Towards Generating Transformation Rules without Examples for Android API Replacement

Ferdian Thung, Hong Jin Kang, Lingxiao Jiang, and David Lo
Singapore Management University
Email: {ferdianthung, hjkang.2018, lxjiang, davidlo}@smu.edu.sg

***Abstract*—Deprecation of APIs in software libraries is common when library maintainers make changes to a library and will no longer support certain APIs in the future. When deprecation occurs, developers whose programs depend on the APIs need to replace the usages of the deprecated APIs sooner or later. Often times, software documentation specifies which new APIs the developers should use in place of a deprecated API. However, replacing the usages of a deprecated API remains a challenge since developers may not know exactly how to use the new APIs. The developers would first need to understand the API changes before they can replace the deprecated API correctly. Previous work has proposed an approach to assist developers in deprecated Android API replacement by learning from examples. In this work, we also focus on Android APIs and propose an approach named No Example API Transformation (NEAT) to generate transformation rules that can assist developers in deprecated API replacement even when code examples are not available (e.g., when the deprecation has happened recently). We have validated the effectiveness of NEAT on generating transformation rules for deprecated Android APIs. Using NEAT, we can generate 37 transformation rules for 37 out of a selection of 100 deprecated APIs and have validated these rules to be correct.**

***Index Terms*—API deprecation, API replacement, program transformation, Android**

I. INTRODUCTION

Software maintenance is a daily activity for software developers. One of the tasks that developers need to do while maintaining a software project is to make sure that the APIs that they use are up-to-date. However, API updates do not always guarantee backward compatibility and the maintainers can decide to deprecate older APIs. Thus, when APIs are deprecated, developers need to replace the deprecated APIs with other APIs that can provide equivalent functionality.

Many works [2], [10], [16], [17] that support deprecated API replacement only focus on identifying APIs that can replace a deprecated API. There has been very limited work on recommending *how* a deprecated API can actually be replaced by its replacement APIs. Lamonthe et al. [5] has proposed an approach to assist deprecated API replacement by learning from code change examples that have replaced deprecated APIs. Their approach learns API migration patterns from these examples and applies these patterns to Android applications to replace deprecated API invocations in the applications.

In this work, we also deal with the problem of assisting developers in replacing usages of deprecated APIs. In contrast to the work of Lamonthe et al., we propose No Example API Transformation (NEAT) that can provide assistance at the time when an API deprecation occurs and no code change example is yet available from which to learn code changes patterns. Thus, NEAT complements Lamonthe et al.'s approach that requires code change examples. Moreover, different from Lamonthe et al.'s approach, NEAT produces code transformation rules based on semantic patches [3], [6], [14] that can be easily reused, making it easier to replace deprecated APIs across many usage locations.

NEAT assumes that the API replacement mappings from a deprecated API to its replacement API are available, as such mappings can be generated by previous approaches that perform API replacement mining [2], [10], [16], [17]. The mined mappings specify that a method A should be replaced with a method B, but they do not specify how to actually perform this replacement. Thus, NEAT also complements those approaches by providing code transformation rules that specify how deprecated API replacement can actually be realized.

NEAT uses a combination of API source code analysis, type conversion, and heuristics. The approach distinguishes between two cases, depending on whether or not the affected API is an event handler (i.e., APIs that are called when an event occur). If both APIs are non event handlers, NEAT analyzes the source code of the API to check whether the implementation of the deprecated API involves the usage of the replacement API. If such an implementation exists, NEAT generates a transformation rule based on it. Otherwise, we heuristically match the parameters of both APIs, and convert deprecated API parameters and/or return values to the types used by the replacement API (if necessary). On the other hand, if both APIs are event handlers, NEAT uses similar heuristics to generate a transformation rule to implement the replacement method based on the deprecated method's implementation. To make the generated transformation rules easily reusable, we express them in semantic patches, which is widely adopted and has shown to be useful in the context of C. Semantic patches can be applied to client applications using Coccinelle [6], [14]. Specifically, since Android is typically written in Java, we use Coccinelle4J, a recent port of Coccinelle targeting Java programming language [3].

We evaluated NEAT on a dataset of 100 deprecated Android APIs with identified replacements as input. We manually validated whether the generated transformation rules can be used to replace the deprecated APIs correctly. We found that NEAT can generate correct transformation rules for 37 deprecated APIs. Comparing the effectiveness of NEAT to that

of automated program repair [4], [7], [12], which also needs to transform a piece of code from a bad state to a good one and correctly repair programs with the effectiveness of 22.69-53.33%, our current preliminary result is promising. Note that different from automated program repair, we do not have any test cases and/or failures to guide our transformations.

