Data-Driven Inference of API Mappings

Amruta Gokhale, Daeyoung Kim, Vinod Ganapathy
{amrutag, daeyoung.kim, vinodg} @cs.rutgers.edu

Abstract

Porting mobile applications (apps) from one platform to another is one strategy used by developers to write cross-platform apps. One challenging task in porting is transforming the app so as to use the appropriate platform-specific APIs. We have proposed a novel approach to extract functionally equivalent API methods of two platforms that simplifies this task. Our approach is inspired by a technique in natural language processing domain which extracts a translation dictionary from non-parallel corpora of two natural languages. We demonstrate a prototype implementation of the proposed approach.

Keywords API mapping, cross-platform porting, mobile apps, static analysis, inference, data-driven methods.

1. Introduction

There is a variety of mobile platforms available today such as iOS, Android, and Windows Phone. Successful adoption of each of these platforms depends on the availability of a wide range of apps. Platform providers are therefore keen to assist mobile app developers [11, 13] who wish to port their apps to the given platform. App developers are also eager to port their app to different platforms so as to reach a larger user base. Thus, there are different players in the mobile devices' ecosystem interested in making the porting process of mobile apps easier.

However, porting a mobile app across different mobile platforms is challenging. One question faced by developers during porting is: How do you adapt the app to use the respective platform-specific API? Identifying functionally equivalent methods between two APIs can simplify this process, but doing so is difficult. It can be done manually by consulting the documentations of both the APIs, but this process is time consuming and error-prone. Automatically building a *database* of mappings between the source and target APIs solves this problem. The database contains mappings consisting of each source API method paired with a target API method implementing the same functionality. For example, one mapping in the database of mappings between iOS API and Android API can be `CGGeometry.CGRectMake` $\mapsto$ `Paint.drawRect` (since both methods draw rectangle on the screen). It is possible to have one source API method mapped to multiple target API methods, representing different ways of implementing same functionality. Developers can lookup this database to find equivalent API methods on the given platform.

We propose to automatically build a database of mappings between source and target API methods. Our approach first collects a set of apps on source and target platforms. We then statically analyse the reverse-engineered code of the app to construct program paths. These program paths represent potential execution steps which the app might undertake at runtime. The program paths are constructed to contain only the API method invocations. A few statically derived paths may be infeasible during actual app execution. However, in a huge collection of program paths, effects of infeasible paths will be suppressed because of being outnumbered by feasible paths. Each program path forms a *sentence* in an unknown language whose *words* are individual API methods. A collection of program paths thus form the text corpus of the unknown language. We feed the two text corpora to an inference engine which extracts mappings between *words* of the two languages. The inference engine is a technique in natural language processing (NLP) domain [9] that has been successfully applied to infer mappings between words of two natural languages.

The technique *learns* the mappings between words of two languages via statistical inference. Assuming that the two given natural languages have some common vocabulary, the technique works in the following manner. In large corpora of text from two languages, appearances of words that are translations of each other exhibit similar attributes (*e.g.,* frequency of words) in the two corpora. The similarity in the attributes' values can be attributed to the presence of a common, hidden concept that can be imagined to have given birth to the observed attributes. We can compute the mappings by applying statistical modelling algorithms that explain the observed attributes via a generative model.

We believe that the same technique that worked for natural languages would also work for API corpora. First, given the vast number of similar kinds of apps available across two platforms, the APIs exposed to app developers must contain some common methods. Second, from our previous work [8], we have some evidence to suggest that the attributes of

SPLASH '14 Companion, Oct 20-24 2014, Portland, OR, USA.
ACM 978-1-4503-3208-8/14/10.
http://dx.doi.org/10.1145/

appearances of API methods in the apps exhibit similarities. Thus, the assumptions on which the NLP technique works are also true in our case of two mobile APIs.

We have built a tool called DDR - Data-Driven Rosetta - that infers mappings between iOS API and Android API. We have tested the tool on a small dataset of 13 iOS apps and 13 Android apps. We aim to scale the evaluation on a large set of apps.

2. Motivation

In our previous work [8], we addressed the problem of inferring mappings between API methods of two different mobile platforms using independently developed mobile apps. The intuition behind our approach was that if two apps implemented the same high-level functionality (*e.g.,* both are TicTacToe games), the developers of the two apps must have used functionally equivalent API methods in implementing the same high-level functionality. If we exercised similar functionality while running the apps, the two apps would take similar execution paths and thus would invoke similar API methods. The traces generated should therefore contain some of the functionally equivalent API methods that map to each other.

