Automatic Android Deprecated-API Usage Update by Learning from Single Updated Example

Stefanus A. Haryono*, Ferdian Thung*, Hong Jin Kang*, Lucas Serrano†, Gilles Muller‡, Julia Lawall‡, David Lo*, Lingxiao Jiang*
*School of Information Systems, Singapore Management University, Singapore
{stefanusah,ferdianthung,hjkang.2018,davidlo,lxjiang}@smu.edu.sg
†Sorbonne University/Inria/LIP6, France
Lucas.Serrano@lip6.fr
‡Inria, France
{Gilles.Muller,Julia.Lawall}@inria.fr

ABSTRACT

Due to the deprecation of APIs in the Android operating system, developers have to update usages of the APIs to ensure that their applications work for both the past and current versions of Android. Such updates may be widespread, non-trivial, and time-consuming. Therefore, automation of such updates will be of great benefit to developers. AppEvolve, which is the state-of-the-art tool for automating such updates, relies on having before- and after-update examples to learn from. In this work, we propose an approach named CocciEvolve that performs such updates using only a single after-update example. CocciEvolve learns edits by extracting the relevant update to a block of code from an after-update example. From preliminary experiments, we find that CocciEvolve can successfully perform 96 out of 112 updates, with a success rate of 85%.

KEYWORDS

API update, program transformation, Android, single example

1 INTRODUCTION

When an Android API is deprecated, apps using the API should update their usages of the API to ensure that they still work in the current and future versions of Android. For these updates, developers need to learn the new API(s) that should replace the deprecated API, while maintaining backward compatibility with older versions to address Android fragmentation [4, 10]. Moreover, the deprecated API may be used in multiple locations in a codebase. Thus, manually updating the deprecated API may be cumbersome and time consuming.

To help developers in updating usages of deprecated APIs with their replacement APIs, Fazzini et al. [3] proposed AppEvolve to automate the update task. AppEvolve learns to transform applications that use a deprecated API by learning from before- and after-update examples in GitHub. These updates add usages of replacement APIs around usages of the deprecated API along with conditional checks of Android versions in code. AppEvolve learns a generic patch from such examples and applies the transformation from each generic patch in a certain order.

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389285

```
if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.M) {
    hour = picker.getHour();
} else {
    hour = picker.getCurrentHour();
}
```

Figure 1: An example of an after-update for `getCurrentHour` deprecated API

Recently, Thung et al. [19] reported that, in order for AppEvolve to perform a successful update, the target code requiring update has to be written syntactically similar to the before- and after-update example. They demonstrated that AppEvolve's performance can be improved significantly if the app code is *manually* rewritten to have syntactic similarities to the before- and after-update example.

In this work, we propose CocciEvolve. CocciEvolve outperforms AppEvolve in the following aspects:

(1) CocciEvolve eliminates the shortcoming of AppEvolve by normalizing both the after-update example and the target app code to update. In this way, CocciEvolve ensures that both of them are written similarly, thereby preventing unsuccessful updates caused by minor differences in the way the code is written.
(2) CocciEvolve requires only a single after-update example for learning how to update an app that uses a deprecated API. Consider an after-update example in Figure 1. The `if` and `else` blocks correspond to the code using the replacement and deprecated API methods, respectively. The code in the `if` block runs only on versions of Android after the deprecation. Thus, the code in `else` block can be considered as the code that uses the deprecated method before the update.
(3) Transformations made by CocciEvolve are expressed in the form of a semantic patch by leveraging Coccinelle [8]. Semantic patch has a syntax similar to a *diff* that is familiar to software developers. Therefore, the modifications process are more readable and understandable to developers.

The contributions of our work are:

- We propose CocciEvolve, an approach for updating Android deprecated-API usages using only a single after-update example.

- We perform *automatic* code normalization of both the update example and the target code to be updated, addressing the challenge of updating code that is semantically equivalent but syntactically different, which was a limitation of prior work.
- We have evaluated CocciEvolve with a dataset of 112 target files to update that we obtained from Github. The 112 files use 10 deprecated APIs used in the original evaluation of AppEvolve. We show that CocciEvolve can successfully update 96 target files.

The remainder of this paper is structured as follows. Section 2 provides some preliminaries. Section 3 details our proposed approach. Section 4 describes our preliminary experiments and results. Section 5 presents related work. Finally, we conclude in Section 6.