The main contributions of our work are as follows:

1) To the best of our knowledge, we are the first to generate transformation rules for replacing deprecated APIs *when there are no available examples* to learn from.
2) We propose an approach named No Example API Transformation (NEAT) to generate transformation rules given a deprecated method and its replacement through a combination of API source code analysis, type conversion, and some heuristics.
3) Evaluation on 100 deprecated Android APIs with identified replacements shows that NEAT generates correct replacement rules for 37 deprecated Android APIs.

II. Background

A. Coccinelle

Coccinelle is a program matching and transformation engine [6], [11], [14]. It was initially developed to automate API evolutions in the Linux kernel. Coccinelle accepts as input a program to transform, and a *semantic patch* that describes the transformation to be performed. The semantic patch generalizes the standard `diff` format. This is intended to make it easy for developers to write and understand tranformation rules, since the format should be familiar to developers. It then transforms the input program according to the semantic patch if the program matches it. Transformations are performed at all matching locations. Coccinelle analyzes the program's abstract syntax trees and intraprocedural control flow graphs to perform transformation.

Like Coccinelle, our work also aims to automate API evolutions, specifically for migrating deprecated API usages in client applications. To benefit from the fact that Coccinelle semantic patches are readable and reusable, we use Coccinelle in our work. Specifically, as we target Java code, we generate semantic patches for input to Coccinelle4J [3], which is a recent port of Coccinelle to the Java programming language.

B. Semantic Patch Language (SmPL)

Coccinelle semantic patches are written in the Semantic Patch Language (SmPL). A semantic patch describes the code that should be removed as lines starting with $-$, the code that should be added as lines starting with $+$, and any necessary surrounding code that defines the context of the code addition and removal.

A semantic patch written in SmPL contains two parts: (1) metavariable declarations; (2) context and change operations that are described using the declared metavariables. A metavariable is a symbol that represents an element in the grammar of the target programming language. As illustrated in Figure 1, the lines surrounded by `@@` declare the metavariables. In Figure 1, `arg` is declared to be a metavariable that can match any expression. The remaining lines are the context and change operations. In this case, the patch specifies that all calls to `setBackgroundDrawable` should be replaced with a call to `setBackground`, while keeping the argument of the method intact.

```
@@
expression view, arg;
@@
- view.setBackgroundDrawable(arg);
+ view.setBackground(arg);
```

Fig. 1: An example of a simplified semantic patch replacing a deprecated method from `android.view.View` class

SmPL supports more features for writing semantic patches. Further details about SmPL are available in previous work [1], [6], [15].

III. Approach

NEAT focuses on *one-to-one mapping* (i.e., a mapping from one deprecated API to one replacement API), and takes as input a deprecated API signature (i.e., the method name, the class name, and the parameters), its replacement API signature, and the source code of the library containing the two APIs. The output of NEAT is a semantic patch written in SmPL, that specifies the transformation rule to replace a usage of the deprecated API.

We observe that a deprecated API can either be designed to be invoked from a client application, which we refer to as *use by invocation*, or be designed to be implemented by a client application as an event handler, which we refer to as *use by implementation*. Different API migration strategies are required in each case. To distinguish between these cases, we rely on the API name; if the API name starts with `on` (e.g., `onMetadataChanged`, and `onAudioStateChanged`), we consider it to be an event handler. We consider API migration for each of these cases below.

A. Migrating Deprecated API Invocations

We observe that the implementation of a deprecated API may contain the usage of the replacement API and this implementation can be used to replace the deprecated API. Based on this observation, NEAT first checks whether the implementation of the deprecated API uses the replacement API. NEAT uses different strategies to generate semantic patches depending on the outcome of this check:

1) The replacement API is invoked in the body of the deprecated API: In this case, we follow the body of the deprecated API to construct our semantic patch. Our first task is to adapt the code of the deprecated API body to work in the context of client applications. NEAT adapts references to `public` methods and fields so that they can be accessed correctly from the client application. It does not adapt `private` or `protected` fields and methods since they are inaccessible from outside the class that owns them. In such case, NEAT is unable to generate a semantic patch following the body of the deprecated API. In that case, it falls back on the procedure described below in Section III-A2.

Semantic Patch Generation. After performing adaptations, NEAT constructs a semantic patch as follows. If the body of the deprecated API does not contain the use of `private` or `protected` fields or methods, NEAT can fully adapt the usage of replacement API for client applications and proceeds to construct a semantic patch with two rules. The first rule finds a method that invokes the deprecated API. The second rule replaces the invocation of the deprecated API with the invocation of a new private method inside the client application. This private method contains a usage of the replacement API that has been adapted for client applications.

