A4: Automatically Assisting Android API Migrations Using Code Examples

Maxime Lamothe, Weiyi Shang, Tse-Hsun Chen
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
(max_lam, shang, peterc)@encs.concordia.ca

Abstract—The fast-paced evolution of Android APIs has posed a challenging task for Android app developers. To leverage the newly and frequently released APIs from Android, developers often must spend considerable effort on API migrations. Prior research work and Android official documentation typically provide enough information to guide developers in identifying both the changed API calls that need to be migrated and the corresponding API calls in the new version of Android (what to migrate). However, the API migration task is still challenging since developers lack the knowledge of how to migrate the API calls. In addition to the official documentation, there exist code examples, such as Google Samples, that illustrate the usage of APIs. We posit that by analyzing the changes of API usage in code examples, we may be able to learn API migration patterns to assist developers with API migrations.

In this paper, we propose an approach that automatically learns API migration patterns from code examples and applies these patterns to the source code of Android apps for API migration. To evaluate our approach, we migrate API calls in open source Android apps by learning API migration patterns from both public and manually generated code examples. We find that our approach can successfully learn API migration patterns and provide API migration assistance in 71 out of 80 cases. In particular, our approach can either automatically migrate API calls with little to no extra modifications needed or provide guidance to assist with the migrations. Our approach can be adopted by Android developers to reduce the effort they spend on regularly migrating Android APIs.

I. INTRODUCTION

Software maintenance is one of the most expensive activities in the software development process [1]. To reduce the maintenance cost, developers often rely on reusing available software. The reused software helps abstract the underlying implementation details and can be integrated into innumerable software projects. In particular, calling application programming interfaces (APIs) is a common software reuse technique used by developers [2], [3]. These APIs provide well-defined programming interfaces that allow their users to obtain desired functionality without forfeiting development time.

However, in today’s fast-paced development, APIs are evolving frequently. One of the fastest-evolving and widely used API is the Android API [4], [5]. Prior studies [5], [6] found that Android is evolving at an average rate of 115 API updates per month. Such evolution may entail arbitrary release schedules and API deprecation durations and may involve removing functionality without prior warning [7]. Therefore, users must regularly study the changes to existing APIs and decide whether they need to migrate their code to adopt the changes. In fact, a prior study shows that developers are slower at migrating API calls than the API evolution speed itself [5]. As a consequence, there is fragmentation in the user base and slow adopters miss out on new features and fixes [5].

Existing migration recommendation techniques [8]–[11] typically focus on identifying what is the replacement of a deprecated API (e.g., one should now be using methodB instead of methodA), instead of how to migrate the API calls for the replacement (e.g., how to change the existing code to call methodB). However, a recent experience report shows that all too often, Android API official documentation clearly states what to replace for a deprecated API, while actually performing API migrations is still challenging and error prone [6].

There exist many publicly-available code examples online illustrating API usages. As an example, Google provides a set of sample Android projects on the Google Samples repository [12]. Developers often study these sample projects and other code examples (e.g., code from open source Android apps) to help them with API migration [2], [13]–[15]. Nevertheless, studying the code examples to know the changes needed for API migrations is a manual and time-consuming process. Furthermore, identifying where and how to apply migration changes puts an extra burden on developers during software maintenance.

In this paper, we propose an approach, named A4, that leverages source code examples to assist developers with API migration. We focus our study on Android API migrations, due to Android’s wide adoption and fast evolution [5]. Our approach automatically learns the API migration patterns from code examples. Afterwards, our approach matches the learned API migration pattern to the source code of the Android apps to identify API migration candidates. If migration candidates are identified, we automatically apply the learned migration pattern to the source code of Android apps.

We evaluate our approach by applying Android API migration on open source Android apps from FDroid, and leveraging their test suites. We learn Android API migration patterns from three sources of code examples: 1) official Android code examples provided by Google Samples [12], 2) migration patterns that are learned from the development history of open source Android projects i.e., FDroid [16] and 3) API migration examples that are manually produced by users. In particular, we answer three research questions.

RQ1 Can we identify API migration patterns from public
Our approach can automatically identify 83 API migration patterns out of 125 distinct Android APIs that require migrations from both Google Samples and the development history of open source Android projects.

**RQ2** How much effort is required to produce code examples?

Through our user study, we found that the task of manually producing API migration examples is not challenging. On average, over 80% of the time, the code examples for migration can be produced faultlessly in around three minutes. The results show that we can use these examples as inputs to our approach to automatically migrate API calls.

**RQ3** How much assistance can our approach provide when migrating APIs?

Based on 80 migrations candidates in 32 open source apps, our approach can generate 14 faultless migrations, 21 migrations with minor code changes, and 36 migrations with useful guidance to developers.

Our approach can be adopted by Android app developers to reduce their API migration efforts to cope with the fast evolution of Android APIs. Although created for the Android API, A4 uses the Java programming language and could be easily modified to work with other APIs. Our approach also exposes the value of learning the knowledge that resides in rich code examples to assist in the various tasks of API related software maintenance.

**Paper Organization.** Section II provides a real-life example of an API migration to motivate this study. Section III describes our automated approach, A4, that assists in Android API migration. Section IV presents the design of the experiments used to evaluate our approach and Section V presents the results of our experiments. Section VI provides a short survey of related work. Section VII describes threats to the validity of this study. Finally, Section VIII concludes the paper.

**II. A Motivating Example**

In this section, we present an example, which motivates our approach based on learning migration patterns from code examples to assist in API migration.

In Android API version 23, the `Resources.getColor` API (as shown in Listing 1) was deprecated and replaced (as shown in Listing 2). In fact, the deprecation and replacement (what to migrate) are clearly shown in the official Android documentation [17]. However, even with the help from the documentation, the addition of a new parameter provides new challenges to the Android app developers. In particular, developers may not have the knowledge necessary to retrieve the `Theme` information nor to initialize a new object for the `Theme` to make the new API call. Moreover, since the old API call does not require any `Theme`, even if developers can provide a `Theme`, there is no information on how to preserve backward compatibility.