2 PRELIMINARIES

AppEvolve. AppEvolve is the state-of-the-art tool automating API-usage updates for deprecated Android APIs. As input, it takes an app to update and a mapping from a deprecated method to its replacement method(s). It has four phases: *API-Usage Analysis*, *Update Example Search*, *Update Example Analysis*, and *API-Usage Update*.

In the *API-Usage Analysis* phase, AppEvolve finds uses of the deprecated method inside the app. In the *Update Example Search* phase, AppEvolve searches GitHub for apps that use both the deprecated and replacement methods in the same file. For each of these apps, AppEvolve searches the app commit history for a change that adds the replacement method(s) without removing the deprecated method. These changes are used to learn how to update deprecated method usages. In the *Update Example Analysis* phase, AppEvolve produces a generic patch from each example. AppEvolve then computes the common core from the produced generic patches. The common core is the longest subsequence of edits across the patches. In the *API-Usage Update* phase, AppEvolve applies generic patches in ascending order of their distance to the common core. To apply a generic patch, AppEvolve tries to match context variables to variables in the app. If matches are found, AppEvolve applies edits in the generic patch and returns the updated app if the edits are successful.

Coccinelle4J. Coccinelle4J [6] is a recent Java port of Coccinelle, which is a program matching and transformation tool [8, 14]. Given the source code of a program and a *semantic patch* describing the desired transformations, Coccinelle4J transforms the parts of the source code that match the semantic patch.

Written in the Semantic Patch Language (SmPL), a semantic patch has two parts: (1) context information, including the declaration of metavariables; and (2) changes to be made to the source code. A metavariable can match program elements of the type indicated in its declaration. Modifications are expressed using fragments of Java as follows: (1) code that should be removed by annotating the start of the lines with −; and (2) code that should be added by annotating the start of the lines with +. Unannotated lines add context to the semantic patch. Figure 2 showed an example of a semantic patch. The line surrounded by `@@` declares the metavariable.

3 APPROACH

An overview of our proposed system and the relevant pipelines are shown in Figure 3. CocciEvolve is built on three main components: (1) Source file normalization, (2) Updated API block detection, and (3) API-update semantic patch creation. These components are the building blocks of the CocciEvolve pipelines: a pipeline to create the update semantic patch, and a pipeline to apply the update to a target file. The create update patch pipeline takes as inputs the API usage change, and a source file containing updated API call. The apply update pipeline takes as inputs the API usage change, target source file, and the update semantic patch file. In the following subsections, we will explain in detail each of the system components.

```
@@
expression timepicker;
@@
- timepicker.getCurrentHour();
+ timepicker.getHour();
```

Figure 2: A simplified example of a semantic patch that transforms uses of the deprecated method, `getCurrentHour`

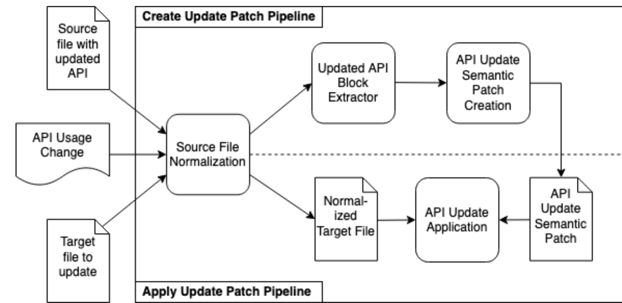

Figure 3: System overview of CocciEvolve

```
if (timePicker.getCurrentHour() > 11)
    itsNoon();

int currentHour = timePicker.getCurrentHour();
if (10 < currentHour)
    itsNoon();
```

Figure 4: Two fragments of semantically-equivalent code expressed differently

3.1 Source File Normalization

Different software developers may have different programming styles, thus, semantically-equivalent code may be expressed in different syntactic forms. As a result, equivalent usages of one API may vary in its expression . Figure 4 shows an example of such cases in which `getCurrentHour()` is expressed differently. Therefore, it is necessary to perform source code normalization.

Focusing on source code that is related to the API usage, we perform the following code normalization steps:

- An API call contained in a compound expression or statement (e.g. if, loop, return, etc) is extracted into a variable assignment.
- Arguments of an API call are extracted into variable assignments.
- The object receiving an API call is extracted into a variable assignment.