2) The replacement API is not invoked in the body of the deprecated API: In this case, NEAT considers the problem of replacing the deprecated API as a problem of mapping and converting the parameters and return value of the deprecated API to those of its replacement API

Parameter mapping. NEAT first matches the parameters of the deprecated API to the parameters of the replacement API. It works in two stages. In the first stage, a parameter of the deprecated API is considered to match with a parameter of the replacement API if both have the same name and type. In the second stage, if there are still unmatched parameters in both the deprecated API and the replacement API, NEAT considers two parameters to match if both have the same type. If multiple parameters have the same type, NEAT considers two parameters to match if they appear in the same position when scanning the remaining parameters from left to right in the API parameter lists.

At the end of the above two stages, we might still have unmatched parameters, which fall into the following cases.

- *Unmatched parameters in the deprecated API.* In this case, we assume that the parameters are no longer necessary. Therefore, we do not map these parameters.
- *Unmatched parameters in the replacement API.* In this case, we assign default values for the parameters according to their types. We assign `null` for `Object`, `""` for `String`, 0 for numbers (e.g., `int`), and `false` for `boolean`.
- *Unmatched parameters in both APIs.* In this case, if there is exactly one unmatched parameter for both APIs, we consider that the unmatched parameter in the deprecated API matches with the unmatched parameter in the replacement API. NEAT currently does not handle the case in which there are multiple unmatched parameters.

If all parameters can be matched, matched parameters of different types are marked for type conversion. This means that a matched parameter in the deprecated API needs to be converted from its original type to the type of its matched parameter in the replacement API.

Type Conversion. Like parameters, return types of the deprecated API and the replacement API may require conversion. If the return types of the deprecated API and the replacement API are different, NEAT marks the return type of the replacement API for conversion to the return type of the deprecated API. To perform type conversion for all types that have been marked for it, NEAT generates a signature graph from the source code of the API library, which is inspired by the work of Mandelin et al. on jungloid mining [9]. This signature graph is a directed graph that can be traversed to convert one type to another. The nodes in the graph are the set of types found in the API source code. Each directed edge in the graph indicates an operation to convert one type to another. Possible operations include field access, method invocation, or widening. Given a type requiring conversion, NEAT finds the shortest path from the source type to the destination type. In case of multiple shortest paths, NEAT selects the path that makes the most use of members of the current type.

Semantic Patch Generation. At this point, NEAT knows how to replace a given deprecated API invocation with its replacement API invocation. To create a semantic patch to make the change, NEAT considers each parameter and the return type of the deprecated API as expressions in SmPL. As we want to replace the deprecated API invocation, NEAT creates a semantic patch that specifies the deletion of a deprecated API invocation. It also specifies the addition of the replacement API invocation and the code to convert the required types. Furthermore, it adds import statements for all types that are not found in the deleted lines. These may include types of parameters of the replacement API, the return type of the replacement API, and all types that are used in type conversion. Also, if the deprecated API and the replacement API belong to different classes, we assume the object of the replacement class does not exist yet and create it using a default constructor. The invocation of the replacement API would also be adjusted to use the newly created object.

B. Migrating Deprecated API Implementation

When migrating a deprecated API implementation for an event handler, NEAT follows a strategy similar to the one used for migrating a deprecated API invocation in the case that no usage of the replacement API is found inside the body of the deprecated API. However, the type conversion direction for method parameters is the opposite of the one used for deprecated API invocation. This is to ensure that the implementation in the body of the deprecated API can be reused in the body of the replacement API. More specifically, for a deprecated API implementation, NEAT converts a parameter of the replacement API from its type to the type of the matching parameter in the deprecated API, and converts the return value of deprecated API from its type to the return type of the replacement API.

At this point, NEAT knows how to replace a given deprecated API implementation with its replacement API implementation. If the deprecated API does not return anything, the patch replaces the declaration of the deprecated API with a declaration of the replacement API. It then adds a piece of code that maps and converts parameters of the replacement API to the parameters of the deprecated API inside the body of the replacement API. Finally, it appends the implementation of the deprecated API.

For cases in which the deprecated API returns something, NEAT treats the parameters of the deprecated API as identifiers in SmPL and the return type of deprecated API as an expres-

sion in SmPL. The generated patch consists of two rules. The first rule finds the implementation of the deprecated API and records the body of the deprecated API. The second rule puts the body of the deprecated API into a new private method with the same name as the deprecated API as added lines in the patch. It also generates a declaration for the deprecated API and its body as the deleted lines in the patch, and generates a declaration for the replacement API, a call to the new private method, and the code to convert the required types as the added lines in the patch.