On the other hand, there exists open source example projects on Google Samples [12], i.e., `androidtv-Leanback`, a project on Github which presents several uses of the `Resources.getColor` method. With the introduction of a new Android API version, these code examples are also updated. By looking at an example, we find that it clearly demonstrates how to call the `Resources.getColor` API (see Figure 1). From the changes to the code example, we can see that to maintain backward compatibility, developers can simply pass a `null` object to the API. Without such an example, figuring out that a null value preserves backward compatibility would require trial and error from developers. By learning such a migration pattern in the code examples, the effort of the challenging task of how to migrate an API call can be reduced for developers.

### Listing 1. Resources.getColor API before migration

```java
public class Resources extends Object {
    public int getColor(int id, Resources.Theme theme)
}
```

### Listing 2. Resources.getColor API after migration

```java
public class Resources extends Object {
    public int getColor(int id, Resources.Theme theme)
}
```

However, finding these migrations on Google Samples is a laborious endeavor. First of all, the code examples do not index their usage of APIs. Developers may need to search for the API usage of interest from a large amount of source code. Second, even with the API usage in the code examples, developers need to go through the code history to understand the API migration pattern, i.e., how to apply these changes, which can be much more complex than the aforementioned example. Finally, even with the migration pattern, developers need to learn how to apply the migration on their own source code, which can be a challenging task [6].

Taking the aforementioned API `Resources.getColor` as an example, we can detect a total of 1,626 places where the `Resources.getColor` Android API is called in its deprecated form on a sample of 1,860 open source Android apps from FDoRid [16]. Migrating `Resources.getColor` to Android API version 23 or later for all of those apps requires a significant amount of effort. Automating the above-described migration approach and providing the information of the learned pattern to developers can significantly reduce the required migration effort.
In the next section, we present our approach, named A4, which automatically provides assistance in Android API migration.

III. APPROACH

In this section, we present an automated approach, A4, that assists in API migration by learning how to migrate API calls from code examples. Our approach consists of two steps: 1) identifying API migration patterns from code examples and 2) applying the migration pattern that is learned from the example to the source code of the Android apps. An overview of our approach is shown in Figure 2, and an implementation of our approach is publicly available online.

A. Learning API migration patterns from code examples

In the first step of our approach, we mine readily available code examples to learn API migration patterns. Such code examples can be found through online code repositories such as GitHub and FDroid [16]. Our approach also supports self-made examples, which allows users to produce their own migrated stubs, or use their own projects as data to feed forward into other projects.

1) Extracting APIs: We extract all the Android APIs for every Android version that is available from the official Android API documentation. However, the Android documentation may have discrepancies to the APIs in their source code. Therefore, for every version of Android, we also obtain the Android source code and parse the source code using Eclipse JDT AST parser [13]. We identify the lists of all public methods as available APIs.

2) Searching API calls from code examples: In this step, we identify the API calls that are available in the code examples. A pseudocode algorithm of this step is presented in Algorithm 1. One may design an approach that builds AST for all available code examples as well as the targeted Android app source code. Afterwards, one may use the ASTs to match the API calls in the source code to the code examples. However, such an approach would be time consuming due to the fact that 1) building ASTs is a time-consuming endeavor, 2) the complexity of AST matching is non-negligible and 3) the number of Android APIs that are available is large.

In order to reduce the time needed to identify API calls, we first use basic lexical matching to limit the scope of needed AST building and matching. In particular, we first search all the files in the code example for strings that match API names. These files are then selected for further processing as potential matches. Although the basic lexical matching can lead to false positives, the goal is merely scoping down the search space and the following AST matching can remove these false positives.

Afterwards, we apply AST matching on the files with potential matches. We leverage JDT [18] to parse the source code

Algorithm 1: Our algorithm for searching API migration examples.

| Input  | Code example repository exRepo and available API list apiList |
| Output | Migration example |

1 foreach api in apiList do
  2 apiCallInfo ← api.get(callInfo)
  3 apiParams ← apiCallInfo.getParamCount()
  4 apiName ← apiCallInfo.getName()
  5 potentialMigrations ← null
  6 foreach commit in exRepo do
        7 potentialMigrations ← commit.getAPIcalls(apiName, apiParams)
    end
  8 /* Check all potential migrations */
  9 foreach call in potentialMigrations do
        10 callAST ← call.buildAST()
        11 if callAST.matches(apiCallInfo) then
                  12            saveExample(call)
        end
    end
end

in the code examples to generate their corresponding ASTs. For each method invocation in the AST, we compare with the APIs that are potentially matched from the lexical search. If the AST can be fully built, we aim to obtain the perfect matches between method invocation and API declaration. In particular, a perfect match requires matching the method invocation name to an API declaration as well as having perfectly matched types for all parameters. In some cases, the code examples may not be fully complete (e.g., missing some external source code files or dependencies), leading to partially built ASTs. With the partially built ASTs, we consider a match if there exist the correct import statement of the API, the correct method name, and the correct number of parameters.

3) Learning API migration patterns: In this step, we search API calls for every version of the code examples. For instance, if code examples such as Google Samples [12], or FDroid projects [16] are hosted in version control repositories, we detect API calls in every commit of the repository.

We use the diff command from the version control repository to collect commits that contain changes to API calls. Similarly to the lexical search from the last step, the use of a diff tool is merely scoping down our search space. For the commits that potentially impact API calls, we parse the AST of the changed files in the commit. We compare the ASTs generated from the source code before and after the commit. If the API invocations in the AST are modified in the commit, we consider the commit as a potential API migration pattern. Hence, each API migration pattern consists of an AST built from the example before the migration and an AST built after the migration.

For the code examples that are hosted as text files outside of repositories, we apply a text diff on each two consecutive versions of the example instead of using commits to scope down the search space. The secondary AST matching step on
these text files is identical to that from the version control repositories.