Our prototype tool, Rosetta, worked as follows. We collected functionally equivalent pairs of apps on two platforms (*e.g.,* two independently developed TicTacToe games). We executed the two apps in each pair independently but in similar fashion, *e.g.,* by selecting similar menu options or clicking on similar GUI features. We collected the runtime traces of the apps on both the platforms, consisting of sequences of API methods invoked by the apps during these executions. The generated traces served as input to our API mapping inference algorithm. The inference algorithm was based on a belief propagation method [14] and used factor graphs [10] to infer mappings between API methods appearing in the two traces. We repeated the process of trace generation and inference for each pair of apps and then combined the inference results to give a final set of mappings.

While we had some success with this approach with a small set of apps, we realized that it suffers from a number of shortcomings. First, the technique requires independently-developed app pairs that implement the same high-level functionality and have similar GUIs. Since apps implementing the same high-level functionality often differ in features' richness and GUI, it is not easy to collect such apps. Second, the technique requires the user of Rosetta to manually execute the app pairs in similar manner which is a time-consuming process. Thus, the technique is not scalable to experiment with large number of apps. This directly limits the number of mappings that can be inferred.

We developed a new approach based on static analysis that overcomes the above drawbacks in following ways: First, it does not place any requirement on the nature of apps that we can play with. Second, it analyses the apps' data statically instead of at runtime. At a high level, the technique first generates apps' data on both the platforms by analysing the app binaries statically. This data is treated in the same light as the text from two different natural languages. We use

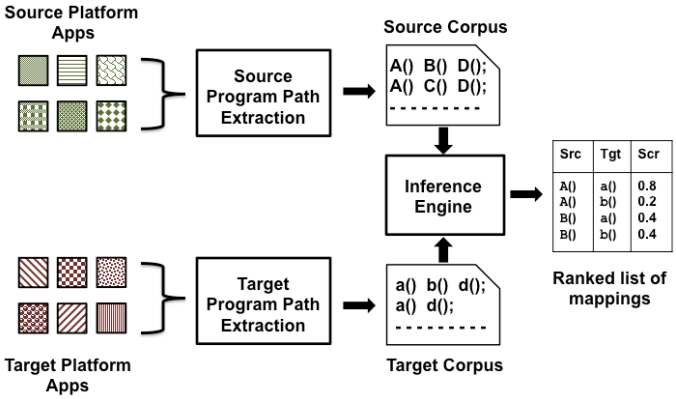

Figure 1. System design

a method employed previously in the NLP domain to infer a translation dictionary between independently written text corpora of two different natural languages. The output is a set of mappings between API methods of two platforms. We will explain the technique in more detail in Section 4.

3. Background

We now describe how we reformulate the problem of finding mappings between API methods of two mobile platforms into the problem of finding matchings between words of two natural languages.

In NLP and machine learning community, researchers have addressed the problem of inducing translation dictionary between two languages from parallel (translated) or non-parallel (independently written) text in two languages. In this paper, our focus is to extract mappings between API methods of two different mobile platforms. For each platform, we analyze apps statically and produce program paths. Program paths represent possible execution sequences that the app might take. We filter out all other program elements except API method invocations during the construction of program paths. Each program path can be viewed as a *sentence* from an unknown language whose *words* are individual API methods. Thus, a collection of program paths can be viewed as the text corpus of an unknown language. We generate two such collections for each of the two mobile platforms. Further, in order for the approach to be scalable, we do not want any restriction on the kinds of apps that we can experiment with. Hence the two sets of program paths should be treated similar to non-parallel text of two natural languages. We have thus redefined our original problem of extracting API mappings as a problem of finding inferring word translations from non-parallel corpora of two languages.

We leverage one work [9] in NLP that induces translation dictionary between two languages from non-parallel corpora. At a high level, the method uses statistical inference technique with an optional seed set of mappings given as input. It computes monolingual feature vectors of words using their attributes in the respective corpora. It establishes a generative model that iteratively explains the observed data

of feature vectors and the mappings inferred so far (if any). The model's parameters are computed by maximizing the log-likelihood of the observed data. The method casts this optimization problem as a maximum bipartite graph matching problem. Output of the matching algorithm is given as its output.

4. System Design

The overall system design is shown in Figure 1 tailored to iOS and Android as the source and target platforms, respectively. We now describe the design in more details.