IV. Empirical Evaluation

A. Dataset

For our dataset, we consider the released Android versions listed in Table I. These releases cover API levels 1-28, except API levels 1, 2, 3, 11, and 12. We assume that different releases with the same API level would have similar content, particularly in terms of their APIs. We exclude API levels 1, 2, 3, 11, and 12 since there are no release versions for them. We also exclude API level 20 since it is reserved only for wearable devices.

TABLE I: Investigated Android Versions

Release Versions
android-9.0.0_r9, android-8.1.0_r48, android-8.0.0, android-7.1.2_r9, android-7.0.0_r7, android-6.0.1_r9, android-5.1.1_r9, android-5.0.2_r3, android-4.4w_r1, android-4.3_r3.1, android-4.2_r1, android-4.1.2_r2.1, android-4.0.4_r2.1, android-4.0.2_r1, android-3.2.4_r1, android-2.3.7_r1, android-2.3.2_r1, android-2.2.3_r2.1, android-2.1_r2.1s, android-2.0.1_r1, android-2.0_r1, android-1.6_r2

We check out the Android release versions listed in Table I. We analyze all *.java files in each version. For each file, we collect public methods that contain the *@deprecated* Javadoc tag or the *@Deprecated* annotation. *@deprecated* is used to document that the method is deprecated and provide comments describing the reason a method is deprecated and what developers should do to deal with the deprecation. *@Deprecated* lets compilers know about the deprecation, in which case the compiler issues a warning when the deprecated method is used.

Given a set of methods that are deprecated for each version, we combine them into a single set of unique deprecated methods across the investigated Android versions. Within this set, we identify the subset that explicitly points to the methods in Android APIs that we should use in place of the deprecated methods. To identify this subset, we collect the methods for which the deprecation message contains method references, which we consider as text of the form of `methodName(arg1, arg2, ...)`. We obtain a set of 843 deprecated methods and their potential replacements.

However, not all of collected deprecated methods actually contain deprecation messages that explicitly point to the replacement methods. We manually check the source code of each deprecated method and drop the ones where the method mentioned in the deprecation message is not the correct replacement. Incorrect cases usually include a message informing developers that the method should not be used at all or only for testing purposes. We are interested only in an explicit suggestion to ensure the correctness of the replacement methods. As a preliminary evaluation, we collect the first 100 of deprecated methods with *one-to-one mapping* to their replacement methods as dataset for our experiments.

B. Research Questions

RQ1. *What are the differences between deprecated APIs and their replacement APIs?*

Identifying the differences between deprecated APIs and their replacements informs us about the kinds of replacements. We investigate the differences in class names, method names, number of parameters, types of parameters, and return type.

RQ2. *To what extent does* NEAT *generate semantic patches that are correct?*

We count the number of generated semantic patches that are correct. We consider a semantic patch to be correct if it can correctly and completely replace a deprecated API usage.

C. Results

1) RQ1. Characteristics of Replacement: We study the *one-to-one mapping* replacement rules in our dataset. We find that they involve any of the following structural differences or a combination of them:

- *Class difference.* This involves replacement of a method with one in another class. This can be the only difference when a class is deprecated and replaced with a new class.
- *Method name difference.* This involves replacement of a method with one having a different name. This may occur when the original name had a typo, or does not well represent the method functionality.
- *Deletion of parameter(s).* This involves deletion of one or more method parameters. This may occur when the parameter is no longer needed.
- *Addition of parameter(s).* This involves addition of one or more method parameters. This may occur if several methods having similar functionality are combined into a single method. In this case, a flag may be added as a new parameter to indicate which of the methods should be used.
- *Return type difference.* This involves replacement using another method having a different return type. This may occur if the return value of the replacement method contains the return value of the deprecated method. As such, the deprecated method is redundant.

2) RQ2. Effectiveness of Semantic Patch Generation: Given the generated patches, we manually inspected them to check whether they are correct. To make this assessment, we read the documentation and the source code of both the deprecated and replacement methods. We verified our assessment with two PhD students who are not authors of this paper and have more than five years of general development experience. We found that NEAT successfully generates 37 correct semantic patches out of the 100 deprecated APIs in our dataset, without the need of any examples.[1]

In some cases, NEAT may not generate a patch for a deprecated API due to the following reasons:

[1]The 37 correct semantic patches generated by NEAT are available at https://github.com/fthung/NEAT

- *Unhandled cases.* NEAT does not handle several cases such as when a deprecated event handler method is replaced by a non-event handler method (or vice versa).
- *Type conversion failure.* For both parameters and return type, if NEAT does not find a path to convert one type to another, it does not generate a patch.