B. Applying learned API migration patterns to API calls in the source code

In our second step, we collect all the API migration patterns (i.e., ASTs built from the example before and after the migration) from the first step and try to apply these patterns to the API calls in the Android app source code.

1) Searching possible migration candidates: Similar to our first step, to reduce the search space, our approach first uses API names to lexically search for API calls with available migration patterns in the source code of the targeted Android app.

Since migrations can be dependent on the context of surrounding code, we cannot assume that one migration example will suit all possible use cases. For example, API such as Resources.getColor(int) can be present in a variety of use cases. To match valid migration patterns, we must not only match API calls, but also determine if an example matches a user’s use case. We leverage data-flow graphs to match the API calls in the migration patterns and the targeted Android app sources. We construct a data-flow graph from the API call example “before” the migration in the migration patterns. We also construct data-flow graphs in the API calls in the Android app source code. Only if the data-flow graph from the example is a subgraph of a potential API call in the Android app source code do we then consider this a migration candidate. This allows us to assume that the example API being used as a migration pattern has a similar use case to the API call in the targeted Android app.

2) Applying the migration from the example to API calls in the source code: If any migration candidate is found, we then attempt the migration. We first compute the migration mappings by comparing the examples before and after API migration. This mapping contains any changes that must be made to existing code statements, obtained by comparing the names and types of each code statement in the data-flow graph. The migration mapping also contains any new code statements that are present in the “after” migration example but were missing in the “before” migration example.

In order to obtain an accurate migration mapping between the “before” and “after” examples (i.e., changes that were made to migrate the API), we need to eliminate the changes on AST that are not related to the API migration. We achieve this by relying on the data-flow graphs that are built from the examples. We first remove all the nodes in the data-flow
Algorithm 2: Our algorithm for migrating code.

Input: Migration mapping \texttt{mappedDFG} and client data-flow graphs \texttt{clientDFG}

Output: Migrated data-flow graph

\begin{verbatim}
1 /* Traverse all data – flow graphs */
2 foreach DFG in clientDFG do
3     \texttt{DFGMap} ← \texttt{mappedDFG}.\texttt{getDFG}()
4     \texttt{changedAPIs} ← \texttt{DFGMap}.\texttt{getChangedAPI}()
5     /* Migrate all migrateable APIs */
6     foreach \texttt{changedAPI} in \texttt{changedAPIs} do
7         \texttt{changedNodes} ← \texttt{DFGMap}.\texttt{getDataLinks}()\texttt{changedAPI()}
8         \texttt{missingNodes} ← \texttt{DFGMap}.\texttt{getNodesToAdd}()\texttt{changedAPI()}
9         /* Modify the data – flow graph */
10        \texttt{DFG}.\texttt{addNewNodes}()\texttt{missingNodes()}
11        \texttt{DFG}.\texttt{migrateDFG}()\texttt{migrateableNodes()}
12 end
\end{verbatim}

IV. Evaluation design

To determine the extent to which our approach, A4, can be used to assist with API migration, we conducted a series of experiments. We first present the research questions which we use to evaluate our approach. Then, we present our data acquisition approach for Android apps and code examples.

A. Research questions

We aim to evaluate our approach by answering three research questions. We present our research questions and their motivations in the rest of this subsection.

\textbf{RQ1} Can we identify API migration patterns from public code examples?

Our approach uses code examples to automatically learn API migration patterns. Therefore, the availability of migration code examples is very important to our approach. In this RQ, we study how many API migration patterns we can find in publicly available code examples.

\textbf{RQ2} How much effort is required to produce code examples?

Sometimes, apps may leverage APIs that do not have corresponding migration examples that are publicly available. In such cases, our approach can learn migration patterns from manually produced migration examples. However, if it is time consuming to manually produce an API migration example, our approach may not be very practical. Therefore, in this RQ, we conduct a user study to see how much time is required for users to manually produce a migration example.

\textbf{RQ3} How much assistance can our approach provide when migrating APIs?

In this RQ, we use the publicly available and user produced examples to automatically learn and apply migration patterns. We evaluate and quantify the extent to which our approach, A4, can leverage these examples to assist developers in migrating API calls.

B. Data acquisition

In this subsection, we present our data acquisition approaches for the Android apps that we want to migrate and the sources of our code examples.

\textbf{1) Android apps for migration}: We selected our Android apps through the free and open source repository of Android apps: FDroid [16]. We cloned all the git repositories of FDroid apps that are hosted on GitHub and implemented in Java. In total, we obtained 1,860 apps. From these apps, we would like to perform API migration on the ones that are still actively under development, since they are more likely to benefit from

graph where all the associated nodes are perfectly matched between the “before” and “after” examples. Since they are perfect copies of one another, those nodes cannot contribute to a migration. We keep the nodes in the data-flow graphs that remain unmatched and are associated with the node that is of interest to the API call. Finally, we compare the nodes that are kept to find the matched data-flow graph for the API call in the Android app after migration.

Once we obtain the most likely migrated data-flow graph in the “after” API migration example, we produce a backward slice of the data-flow graph starting from the API call. In other words, we only look at nodes that give data to the API call. Based on our sliced data-flow graph, we then map each node in the “before” example to the most closely matched node in the “after” example. Any unmatched data linked nodes are considered to be new nodes and are saved to be added during migration.

Finally, we use the migration mapping to transform the project source code into the “after” API migration example. Pseudocode of our migration algorithm can be found in Algorithm 2. The transformation also looks for any object types that are matches between the “before” example and the project source code to infer the names used in the Android app source code, to prevents the introduction of new variable names.