Two components are responsible for normalization in CocciEvolve:

3.1.1 Statement Extractor. For calls to API methods that return a value, the Statement Extractor extracts API calls that are part of compound expressions or statements. Before each of such a compound expression or statement, a new simple statement is inserted, which initializes a new temporary variable with the return value of the API call. This extraction is performed by using Coccinelle4J,

```
final String tmDevice;
+ String tempFunctionReturnValue;
+ tempFunctionReturnValue = tm.getDeviceId();
- tmDevice = "" + tm.getDeviceId();
+ tmDevice = "" + tempFunctionReturnValue;
```

Figure 5: Example of statement extraction for `getDeviceId` API call in a compound expression.

```
+ Context paramVar0 = context;
+ int paramVar1 = android.R.style.TextAppearance_Large;
+ TextView classNameVar = tvTitle;
- tvTitle.setTextAppearance(context,
-   android.R.style.TextAppearance_Large);
+ classNameVar.setTextAppearance(paramVar0, paramVar1);
```

Figure 6: Example of variable extraction for call to method `setTextAppearance`

with a semantic patch that is built from the type signature of the input API. An example of statement extraction can be seen in Figure 5.

3.1.2 Variable Extractor. The Variable Extractor is used to extract the arguments and object that the API call is invoked on. These arguments and object are assigned to temporary variables. As before, this extraction is done through Coccinelle4J using a semantic patch. An example of this extraction is shown in Figure 6.

3.2 Updated API Block Extractor

The Updated API Block Extractor identifies the relevant block of code containing the updated API call. This block is extracted and used as the source of the update patch. In order to prevent false positives due to irrelevant code blocks, we use the following rules to identify a valid updated API block:

(1) A valid updated block contains the updated API call in the `if` branch and the old API call in the `else` branch or vice versa.
(2) A valid updated block will have an Android version check as the `if` condition.

These classification rules are implemented in two components:

3.2.1 Version Statement Normalization. One of the important criteria for a valid updated block is the presence of an if statement that checks for the Android version. However, this check may take different forms in different projects (e.g. the Android version may first be assigned to a local variable). To alleviate this problem, CocciEvolve normalizes the conditions involving the Android version. This normalization is done automatically by first detecting any assignment of an Android version constant to a variable, and then replacing usage of the local variable in a condition with the relevant Android version constant. An example can be seen in Figure 7.

```
int currentBuildVersion = Build.VERSION.SDK_INT;
int marshmallowVersion = Build.VERSION_CODES.M;
- if (currentBuildVersion >= marshmallowVersion) {
+ if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    timePicker.setHour(1);
}
```

Figure 7: Sample Android version statement normalization

3.2.2 Update Block Extractor. The Update Block Extractor takes as an input the source file that has been normalized, then it detects a valid block based on the aforementioned criteria and extracts it from the file. This block is input to the API Update Semantic Patch Creation component.

3.3 API Update Semantic Patch Creation

Using the normalized update block as an input, this component creates a Coccinelle4J semantic patch that can be used to update a normalized target file. This component will replace variables and expressions with metavariables. Metavariables will bind to program elements in the input code passed into Coccinelle4J. The patch works by detecting the location of the old API call and then adding the new code which consists of a surrounding if block, the updated API call, and new variables introduced by the new API.

To increase the robustness of the system, for APIs that return a value, two different update rules are created. One rule is for cases where the return value is assigned into a variable, while the other is for cases where the return value is not used. An example of the update semantic patch can be seen in Figure 8.

```
@bottomupper@
expression exp0;
identifier classIden;
@@
TimePicker classIden = exp0;
...
+ if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
+ classIden.getMinute();
+ } else {
 classIden.getCurrentMinute();
+ }

@bottomupper_assignment@
expression exp0;
identifier classIden;
@@
TimePicker classIden = exp0;
...
+ if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
+ tempFunctionReturnValue = classIden.getMinute();
+ } else {
tempFunctionReturnValue = classIden.getCurrentMinute();
+ }
```

Figure 8: Example of update patch for `getCurrentMinute` API

4 EXPERIMENTS

4.1 Dataset

To assess the performance of CocciEvolve for practical usage, we use a dataset of real-world Android projects retrieved from public Github repositories. For this purpose, we use AUSearch [1], a tool for searching Github repositories, to find Android API usage. For the update semantic patch creation, we use the existing after-update examples provided in the AppEvolve replication package. For each API, only a single after-update example is used. We obtained a total of 112 target source files from Github for the 10 most commonly used APIs that were used in the original evaluation of AppEvolve. These target files are disjoint from the target files used by AppEvolve and thus are used to evaluate AppEvolve's generalizability in updating other target files. Detailed statistics for this dataset can be seen in Table 1. This dataset is published with our replication package.[1]