Our approach takes two sets of apps developed for two different mobile platforms as input. We leverage the notion of category of apps in the mobile app markets to ensure that the fraction of apps belonging to each category is approximately the same across two platforms. This is easy to ensure since app markets of all popular mobile platforms have similar catgories of apps because of expectations from the users to have these apps available across each platform. Using such a dataset of apps helps to ensure that the generated program path data comes from related domains of apps rather than completely unrelated domains.

(1) Generate program paths. Given the apps for two platforms, the next step is to collect the program paths. We first decompile the apps' binaries using the available decompiling tools for the respective platforms to obtain bytecode representation of the apps. We then construct a control flow graph (CFG) from the decompiled bytecode representation using static code analysers for the respective languages. The next step is to traverse the CFG to produce static program paths of apps. For each function in the source code, we first construct its CFG representation that contains only the platform API methods, filtering out everything else. Figure 2 shows the generated CFG of an iOS app. We then convert the CFG into a directed acyclic graph (DAG) by removing back edges. This step ensures that we do not run into infinite number of paths. Here's how the program path generation would work on the CFG given in Figure 2:

(a) Identify all the entry and exit nodes in the graph. There is a single entry node *I1* and there are two exit nodes, *I3* and *I4*.

(b) For each pair of (*entry*, *exit*) nodes, compute all paths that start at *entry* and end at *exit*. You can use any standar graph algorithm to find all paths between two nodes of a DAG. For the pair (*I1*, *I3*), there is a single path in between them that goes via *I2*. Similarly for the pair (*I1*, *I4*), only one path exists in between them which goes through *I2*.

(c) For each path, print the sequence of API methods inside each of the node, in the same order as the nodes appear in the path. For the given graph, we generated two node sequences: *I1 I2 I3* and *I1 I2 I4*. These would yield two program paths: *S1* and *S2* as shown in Figure 4.

Similarly, for Android platform, program paths would be *T1* and *T2* shown in Figure 4.

(2) Infer mappings. The next step is to feed these program paths composed of sequences of API methods to the inference engine which gives mappings between API methods. We directly rely on the MCCA method [9] as our inference engine which uses a statistical modelling technique. The MCCA method takes two text corpora, one in source language and the other in target language as input. The two corpora can be completely unrelated and can be from different domains. At a high level, the method defines a generative model over the observed data which consists of the feature vectors of words in the corpora and the mappings generated so far. The generative model explains the observed data in terms of vectors in a common, hidden space. Their model uses canonical correlation analysis [6] which is one way of measuring the relationships between two sets of variables. If we view the generated source and target API program paths as belonging to two unknown source and target languages, we can feed the program paths to MCCA method to obtain mappings between API methods.

Let S and T denote the iOS corpus and Android corpus shown in Figure 4. We feed these two corpora to MCCA method. The method outputs a set of mappings between the elements of the corpora which are API methods in our case. In MCCA model, the objective is to maximize the log-likelihood of the observed data (feature vectors and the mappings generated so far). It casts this optimization problem as a maximum weighted bipartite matching problem over a graph consisting of source words and target words. The weight on the edge between a source word and target word denotes confidence in the mapping. The default model outputs at most one mapping for each API method. We modified the default model so as to output top 10 mappings for each source API method corresponding to top 10 weights on edges originating from the given source API method. We consider all non-negative weights on edges, in case the given API method does not have 10 mappings in the final bipartite matching output.

5. Implementation and Evaluation

We now describe the implementation details of our tool DDR which currently infers mappings between iOS and Android APIs.

iOS program paths generation. We decrypt the downloaded iOS apps using a tool called Clutch [1]. To disassemble the iOS binaries, we use a popular disassembler called IDA Pro. Similar to the technique employed in [7], we use backward slicing [12] to overcome a limitation of IDA Pro to resolve the actual targets of methods calls. For constructing CFG, we use IDAPython, an IDA Pro plugin that allows python scripts to run in IDA Pro, and also use NetworkX library [4] which provides support for graph functions. We then traverse the CFG as explained in Section 4. If necessary, we limit ourselves to generate paths upto a certain length for practical reasons.