To better explain our approach, Figure 3 illustrates an example of applying a migration from an example to an API call. The \textit{before} and \textit{after} migration examples illustrate potential example code obtained from sample projects. We can see that methods which are exact matches are seen as irrelevant since they add no information to the migration. Similarly, we can see that nodes after the migration node of interest, in this case a return statement, are also seen as irrelevant nodes. This is because it does not matter what the user does with their code after the API call (i.e., not related to the usage of the API). Other nodes are then matched between \textit{before} and \textit{after} examples to determine which node is most likely the migration. Any nodes that do not obtain a match are considered new nodes required for migration and will be added in user projects, as denoted by the color blue in our example. Once a match is produced between migration examples, the migration mapping can be applied to the user example, as presented by the arrows in Figure 3 and as explained by the pseudocode in Algorithm 2.
migrating to the most recent Android APIs. Therefore, we only selected apps that had code committed in the six months prior to our study. In order to assist in later verifying the success of the migration, we selected only the apps that contained readily available tests. Furthermore, we only selected the apps that could be built with the official Android build system, Gradle, so that we could verify the functionality of apps after migration. We focused the evaluation of our approach for API migration on the 10 latest versions of the Android API since they account for 95.6% of Android devices in the world [19]. Therefore, we selected the apps that target the 10 latest versions (API versions 19-28) of the Android API. This allowed us to collect a sample of 164 FDroid apps on which to test our migrations. All the other apps (1,696) were used to extract API migration examples.

2) Sources of public code examples: Our approach relies on code examples to learn API migration patterns. In the evaluation of our approach, we focus on two sources of public code examples: 1) official Android examples, i.e., Google Samples [12] and 2) FDroid app development history. We also use Android API usage patterns extracted from our FDroid sample to construct the original examples given to our user study participants.

Google Samples. The Android development team provides code samples to assist app developers understand various use cases of the Android APIs. This code sample repository, aptly named Google Samples, contains 234 sample projects, 181 of which are classified as Java projects [12]. We mined the Google Samples repository, hosted on GitHub, for Android API migration examples.

FDroid app development history. Due to the widely available open source projects, if one open source project migrates a deprecated API in their source code, other developers may leverage that migration as an example. With the publicly available development history of FDroid, we can leverage the API migrations that exists in the FDroid apps as code examples.

V. Evaluation results

In this section, we present the results of our evaluation by answering three research questions.

**RQ1: Can we identify API migration patterns from public code examples?**

We focus on two sources of examples in this RQ: 1) official Android examples, i.e., Google Samples [12] and 2) FDroid app development history. We measure the prevalence of API migration examples by determining the number of API migration examples that we can mine from our example sources. We only consider the APIs with migrations that have officially been documented by Android developers to validate our results.

In the 10 latest versions of Android (versions 19-28), we were able to manually identify 262 APIs which had documented migrations. Out of all the Android APIs with migrations, only 125 of those occurred in our sources of examples.

![Fig. 3. Example of applying a migration from an example to Android app source code](image-url)

By searching for API migrations (c.f. Step 1 in our approach in Section III-A) in the Google Samples [12] and the development history of FDroid apps, our approach can automatically identify 10 and 82 API migration examples out of 125 API occurrences, respectively. Among those migration examples from Google Samples, only one API is not covered
in the FDroid examples, giving us a total of 83 distinct API migrations found (see Table I). However, we note that the nine overlapping examples from both sources are still all useful since various migration use cases can occur when we apply automated migration. Some examples may provide a more suitable migration with less human effort from developers (c.f. RQ3).

We developed a user study with 20 participants to examine the difficulty of producing API migration examples. From RQ1, we found that 42 possible API migrations in FDroid apps do not have readily available public examples. Among those APIs, we selected the 30 APIs that are called the most frequently in FDroid. Based on these 30 APIs, we produced 30 Java files, each of which contained a different simplified deprecated API call found within the source code of FDroid samples. We also prepared migration instructions that we took directly from the Android developers’ website [20]. We wanted to observe a scenario where the developers are aware that a needed migration exists, i.e., an API call is deprecated, and they must try to make a simple example to illustrate the new API code. The migration examples in our approach are not required to be in a compilable Java project, and participants are instructed as such to ease their example production. For instance, an example can be inside a text file with a pseudo class named “Test” and method name “bar” with no access to imported classes. The only required dependency is that the API call structure is compatible with the original example.

Each of the 20 participants was given a randomized sample of five different API calls to produce migration examples. Each API migration was individually timed to determine the amount of time needed to produce a migration. The first four questions in the questionnaire asked the developers to self-report their app development experience in years and months, and to rank their Java programming experience on a scale of five. Our participants were a mix of 6 professional developers and 14 graduate students. Our participants had a median of 1 month of Android app development experience, and their self-reported Java programming experience has a median of 3.5 out of 5 (where 3 represents an average confidence on programming in Java). Furthermore, each participant was asked to rank the difficulty of each migration on a scale of one to five, five being the most difficult.

The results obtained from each participant were manually examined to determine whether the example had been correctly made. Any mistakes found were classified into two categories: simple or complex. Simple mistakes are the ones that could have been addressed by using an IDE, e.g., simple type casting errors or spelling mistakes. Meanwhile, complex mistakes were seen as mistakes in the fundamental understanding of the API. Some examples of complex mistakes are: not completing the migration, keeping a deprecated method, or not properly instantiating a new object.

**User study results on the effort of producing migration examples.** Table I shows the results of our user study. We examine the ease of producing migration examples based on a prior classification system, which classifies the API migrations into three categories: fully automatable, partially automatable and hard to automate [21]. For each example, we only consider a migration successful if it is produced without any error.

We find that it is not a challenging task for the participants to produce the migration examples even for the ones that are hard to automate. On average, successful and faultless migrations could be produced 81% of the time for the 100 samples (30 APIs) that are presented to the participants. The APIs were ranked on average to have a difficulty rating of 2.2 out of 5 (5 being the most difficult). It took on average only 212 seconds for a participant to complete a migration example. In short, from our finding we see that even novice Android API users can produce quality examples.

