by the statements affected by the edits and are not defined in any of the statements. These variables are computed through dependency analysis. Generic edits and context variables define the content of a generic migration patch.

b) Generic Migration Patch Prioritization. After translating migration examples into generic migration patches, the approach groups patches according to the AU change they address and prioritizes them based on how closely related they are to the common core of edits shared across patches of the same group. The approach determines the core byfi nding a solution to the multiple longest common subsequence problem instantiated over the list of edit abstractions of each patch and ranking higher examples that have more edits in the core. Thefi nal output of this part of the technique is the list of patches, one for each API usage that requires migration.

API-Usage Migration. This phase of the technique (i) leverages the generic migration patches to migrate the old API usages at the locations reported by thefi rst phase of the approach and (ii) validates the migrations through differential testing. The technique performs one API-usage migration task at a time to avoid validation issues caused by migrations that interfere with each other. If two or more API-usages share dependencies, they are considered together.

For each code location l (and corresponding method m) that requires migration, the approach starts from the patch at the top of the list (i.e., the patch closest to the migration core). It then considers (in order) other patches if the current one cannot be applied in l. A patch is applicable in l when (i) it is possible to map the context variables of the patch to the variables in m, and (ii) generic edits can be applied (in their entirety) to the AST of m. To apply a patch, the technique iterates over the statements in m and tries to identify a mapping between the context variables and the variables in scope at the considered statement (by solving an instance of the assignment problem). If itfi nds a mapping, the approach tries to perform the edit operations associated with the generic edits to the AST of m. If all the operations can be applied successfully, the approach validates the migrated method through differential testing (either leveraging an existing test suite or building a new one through random input generation).

After processing all the locations that require migration, this phase produces as thefi nal output of the technique an evolved target app and an API-usage migration report, where the technique documents validated and applicable migrations.

4 TOOL DESCRIPTION

We implemented our technique in a tool, called APIMIGRATOR, written in Python and Java. We developed and tested APIMIGRATOR on a machine running Ubuntu 16.04 and created a virtual machine containing the tool, which can be downloaded at https://doi.org/10.5281/zenodo.3668385. In the video demo associated with this paper, we show how to use APIMIGRATOR to perform an API-usage migration task using the virtual machine.

The tool has three main modules: the *analysis module*, the *search module*, and the *migration module*. These three modules implement the capabilities described in the API-usage analysis phase, the migration-examples-search phase, and the migration-examples-analysis and API-usage migration phases of the technique, respectively. We created a single module for the last two phases to avoid serializing and de-serializing a large number of in-memory data structures. We now describe each module in more detail.

Analysis Module. This module identifies API-usages that require migration in the target app and is built on top of IctApiFinder [8]. It computes the versions on which each statement can execute using Soot [11] and Doop [22]. APIMIGRATOR also uses Soot to identify API-usages that require migration.

Search Module. This module searches for migration examples on GitHub [5]. The module uses the GitHub API to perform its keyword-based search, which identifies source codefi les from app repos that may contain migration examples. This part of the module uses Python to invoke the API. After performing the keyword-based search, the module analyzes the source codefi les retrieved by the search by downloading the corresponding repos locally. The module uses the type solver from JavaParser [10] to recognize the signature of the method calls characterizing API usages in codefiles.

Migration Module. This module analyzes the source code of the migration examples to extract and apply generic migration patches. The module takes as inputs the sourcefi les from the app repos, in the form of Eclipse projects, and uses the Eclipse JDT API to build and analyze the AST of the methods considered. The module leverages the LASE [19] infrastructure to encode AST nodes into statements and extract edits from a pair of ASTs. Furthermore, APIMIGRATOR leverages the Crystal [9] framework to perform dependency analysis on the statements in the ASTs. Finally, APIMIGRATOR uses Monkey [7] to create tests for validating migrations automatically, in case a test suite is not available.

5 EVALUATION

In this section, we summarize the evaluation of APIMIGRATOR (for additional details, see [3]). To evaluate APIMIGRATOR, we measured (i) the percentage of migrations it can perform on a set of benchmark apps and (ii) the cost of running the tool. We used 15 real-world apps from the F-Droid repository [2] as benchmarks. Specifically, we used three sets offi ve apps with two characteristics. *First*, the three sets contain apps designed for three different versions of the API. For each API version, we manually generated the API-usage change specification by studying the corresponding API documents. Note that generating the specification took us less than an hour. Moreover, these specifications must be computed only once for each new API version and can be shared across developers. *Second*, using the generated API-usage change specification, we made sure that each app contained at least one API usage that (i) was different from those in the other apps and (ii) had to be migrated in the subsequent API version. Overall, the 15 apps considered contain 20 different API usages requiring migration, and these API usages occur 41 times in the apps.

Results: APIMigrator was able to migrate 17 (85%) of the 20 API usages and 37 (90%) of their 41 occurrences. Furthermore, out of the 37 migration it performed, APIMigrator automatically validated 25 (68%) of them. We manually analyzed all of the 37 migrations and confirmed that they were correct, according to the changes described in the corresponding API documents. APIMigrator was not able to migrate one of the 20 API usages (one occurrence) because it could notfi nd migration examples, and two of the 20 API usages (three occurrences) because the migration examples had edits spanning multiple methods (APIMigrator currently considers only edits fully contained in a single method). (In [3], we also show APIMigrator's effectiveness over LASE [19].)