4.2 Experimental Settings

Our experiments are done by comparing the performance of CocciEvolve against AppEvolve based on the number of applicable

[1] https://sites.google.com/view/cocci-evolve/

Table 1: Number of targets in our evaluation dataset

API Description	# Targets
addAction(...)	0
getAllNetworkInfo()	8
getCurrentHour()	9
getCurrentMinute()	12
setCurrentHour(Integer)	12
setCurrentMinute(Integer)	10
setTextAppearance(...)	12
addGpsStatusListener(...)	0
fromHtml(...)	0
release()	14
removeGpsStatusListener(...)	0
shouldOverrideUrlLoading(...)	0
startDrag(...)	0
abandonAudioFocus(...)	0
getDeviceId()	12
requestAudioFocus(...)	8
saveLayer(...)	15
setAudioStreamType(...)	0
vibrate(long)	0
vibrate(long[], int)	0

updates produced. To generate the update patch, we utilize a single update example for each API from the available AppEvolve examples.

The target files for the experiments are the public Android project dataset that have been collected from Github through the use of AUSearch [1]. CocciEvolve is applied to every target file using the relevant API update patch that was created. For experiments involving AppEvolve, we need to configure each target project as an Eclipse project and create an additional XML file that contains the deprecated API description and their locations in the file. Due to this limitation, our experiments on AppEvolve are focused on first instance of each API call for each target project.

4.3 Results

In our experiments, CocciEvolve attains a better performance compared to AppEvolve. For most APIs, CocciEvolve achieves a near perfect result. We ask a software engineer with three years experience in Android, who was not part of this project, to validate the correctness of the update by verifying that there are no semantic changes in the update. Our experimental results are also included in our replication package.

In most cases, AppEvolve does not produce any code update. Thung et al. [19] note that AppEvolve requires some manual code refactoring and modifications to be able to perform the automated update. Table 2 shows the statistics of our evaluation.

Based on the experiments result, we can see that CocciEvolve mainly failed for two APIs: `getAllNetworkInfo()` and `requestAudioFocus(...)`. Updating these two APIs requires the creation of new objects for arguments to the replacement APIs. These objects are frequently created outside of the updated API block, and will require a data flow analysis to detect and construct the update correctly. The current version of CocciEvolve does not support sophisticated data-flow analysis.

Table 2: Statistics of updating target files per API

API	CocciEvolve		AppEvolve	
	Success	Fail	Success	Fail
getAllNetworkInfo()	0	8	0	8
getCurrentHour()	9	0	1	8
getCurrentMinute()	12	0	1	11
setCurrentHour(Integer)	12	0	10	2
setCurrentMinute(Integer)	10	0	6	4
setTextAppearance(...)	12	0	1	11
release()	14	0	0	14
getDeviceId()	12	0	1	11
saveLayer(...)	15	0	0	15
requestAudioFocus(...)	0	8	0	8
Total	**96**	**16**	**20**	**92**

Compared to AppEvolve, CocciEvolve has several advantages:

- CocciEvolve does not need extensive setup or configuration;
- CocciEvolve is capable of updating multiple API calls in the same file without additional configuration;
- CocciEvolve provides an easily readable and understandable semantic patch as a by-product;
- CocciEvolve only needs a single updated example.

5 RELATED WORK

There are many studies on API deprecation [2, 7, 9, 11, 12, 17, 18, 20–23]. Kapur et al. [7] discovered that APIs can be removed from a library without warning. Zhou and Walker [23] proposed a tool to mark deprecated API usages in StackOverflow posts. Some studies propose approaches [9, 11, 18, 20–22] to detect API compatibility issues. Brito et al. [2] showed that not all APIs are annotated with replacement messages. Sawant et al. [17] found 12 reasons for deprecation. Li et al. [12] characterized Android APIs and found inconsistencies between their annotation and documentation. Unlike these studies, our work aims to automatically update usages of deprecated Android APIs.