The participants of the user study made several types of errors in their migration code. Some participants made simple mistakes such as casting errors, syntax errors, or calling an incorrect but similarly named API. However, we also observed more complex errors such as failure to completely migrate away from the original API, failure to consider new conditions generated by the migration, and failure to understand the consequences of the new functionality. In total, the user study participants made 21 errors, 10 of them were simple and 11 of them were complex. In particular, the most common type of migration is change type which belongs to the hard to automate category. This type of migration category encompasses any change in return type or a type change to one or more API parameters. In the user study, participants made eight errors for the change type examples, seven of them were simple casting errors or syntax mistakes, and the other was a complex error where the migration was unfinished.

**Our user study shows that it is not a challenging task to manually produce faultless API migration examples to complement public code examples. Novice Android API users can generate a migration example in around three minutes on average.**

**RQ3: How much assistance can our approach provide when migrating APIs?** In this RQ, we examine whether our approach can provide assistance to API migration in Android apps. In particular, we apply our approach to migrate 80 API calls in 32 FDroid apps (based on both the migration patterns learned from from public code examples and user

### Table I

| Sample                        | API identified |
|-------------------------------|----------------|
| FDroid API migrations         | 82             |
| Google Samples API migrations | 10             |
| Total distinct API migrations found | 83 |
| Total API uses found in apps  | 125            |
| Total possible                | 262            |
generated examples). In order to examine whether a migration is successful, we leverage the tests that are already available in the apps and collect the ones that can exercise the migrated API calls. In order to avoid tests that are already failing before the migration, we run all the tests before the migration of the apps and only keep apps with passing builds. The apps that we selected had an average of 10 tests, with a minimum of 6 tests and a maximum of 52. Afterwards, we try to build the migrated apps. For the apps that can be successfully built, we run the collected tests again to check whether the migration is completed successfully.

Previous studies have shown that producing exact automated migrations is a difficult task [21]. We consider this fact and attempt to mitigate it through the use of our approach. However, in cases where it is possible to present an exact migration to a user, such a task should be attempted to save development time. Therefore, in the best case scenario we attempt to provide an exact and automatic migration to users. Our approach can provide assistance in 71 out of 80 migrations through faultless migration, migrations with minor modifications, and through the examples we suggest when we experience unmatched migrations, migrations with minor modifications, and through the use of our approach. However, we consider the number of tokens that must be changed for the migration to be successful. We use the absolute number of tokens changed rather than a percentage of tokens matched since the automatically matched tokens do not require any effort. We consider any modifications to a code token as a token modification.

We found that the modifications needed for an imperfect but successful migration requires modifying between one to seven tokens (3.65 tokens on average). We consider the amount of effort needed on such modifications rather small, especially since they are mostly simple renames and the addition or removal of keywords. The brunt of the work, namely finding a migration candidate, finding a migration pattern and matching this pattern, is provided by our approach, A4. Therefore, the user must simply “glue” any unattached pieces of example code into their application.

We examined the different scenarios that require minor modification to qualitatively understand the effort needed for such modifications. In total, we identify four reasons for such modifications: variable renaming, missing the keyword this, wrongly erasing casting, and removing API calls.

As a design choice, we opt to conservatively not remove any API calls for migration, since mistakenly removing API calls may cause large negative impact to developers. Instead, our approach tells the users that a change must be made to the API, and the API call must then be manually removed. Although this may require the manual modification of removing several tokens, the effort of the change is minimal. We experienced three removing API call cases. For example the View.setDrawingCacheQuality(int) method was made obsolete in API 28 due to hardware-accelerated rendering [20]. This means that old code referring to drawing cache quality can be simply deleted, as the OS now handles this through hardware.

**Unmatched guidance.** It is possible for our approach to fail to match an example to a known migration (see Section III-B1). These cases exist due to the nature of our example matching.

---

**TABLE II**

| Ease of migration (based on [21]) | Migration type | Examples | Samples | Avg. migration time (s) | Avg. migration success (%) | Avg. rated difficulty |
|----------------------------------|----------------|----------|---------|------------------------|---------------------------|----------------------|
| **Fully automateable** | Move method | 1 | 4 | 153 | 75 | 2.0 |
| | Rename method | 4 | 13 | 54 | 100 | 1.2 |
| | Remove parameter | 1 | 3 | 260 | 67 | 2.7 |
| | Encapsulate | 1 | 3 | 48 | 100 | 1.0 |
| **Partially automateable** | Expose implementation | 7 | 25 | 276 | 59 | 2.4 |
| **Hard to automate** | Change type | 10 | 37 | 232 | 79 | 2.2 |
| | Add contextual data | 5 | 17 | 175 | 77 | 2.5 |
| | Replaced by external | 1 | 3 | 153 | 100 | 1.0 |
| **Total** | | 30 | 100 | | 212 | 81 | 2.2 |

**TABLE III**

| Automated migration results based on migration patterns learned from three different sources | FDroid | Google Samples | User study |
|----------------------------------|---------|----------------|-------------|
| Faultless migration | 5 | 0 | 9 |
| Migrated with minor mod. | 5 | 4 | 12 |
| Unmatched guidance | 18 | 1 | 9 |
| False positive | 0 | 0 | 3 |
| Ex. was not a migration | 6 | 0 | 0 |
| Unsupported cases | 7 | 0 | 1 |
| **Total migrations** | 41 | 5 | 34 |
| **Distinct API** | 21 | 4 | 15 |

**Faultless migration.** If all the tests are passed without any further code modification, we consider the migration as exact. We succeeded at giving users an exact migration in 14 cases. Nine of these migrations were learned from the manually produced code examples from our user study. However, we were able to use five examples from FDroid app development history to produce faultless migrations. Such a result tells us that given a rich example, it is possible to provide fully automated working migrations.

**Migration with minor modifications.** In cases where an app, after the migration, does not build or does not pass the tests, we manually checked the error message. In such cases, the migration pattern may be correct, yet our automatically generated migration may need minor modifications to build the app and pass the tests. For each case, we determined a way for the migration to succeed with minimal code modification. An example of such modification can be adjusting a variable name. We were able to provide 21 migrations with minimal modifications.

