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
{stefanush,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.

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.

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
• 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 –; (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

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

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

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

(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.

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,
3.3 API Update Semantic Patch Creation

Using the normalized update block as an input, this component creates a CocciEvolve 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 CocciEvolve. 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.

```java
@bottomupper_assignment0
expression exp0;
identifier classIden;
@@
TimePicker classIden = exp0;
@@

... if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
  + classIden.getCurrentMinute();
+ } else {
  + 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

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

5 RELATED WORK

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 1: Number of targets in our evaluation dataset

<table>
<thead>
<tr>
<th>API Description</th>
<th># Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>addAction(...)</td>
<td>0</td>
</tr>
<tr>
<td>getAllNetworkInfo()</td>
<td>8</td>
</tr>
<tr>
<td>getCurrentHour()</td>
<td>9</td>
</tr>
<tr>
<td>setCurrentHour()</td>
<td>12</td>
</tr>
<tr>
<td>setCurrentHour(Integer)</td>
<td>12</td>
</tr>
<tr>
<td>setCurrentMinute()</td>
<td>10</td>
</tr>
<tr>
<td>setTextAppearance(...)</td>
<td>12</td>
</tr>
<tr>
<td>addGpsStatusListener(...)</td>
<td>0</td>
</tr>
<tr>
<td>release()</td>
<td>14</td>
</tr>
<tr>
<td>removeGpsStatusListener(...)</td>
<td>0</td>
</tr>
<tr>
<td>shouldOverrideUrlLoading(...)</td>
<td>0</td>
</tr>
<tr>
<td>startDrag(...)</td>
<td>0</td>
</tr>
<tr>
<td>abandonAudioFocus(...)</td>
<td>0</td>
</tr>
<tr>
<td>getDeviceId()</td>
<td>12</td>
</tr>
<tr>
<td>requestAudioFocus(...)</td>
<td>8</td>
</tr>
<tr>
<td>saveLayer(...)</td>
<td>15</td>
</tr>
<tr>
<td>setAudioStreamType(...)</td>
<td>0</td>
</tr>
<tr>
<td>vibrate(long)</td>
<td>0</td>
</tr>
<tr>
<td>vibrate(long[], int)</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 2: Statistics of updating target files per API

<table>
<thead>
<tr>
<th>API</th>
<th>CocciEvolve</th>
<th>AppEvolve</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Success</td>
<td>Fail</td>
</tr>
<tr>
<td>getAllNetworkInfo()</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>getCurrentHour()</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>setCurrentMinute()</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>setCurrentHour(Integer)</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>setCurrentMinute(Integer)</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>setTextAppearance(...)</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>release()</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>getDeviceId()</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>saveLayer(...)</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>requestAudioFocus(...)</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

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.

### 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.

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

REFERENCES