In the evaluation, the running time of APIMigrator was dominated by the *Migration Examples Search* phase (about 10 hours on average, whereas all other phases took less than one minute, on average). Note that, even if the search phase takes a few hours to complete, (i) it can be performed overnight, (ii) must be run only once per version release, (iii) its results can be shared across developers. Moreover, the search can be further improved via offline repository indexing and can be run in parallel.

6 RELATED WORK

APIMigrator primarily relates to example-based program transformation techniques (e.g., [1, 13, 16, 18, 20, 21, 23]). APIMigrator differs from some of these techniques [1, 16, 18, 20, 21] in that it uses a more general approach to identify when a migration patch is applicable. This makes our tool effective in using examples from different codebases. As opposed to A4 [13], APIMigrator identifies migration examples in remote repositories, handles changes in return values, and is able to prioritize migration examples. Finally, unlike Meditor [23], APIMigrator can analyze and compare multiple examples while performing migrations, allowing for prioritizing and using migration patches that are closer to the essence of the required changes. Additionally, our tool also performs differential testing to validate the migrations performed.

7 CONCLUSION

We presented APIMigrator, a tool for automatically migrating API-usages in Android apps. APIMigrator identifies migration examples from publicly available codebases, analyzes the examples to generate and rank migration patches, and validates the performed migrations through differential testing. We used APIMigrator to migrate the API usages in a benchmark of 15 real-world apps. The tool was able to migrate 85% of the API usages considered and automatically validated 68% of these migrations. In future work, we plan to extend APIMigrator's evaluation and investigate how to support migrations across method boundaries. We also plan to study ways to compute API-usage change specifications automatically, using the version control history of the API and its documentation. Finally, we plan to investigate ways to extend our technique so that it can handle API migrations in more general settings (i.e., for other languages and libraries).

ACKNOWLEDGMENTS

This work was partially supported by NSF, under grants CCF-1161821 and CCF-1563991, DARPA, under contracts FA8650-15-C-7556 and FA8650-16-C-7620, ONR, under contract N00014-17-1-2895, and gifts from Facebook, Google, IBM Research, and Microsoft Research.

REFERENCES

[1] Georg Dotzler, Marius Kamp, Patrick Kreutzer, and Michael Philippsen. 2017. More Accurate Recommendations for Method-level Changes. In *Proceedings of the 2017 meeting on foundations of software engineering*. ACM, Paderborn, Germany, 798–808.

[2] F-Droid 2018. *F-Droid*. Retrieved April 7, 2020 from https://f-droid.org

[3] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-Usage Update for Android Apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, Beijing, China, 204–215.

[4] Beat Fluri, Michael Wursch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-grained Source Code Change Extraction. *IEEE Transactions on software engineering* 33 (2007), 725–743.

[5] GitHub 2019. *GitHub*. Retrieved April 7, 2020 from https://github.com

[6] Google 2019. *ConnectivityManager*. Retrieved April 7, 2020 from https://developer.android.com/reference/android/net/ConnectivityManager#getAllNetworkInfo()

[7] Google. 2019. *UI/Application Exerciser Monkey*. Retrieved April 7, 2020 from https://developer.android.com/studio/test/monkey

[8] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and Detecting Evolution-induced Compatibility Issues in Android Apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, Montpellier, France, 167–177.

[9] Ciera Jaspan, Kevin Bierhoff, and Jonathan Aldrich. 2009. *Crystal-izing Sophisticated Code Analyses*. Retrieved April 7, 2020 from https://code.google.com/archive/p/crystalsaf

[10] JP 2019. *JavaParser*. Retrieved April 7, 2020 from https://javaparser.org

[11] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*.

[12] Maxime Lamothe and Weiyi Shang. 2018. Exploring the Use of Automated API Migrating Techniques in Practice: An Experience Report on Android. In *Proceedings of the 15th International Conference on Mining Software Repositories*. Association for Computing Machinery, Gothenburg, Sweden, 503–514.

[13] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Chen. 2018. A4: Automatically Assisting Android API Migrations Using Code Examples. *CoRR* (2018).

[14] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-related Compatibility Issues in Android Apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Amsterdam, Netherlands, 153–163.

[15] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *Proceedings of the 2013 meeting on foundations of software engineering*. ACM, Saint Petersburg, Russia, 477–487.

[16] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H. Deng. 2017. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 229–246.

[17] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*. IEEE Computer Society, Eindhoven, Netherlands, 70–79.

[18] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, San Francisco, CA, USA, 502–511.

[19] Na Meng, Miryung Kim, and Kathryn S McKinley. 2013. *LASE Tool*. Retrieved April 7, 2020 from http://people.cs.vt.edu/nm8247/research.html

[20] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE, Buenos Aires, Argentina, 404–415.

[21] G. Santos, K. V. Paixao, N. Anquetil, A. Etien, M. de Almeida Maia, and S. Ducasse. 2017. Recommending source code locations for system specific transformations. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Klagenfurt, Austria, 160–170.

[22] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded*. Springer-Verlag, Berlin, Heidelberg, 245–251.

[23] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, Montreal, QC, Canada, 335–346.