We consider the number of tokens that must be changed for the migration to be successful. We use the absolute number of tokens changed rather than a percentage of tokens matched since the automatically matched tokens do not require any effort. We consider any modifications to a code token as a token modification.

We found that the modifications needed for an imperfect but successful migration requires modifying between one to seven tokens (3.65 tokens on average). We consider the amount of effort needed on such modifications rather small, especially since they are mostly simple renames and the addition or removal of keywords. The brunt of the work, namely finding a migration candidate, finding a migration pattern and matching this pattern, is provided by our approach, A4. Therefore, the user must simply “glue” any unattached pieces of example code into their application.

We examined the different scenarios that require minor modification to qualitatively understand the effort needed for such modifications. In total, we identify four reasons for such modifications: variable renaming, missing the keyword this, wrongly erasing casting, and removing API calls.

As a design choice, we opt to conservatively not remove any API calls for migration, since mistakenly removing API calls may cause large negative impact to developers. Instead, our approach tells the users that a change must be made to the API, and the API call must then be manually removed. Although this may require the manual modification of removing several tokens, the effort of the change is minimal. We experienced three removing API call cases. For example the View.setDrawingCacheQuality(int) method was made obsolete in API 28 due to hardware-accelerated rendering [20]. This means that old code referring to drawing cache quality can be simply deleted, as the OS now handles this through hardware.
Since we consider the data-flow surrounding a migration, our examples must contain a similar data-flow graph to the Android app considered for migration. We conservatively opt for such an approach to reduce the number of falsely generated migrations, since they may introduce more harm to developers than assistance. As a result, if the Android app instantiates their API call in a different way than our examples, our approach will not attempt to automate the migration. For example, if the user uses nested method calls inside an API call and we cannot reliably map the return types of the nested method calls, we will not consider the code to match. Examples that contain unmatched but similar API migrations will be presented to the user, and they can choose the correct migration afterwards.

For the 28 migrations for which none of our examples contained a match, 18 of these are from FDroid development history. This implies that the FDroid API usage is more often tailored for the apps’ needs and is not often coded in a general manner. Future research may investigate automatically generalizing the usage of API to address this issue.

**False positives.** We consider a migration to be a false positive when our approach presents an unnecessary migration. This would occur if a migration example were erroneously matched to a non-migrateable method invocation in Android app code. This only occurred three times in our tests and the occurrences were all from the manually produced examples. We believe this to be the case due to the simplicity of the manually produced examples. Since the examples are simplified and contain very little context code, the corresponding data-flow graphs are often simple. This allows the examples to be used in a wider range of situations than the mined code samples. However, this leads to false positives as a trade-off, especially if method names are commonly used and few parameters are used (e.g. `setContent()`).

**Example was not a migration.** As previously mentioned, since we obtain our migration examples through automated tool assistance (c.f. RQ1), it is possible for our approach to present faulty migration examples. This normally occurs when a deprecated API call is modified, but not migrated to the updated version of the API. An example of such a modification would be to rename the variables in the API call. This change would be mistakenly identified by our approach as a modification to an API call. Therefore, if such an example were to be used to attempt a migration, we would present a migration with no effect, and therefore cause no harm to the Android app. We experienced six such instances in the FDroid examples. However, we did not encounter these in our Google Samples nor the manually produced examples.

**Unsupported.** There exist a few unsupported corner cases that our approach would not support. For example, we cannot automatically produce migrations if the API call is spread across a try catch block, or within a loop declaration or conditional statement declaration. As with unmatched guidance cases, we provide the user with migration examples that may be relevant to their migration, instead of attempting to provide automated migration on their code. We experienced eight such cases.

Our approach can provide API migration assistance in 71/80 cases. We can automatically generate 14 faultless migrations, 21 migrations with minor code changes, and 36 migrations with useful guidance to developers. The effort needed to post-modify our generated API migration is low, an average of 3.65 tokens require modification.

VI. RELATED WORK

In this section we discuss prior research in the field of APIs. We concentrate on prior work based on API evolution, API migration, and the usefulness of code examples for APIs.

**API evolution studies**

As the world of software APIs expanded so did the number of studies on API usage and evolution [2], [22]–[27]. More recently, several research papers have been published on the Android API and its evolution [5], [6], [38]–[41]. The Android API is both large in code size and has numerous versions. It has therefore been found to be a suitable case study for various types of software research, from API usage [42] to mutation testing [43], Stack Overflow discussions [44], and user ratings [4]. We use the Android API as a case study for API evolution and migration.

In 2013, using an extraction tool created for their study, McDonnell et al. [5] extracted the change information of the Android API. McDonnell et al. find that the Android API is rapidly evolving and provides an average of 115 API updates per month. API updates are said to occur to fix bugs, enhance performance, and to respect new standards [5]. They suggest that although there are currently many tools to automate the API updating process, these tools are insufficient for current needs and that new studies and tools are necessary to encourage proper API updates [5]. The work by McDonnell et al. is a strong motivation for the ideas presented in this paper. Due to the rapid nature of changes in the Android API, it is necessary to have tools such as the one presented in this paper to aid developers in adapting API changes.

Li et al. [38] built a research tool to investigate deprecated Android APIs. They find that 37.87% of apps make use of deprecated APIs [38]. This knowledge further strengthens the need for tools to help developers migrate away from these methods. They also found that although the Android framework developers consistently document replacements for deprecated APIs, this documentation is not always up to date [38]. This presents a perfect opportunity for tools such as the one presented in this paper to use the most up-to-date examples to help users migrate away from deprecated API rather than rely on potentially out of date documentation.

**API migration techniques**

Past research has presented tools and suggestions to improve API migrations [3]–[10], [45], [45]–[52]. API migration tools usually rely on fully automatic or semi-automatic means to provide API migration recommendations to developers. These approaches concentrate on locating a migration candidate and recommending an alternative API. Meanwhile, our approach assumes that the user knows an alternative API exists, either
through these tools or through documentation, and does not understand how to produce the migration. This is where our approach, A4, aims to guide users.