There are many studies on program transformations inference [3, 5, 13, 15, 16]. LASE [13] creates edit scripts by finding common changes from a set of change examples. REFAZER [15] employs a programming-by-example methodology to infer transformations from a set of change examples. REVISAR [16] finds common Java edit patterns from code repositories by clustering the edit patterns. Jiang et al. [5] proposed GenPat, which builds source code hypergraphs to infer transformations. Fazzini et al. [3] proposed AppEvolve to transform app with Android deprecated-API usages into ones that are backward compatible. Our work shares the same goal. However, while AppEvolve learns from a before- and after- update example, our approach requires only one after-update example.

6 CONCLUSION AND FUTURE WORK

In this work, we propose CocciEvolve, which can learn an Android API update from a single after-update example. CocciEvolve performs code normalization to standardize the code and writes the update in the Semantic Patch Language (SmPL) to make the transformation transparent and readable to developers. Our experiments on 112 target files on 10 deprecated Android APIs show that CocciEvolve successfully updates usages in 96 target files. On the other

hand, AppEvolve can only update deprecated-API usages in 20 of these target files. For future work, we plan to perform code slicing to find code relevant to the API update, including code beyond method boundaries. We also plan to perform code denormalization to restore the original coding style.

Acknowledgement. This research is supported by the Singapore NRF (award number: NRF2016-NRF-ANR003) and the ANR ITrans project.

REFERENCES

[1] Muhammad Hilmi Asyrofi, Ferdian Thung, David Lo, and Lingxiao Jiang. 2020. AUSearch: Accurate API Usage Search in GitHub Repositories with Type Resolution. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*.

[2] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. 2016. Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems. In *SANER*, Vol. 1. IEEE, 360–369.

[3] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *ISSTA*. ACM, 204–215.

[4] 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 *ASE*. ACM, 167–177.

[5] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 255–266.

[6] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. 2019. Semantic Patches for Java Program Transformation (Experience Report). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[7] Puneet Kapur, Brad Cossette, and Robert J. Walker. 2010. Refactoring References for Library Migration. In *OOPSLA*. ACM, 726–738.

[8] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX Annual Technical Conference*. 601–614.

[9] Cong Li, Chang Xu, Lili Wei, Jue Wang, Jun Ma, and Jian Lu. 2018. ELEGANT: Towards Effective Location of Fragmentation-Induced Compatibility Issues for Android Apps. 278–287. https://doi.org/10.1109/APSEC.2018.00042

[10] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of API-related compatibility issues in Android apps. In *ISSTA*. ACM, 153–163.

[11] Li Li, Tegawendé Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: automating the detection of API-related compatibility issues in Android apps. 153–163. https://doi.org/10.1145/3213846.3213857

[12] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising deprecated Android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*. ACM, 254–264.

[13] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *ICSE*. IEEE Press, 502–511. http://dl.acm.org/citation.cfm?id=2486788.2486855

[14] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *European Conference on Computer Systems (EuroSys)*. ACM, 247–260.

[15] 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 *ICSE*. IEEE Press, 404–415. https://doi.org/10.1109/ICSE.2017.44

[16] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D’Antoni. 2018. Learning Quick Fixes from Code Repositories. *CoRR* abs/1803.03806 (2018). arXiv:1803.03806 http://arxiv.org/abs/1803.03806

[17] Anand Ashok Sawant, Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski, and Alberto Bacchelli. 2018. Why are features deprecated? An investigation into the motivation behind deprecation. In *ICSME*. IEEE, 13–24.

[18] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. 2019. Data-Driven Solutions to Detect API Compatibility Issues in Android: An Empirical Study. 288–298. https://doi.org/10.1109/MSR.2019.00055

[19] Ferdian Thung, Stefanus A. Haryono, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. 2020. Automated Deprecated-API Usage Update for Android Apps: How Far Are We?. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*.

[20] Lili Wei, Yepang Liu, S.C. Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. 2018. Understanding and Detecting Fragmentation-Induced Compatibility Issues for Android Apps. *IEEE Transactions on Software Engineering* PP (10 2018), 1–1. https://doi.org/10.1109/TSE.2018.2876439

[21] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: characterizing and detecting compatibility issues for Android apps. 226–237. https://doi.org/10.1145/2970276.2970312

[22] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2019. PIVOT: Learning API-Device Correlations to Facilitate Android Compatibility Issue Detection. 878–888. https://doi.org/10.1109/ICSE.2019.00094

[23] Jing Zhou and Robert J Walker. 2016. API deprecation: a retrospective analysis and detection method for code examples on the web. In *ICSE*. ACM, 266–277.