Android program paths generation. Due to easy availability of tools to analyse Java bytecode, we first retarget Android apps' Dalvik bytecode to Java bytecode using dare [2] tool. We analyse the resultant Java bytecode using a popular static analyser for Java programs called Soot [5]. We make use of Soot's API to construct intra-procedural control flow graph (CFG) of functions defined in the app. We then

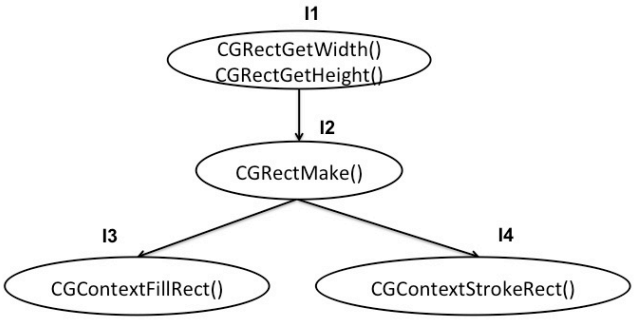

Figure 2. Control flow graph of an iOS app.

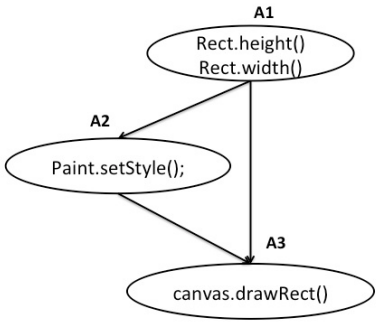

Figure 3. Control flow graph of an Android app.

***S* (iOS corpus):**
S1: CGGeometry.CGRectGetWidth() CGGeometry.CGRectGetHeight() CGGeometry.CGRectMake() CGContext.CGContextFillRect()
S2: CGGeometry.CGRectGetWidth() CGGeometry.CGRectGetHeight() CGGeometry.CGRectMake() CGContxt.CGContextStrokeRect()
***T* (Android corpus):**
T1: Rect.height() Rect.width() Paint.setStyle() Canvas.drawRect()
T2: Rect.height() Rect.width() Canvas.drawRect()
Note: iOS and Android classes are prefixed with `/System/Library/Frameworks/CoreGraphics.framework/CoreGraphics/` and `android/graphics`, respectively. For brevity, we only refer to method names and not their classes.

Figure 4. iOS corpus and Android corpus

traverse the CFG to produce program paths as explained in Section 4.

Inference engine. We use MCCA as our inference engine. We adapted MCCA in order to work with the dataset of API methods and modified the source code [3] in many ways. We mention here the major modifications we did: (1) We changed *edit-distance* function to compare only the method names instead of the entire method signatures that form the *words* in our corpus. (2) We modified the tool so as to output a list of top 10 mappings for each iOS API method. Instead of emitting a single mapping with the highest edge weight, we find top 10 edges for the same source API method in decreasing order of the edge weights. The mappings associated with such top 10 edges are given as output by our tool DDR.

Evaluation. We collected 13 Android apps and 13 iOS apps and ran DDR on this set of apps. There are 3,414 unique iOS API methods and 2,229 unique Android API methods invoked by the respective apps. Since there is no ground truth available to verify the output, we are in the process of designing an evaluation plan to evaluate the mappings and also evaluate DDR on a bigger set of apps.

6. Conclusion

We have described a novel technique to infer mappings between APIs of two mobile platforms. At the heart of our technique lies a method in the NLP domain that learns a translation dictionary from non-parallel corpora of two natural languages. We believe that the same technique works well for the case of mobile platform APIs. Our work to devise an evaluation plan of the resultant mappings is in progress. We see the proposed approach a promising one to infer mappings between two APIs.

References (URLs verified on October 6, 2014)

[1] Clutch: High-speed ios decryption system. https://github.com/KJCracks/Clutch.

[2] Dare: Dalvik retargeting. http://siis.cse.psu.edu/dare/.

[3] MCCA software. http://cs.stanford.edu/~pliang/software/unsuplex.zip.

[4] Networkx graph library. https://networkx.github.io/.

[5] Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[6] F. R. Bach and M. I. Jordan. A probabilistic interpretation of canonical correlation analysis. Technical report, University of California, Berkeley, 2005.

[7] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. NDSS, 2011.

[8] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. ICSE, 2013.

[9] A. Haghighi, P. Liang, T. Berg-kirkpatrick, and D. Klein. Learning bilingual lexicons from monolingual corpora. ACL, 2008.

[10] F. Kschischang, B. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Inf. Theory*, 47 (2), 2001.

[11] Qt. Qt API mapping for iOS developers. http://www.developer.nokia.com/Develop/Porting/API_Mapping.

[12] M. Weiser. Program slicing. ICSE, 1981.

[13] Windows. Windows phone interoperability: Windows phone API mapping. http://windowsphone.interoperabilitybridges.com/porting.

[14] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium*, 2003.