CatchUp! [10] provides a semi-automatic migration solution which requires that API developers record API source code changes from their IDE. These changes can then be replayed on API user systems to adapt their application to the changed API. This tool can provide safe migrations due to the nature of the tool chain. However, the tool requires the API developers to record their own work. Contrarily to CatchUp!, the tool presented in this paper can use a myriad of examples to guide the migration from any available source.

Dagenais and Robillard, and Wu et al. provide different semi-automatic migration tools, SemDiff [9], and AURA [8]. Both tools similarly mine two versions of an API for method changes, as well as internal adaptations to these API changes. These internal changes are then used to provide migration recommendations to API users. SemDiff presents multiple migration scenarios by heuristically ranking the most likely candidates. Meanwhile, AURA uses heuristics to provide only the best fit migration. Our approach can consider any source of example to extract migrations. This allows users to obtain high quality migration examples even if the API source code is proprietary, as long as open source examples of migrations exist or can be created.

Code examples and APIs

The approaches presented in this report rely on the existence of source code examples of API migrations. Prior research has covered API documentation enhancement and API examples extensively [53]–[66].

EXAMPLERO presents a visualization tool to assist users in understanding common uses of APIs [67]. The tool uses a corpus of API examples to build common usage patterns which can then be displayed to a user in an interactive graphical way. Since tools like EXAMPLERO have shown that they can assist users to answer API questions with detail and confidence [67], we believe that using examples to provide API insights could similarly be used for a migration tool.

Tools such as Sydit and LASE provide systematic editing through examples [57], [68]. However, these tools require well-built ASTs for dependency analysis, but examples of API migration may not always be compilable (e.g., code snippets from documentation, or examples from users). Furthermore, a developer has to provide the examples with relevant context (e.g., where to apply the changes). In contrast, our approach automatically mines relevant examples, and then finds the most relevant migration pattern for a specific API migration use case with no dependency assumptions.

MAPO is a tool which allows developers to rapidly search for frequent API usages [69]. MAPO uses a combination of a lightweight source code analyzer, various code search engines, and frequent item-set mining methods [69]. Tools like MAPO have shown that it is possible to rapidly and reliably find examples of API usage on the web. Therefore, tools like A4 can rely on this foundation to use these examples for more complex tasks, such as aiding API migration.

VII. Threat to Validity

Construct validity. We assess the validity of our API migrations by running the test suites of the migrated Android apps. Although we focus on the tests that exercise the migrated API calls, it is still possible that defects introduced by the migration are not identified by the tests. User studies on the app users may complement the evaluation of our approach. In our study, there are still cases where our approach cannot migrate faultlessly or only provides guidance. As an early attempt of this line of research, our approach can be complemented by other techniques such as code completion to achieve better assistance in API migration.

External validity. Since this entire study was tested on the Java API of the Android ecosystem, it is possible that the findings in this paper will not generalize to other programming languages. However, while it is true that the approach presented in this paper was tested specifically on a Java based API, all of the approaches are built upon assumptions that are true in other popular programming languages such as C#.

Internal validity. Our findings are based on the Android project and code examples mined and produced for its API. It is possible that we only found easy migrations. It is also possible for the time gap between the release of new API and the update to examples to be larger in other sources. We attempted to mitigate these threats through mining official samples, user projects, and producing a user study in which participants produced migration examples for Android APIs which were taken from frequently used APIs in Android apps. We found that Google Samples updated deprecated API as soon as one month after the release of a new API version, which should allow developers to regularly update their apps. This user study provided us with new and useful API migration examples, showing that the premise of using examples to help automate API migrations is functional and likely dependent on the sample size of examples and not on the difficulty of the examples.

VIII. Conclusion

In this paper, we proposed an approach that automatically assists with Android API migrations by learning API migration patterns from code examples. We evaluate our approach by applying automated API migrations to 32 open-source Android apps from FDroid. We find that our approach can automatically extract API migration patterns from both public code example and manually produced API examples that are created with minimal effort. By learning API migration patterns from these examples, our approach can provide either automatically generated API migrations or provide useful information to guide the migrations. This paper makes the following contributions:

- We propose a novel approach that learns API migration patterns from code examples.
- Our novel approach can automatically assist in API migration based on the learned API migration patterns.

Our approach illustrates the rich and valuable information in code examples that can be leveraged in API related software engineering tasks.
REFERENCES

[1] R. L. Glass, “Frequently forgotten fundamental facts about software engineering.” IEEE Software, vol. 18, no. 3, pp. 112–111, 2001.

[2] P. K. Venkatesh, S. Wang, F. Zhang, Y. Zou, and A. E. Hassan, “What do client developers concern when using web apis? an empirical study on developer forums and stack overflow.” 2016 IEEE International Conference on Web Services (ICWS), 2016.

[3] A. A. Sawant and A. Bacchelli, “fine-grape: fine-grained api usage extractor – an approach and dataset to investigate api usage,” Empirical Software Engineering, vol. 22, no. 3, p. 1348–1371, Apr 2016.

[4] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, “The impact of API change- and fault-proneness on the user ratings of android apps,” IEEE Transactions on Software Engineering, vol. 41, no. 4, pp. 384–407, 2015.

[5] T. Mcdonnell, B. Ray, and M. Kim, “An empirical study of api stability and adoption in the android ecosystem,” 2013 IEEE International Conference on Software Maintenance, 2013.

[6] M. Lamothe and W. Shang, “Exploring the use of automated api migrating techniques in practice: An experience report on android,” MSR ’18: 15th International Conference on Mining Software Repositories, 2018.

[7] A. A. Sawant, R. Robbes, and A. Bacchelli, “On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK,” Empirical Software Engineering, pp. 1–40, 2017.

[8] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, “Aura: A hybrid approach to identify framework evolution.” Proceedings of the 32nd ACM/IEESE International Conference on Software Engineering - ICSE '10, 2010.