V. Related Work

Wu et al. [16] developed a tool named AURA to mine API replacement rules using method call dependency analysis and text similarity. Meng et al. [10] proposed HiMA, that infers API replacement rules between two versions by aggregating rules from each pair of consecutive revisions in the framework history between the two versions. Yu et al. [17] develop an approach named AUC-Miner, which mines API replacement rules by employing context information to refine method call dependency analysis. Li et al. [8] performed an exploratory study on characterizing Android APIs. They have found that, among other things, deprecated Android APIs are not always consistently annotated and documented, and they are also regularly removed. They have also developed a prototype tool that can generate API replacement mappings from the Android framework source code. In this work, given the API replacement mappings, our approach generates transformation rules, in the form of semantic patches, that can be used to perform the API replacement. Thus, our approach complements approaches that mine API replacement mappings.

Nita and Notkin [13] proposed twinning (i.e., a technique to manually specify program changes without modifying the program directly) to migrate program to alternative APIs. Zhong et al. [18] developed the tool MAM, which mines API mappings for language migration from programs with two versions in two programming languages. Lamonthe et al. [5] proposed an approach that learns API migration patterns from code examples and can apply these patterns to migrate deprecated APIs. Lamonthe et al.'s work is the closest to ours. Our work also deals with API migration, particularly replacing deprecated APIs. However, different than their work, our work focuses on the scenario in which there are no examples.

VI. Conclusion and Future Work

We propose NEAT, an approach that can automatically generate transformation rules to assist replacements of deprecated Android APIs, as soon as the APIs are labeled deprecated. NEAT complements the approach by Lamonthe et al. [5] that assists in replacing deprecated Android APIs by learning from examples as it can assist developers when examples are not available. We have tested NEAT on a dataset of 100 deprecated Android APIs with identified replacements. Our preliminary experiments show that NEAT can produce correct transformation rules for 37 deprecated APIs, which can help developers in migrating the usages of these APIs. In the future, we plan to apply NEAT on more deprecated APIs in Android framework and other deprecated APIs beyond those that appear in Android framework. We also plan to add more heuristics and analysis to cover cases that are not currently handled by our proposed approach. It will also be interesting to combine our approach that analyzes library code with an example-based approach (that analyzes client code) to boost the overall effectiveness.

Acknowledgement

This research was supported by the Singapore National Research Foundation (award number: NRF2016-NRF-ANR003).

References

[1] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *Principles of Programming Languages (POPL)*, pages 114–126. ACM, 2009.

[2] Barthélémy Dagenais and Martin P Robillard. Recommending adaptive changes for framework evolution. *TOSEM*, 20(4):19, 2011.

[3] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. Automating program transformation for Java using semantic patches. In *European Conference on Object-Oriented Programming (ECOOP)*, 2019.

[4] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE Press, 2013.

[5] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Chen. A4: Automatically assisting Android API migrations using code examples. *arXiv preprint arXiv:1812.04894*, 2018.

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

[7] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

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

[9] David Mandelin, Lin Xu, Rastislav Bodık, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Programming language design and implementation (POPL)*. ACM, 2005.

[10] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *ICSE*, pages 353–363. IEEE Press, 2012.

[11] Gilles Muller, Yoann Padioleau, Julia L Lawall, and René Rydhof Hansen. Semantic patches considered helpful. *ACM SIGOPS Operating Systems Review*, 40(3):90–92, 2006.

[12] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[13] Marius Nita and David Notkin. Using twinning to adapt programs to alternative APIs. In *International Conference on Software Engineering (ICSE)*, volume 1, pages 205–214. IEEE, 2010.

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

[15] Yoann Padioleau, Julia L Lawall, and Gilles Muller. Smpl: A domain-specific language for specifying collateral evolutions in linux device drivers. *Electronic Notes in Theoretical Computer Science*, 166:47–62, 2007.

[16] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: a hybrid approach to identify framework evolution. In *ICSE*, volume 1, pages 325–334. IEEE, 2010.

[17] Ping Yu, Fei Yang, Chun Cao, Hao Hu, and Xiaoxing Ma. API usage change rules mining based on fine-grained call dependency analysis. In *Internetware*, page 10. ACM, 2017.

[18] Jing Zhou and Robert J Walker. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ICSE)*, pages 266–277. ACM, 2016.