[9] B. Dagenais and M. P. Robillard, “Semdiff: Analysis and recommendation support for api evolution,” 2009 IEEE 31st International Conference on Software Engineering, 2009.

[10] J. Henkel and A. Diwan, “Catchup! capturing and replaying refactorings to support api evolution.” Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005, 2005.

[11] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Redamcher, T. N. Nguyen, and D. Dig, “Api code recommendation using statistical learning from fine-grained changes,” Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016, 2016.

[12] “Samples, android developers.” [Online]. Available: https://developer.android.com/samples

[13] C. Treude and M. P. Robillard, “Augmenting api documentation with insights from stack overflow,” Proceedings of the 38th International Conference on Software Engineering - ICSE 16, 2016.

[14] S. Wang, I. Keivanloo, and Y. Zou, “How Do Developers React to RESTful API Evolution?“ 12th International Conference, ICSC0 2014, Paris, France, November 3-6, 2014. Proceedings, vol. 8831, pp. 245–259, 2014.

[15] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, “Mining energy-greedy API usage patterns in Android apps: an empirical study,” Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014, pp. 2–11, 2014.

[16] “F-droid,” 2017. [Online]. Available: https://f-droid.org/

[17] “Android developers resources,” Jul 2012. [Online]. Available: https://developer.android.com/reference/android/content/res/Resources#getColor(int)

[18] W. Beaton, “Eclipse corner article abstract syntax tree, the eclipse foundation.” [Online]. Available: http://www.eclipse.org

[19] “Distribution dashboard android developers.” [Online]. Available: https://developer.android.com/about/dashboards

[20] “Package index,” Jul 2017. [Online]. Available: https://developer.android.com/reference/packages

[21] B. E. Cossette and R. J. Walker, “Seeking the ground truth: A retroactive study on the evolution and migration of software libraries.” in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ser. FSE ’12. New York, NY, USA: ACM, 2012, pp. 55:1–55:11.

[22] D. Dig and R. Johnson, “The role of refactorings in API evolution.” IEEE International Conference on Software Maintenance, ICSR, vol. 2005, pp. 389–400, 2005.

[23] D. Mendez, B. Baudry, and M. Monperrus, “Empirical evidence of large-scale diversity in api usage of object-oriented software,” in 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), Sep 2013, pp. 43–52.

[24] K. Taneda, D. Dig, and T. Xie, “Automated detection of api refactorings in libraries,” in Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ser. ASE ’07. New York, NY, USA: ACM, 2007, p. 377–380.

[25] J. Businge, A. Serebrenik, and M. van den Brand, “Analyzing the eclipse api usage: Putting the developer in the loop,” in 2013 17th European Conference on Software Maintenance and Reengineering, March 2013, pp. 37–46.

[26] S. Wang, I. Keivanloo, and Y. Zou, “How do developers react to restful api evolution?” in Service-Oriented Computing, X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 245–259.

[27] J. Li, Y. Xiong, X. Liu, and L. Zhang, “How does web service API evolution affect clients?” Proceedings - IEEE 20th International Conference on Web Services, ICWS 2013, no. 61121063, pp. 300–307, 2013.

[28] C. Jaspan and J. Aldrich, “Checking framework interactions with relationships,” in ECOOP 2016 - Object-Oriented Programming. S. Drossopoulou, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 27–51.

[29] D. Dig and R. Johnson, “How do apis evolve? a story of refactoring,” Journal of Software Maintenance and Evolution: Research and Practice, vol. 18, no. 2, pp. 83–107, 2006.

[30] M. Sridharan, S. Artzi, M. Postoia, S. Guarneri, O. Tripp, and R. Berg, “Ft: Taint analysis of framework-based web applications,” in Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, ser. OOPSLA ’11. New York, NY, USA: ACM, 2011, pp. 1053–1068.

[31] W. Wu, A. Serveaux, Y. G. Guéhéneuc, and G. Antoniol, “The impact of imperfect change rules on framework API evolution identification: an empirical study,” Empirical Software Engineering, vol. 20, no. 4, pp. 1126–1158, 2015.

[32] R. Holmes and R. J. Walker, “Informing eclipse api production and consumption,” in Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange, ser. eclipse ’07. New York, NY, USA: ACM, 2007, pp. 70–74.

[33] J. Businge, A. Serebrenik, and M. G. J. van den Brand, “Eclipse api usage: the good and the bad,” Software Quality Journal, vol. 23, no. 1, pp. 107–141, Mar 2015.

[34] C. Jaspan and J. Aldrich, “Checking framework interactions with relationships,” in ECOOP 2009 – Object-Oriented Programming. S. Drossopoulou, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 27–51.

[35] M. W. Godfrey, “Practical data exchange for reverse engineering frameworks: Some requirements, some experience, some headaches,” SIGSOFT Softw. Eng. Notes, vol. 26, no. 1, pp. 50–52, Jan. 2001.

[36] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, “How a good software practice thrwarts collaboration: The multiple roles of apis in software development,” in Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, ser. SIGSOFT ’04/FSE-12. New York, NY, USA: ACM, 2004, pp. 221–230.

[37] M. P. Robillard and R. Deline, “A field study of api learning obstacles,” Empirical Softw. Engg., vol. 16, no. 6, pp. 703–732, Dec. 2011.

[38] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, “Characterising deprecated android apis,” in Proceedings of the 15th International Conference on Mining Software Repositories, ser. MSR ’18. New York, NY, USA: ACM, 2018, pp. 254–264.

[39] M. Mahmoudi and S. Nadi, “The android update problem: An empirical study,” in Proceedings of the 40th International Conference on Software Engineering, ser. MSR 2018. New York, NY, USA: ACM, 2018.

[40] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, “Api change and fault proneness: A threat to the success of android apps,” in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 477–487.

[41] P. Calciati, K. Kuznetsov, X. Bai, and A. Gorla, “What did really change with the new release of the app?” in Proceedings of the 15th International Conference on Mining Software Repositories, ser. MSR ’18. New York, NY, USA: ACM, 2018, pp. 142–152.
