2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Statistical Migration of API Usages

Hung Dang Phan, Anh Tuan Nguyen, Trong Duc Nguyen
Electrical and Computer Engineering Department
Iowa State University
Email: {hungphd,anhnt,trong}@iastate.edu

Tien N. Nguyen
Computer Science Department
University of Texas at Dallas
Email: tien.n.nguyen@utdallas.edu

***Abstract*—To support code migration, we introduce JV2CS, a tool to generate a sequence of C# API elements and related control units that are needed to migrate a given Java code fragment. First, we mine the mappings between single APIs in Java and C#. To overcome the lexical mismatch between the names of Java and C# APIs, we represent an API by its usages instead of its name. To characterize an API with its context consisting of surrounding APIs in its usages, we take advantage of Word2Vec model to project the APIs of Java JDK and C# .NET into corresponding continuous vector spaces. The transformation matrix between the two vector spaces is learned from a small set of human-written pairs of mappings. We use the transformation matrix to derive other mappings, and then use the mappings to generate the corresponding API sequence in C# via a phrase-based translation model. The video demo for JV2CS can be found at https://www.youtube.com/watch?v=MjTfmr9AmR8&feature=youtu.be.**

***Keywords*-Word2Vec; API embedding; API usage migration**

I. Introduction

In modern software development, software vendors often want to develop a software product for multiple operating platforms in different languages. For example, the same mobile app could be developed for iOS (in Objective-C), Android (in Java), and Windows Phone (in C#). To achieve that business need, software companies could originally develop software in one language and then migrate them to another language.

Different languages require developers to use different frameworks and software libraries. For example, in Java, Java Development Kit library (JDK) is a popular toolkit, while .NET is the main framework used in C# software development. Language migration requires not only the mappings between the language constructs (*e.g.,* statements, expressions), but also the mappings between the APIs of the libraries/frameworks used in two languages. For example, to traverse an ArrayList data structure, developers use the JDK APIs ArrayList.iterator() (to get the iterator first), Iterator.hasNext() (to check the existence of the next element), and then Iterator.next() (to obtain that element). The *same functions* can be achieved in C# .NET with the APIs List.GetEnumerator(), IEnumerator.MoveNext(), and IEnumerator.Current, respectively. Some other API usages might need control statements such as for, while, if, etc. Such mappings are called *API mappings* between two languages. Note that the mappings could be 1-to-1 or even many-to-many.

Due to a large number of mappings for APIs, a manual process of defining the migration rules for APIs is tedious and error-prone [1]. To reduce such manual effort, several approaches have been introduced to automatically *mine API mappings* from the corpus of the libraries' client code that already had two respective versions in the two languages [1], [2], [3].

Despite their successes, those mining tools are limited to discovering the mappings that existed in the training corpus. Those tools mine frequently occurring API usages (often called *API usage patterns*) that correspond to one another in the two languages. However, developers often have to migrate arbitrary code with multiple API usages that might be intermixed with one another. For example, a programming task requires to open a file, read the contents line-by-line into a list of strings, perform string operations on each element, and write the resulting lines to a different file. The codes for three sub-usages (opening a file for reading, creating a file for writing, and processing line-by-line from both files with string operations) interleave with one another. Thus, as a whole, the desired API usage might not exactly match with those API usage patterns discovered in the training corpus. As a consequence, an automated mining tool for API mappings would not be able to produce a potentially new API usage in the target language.

In this work, we aim to help developers in code migration process with a tool named JV2CS that takes a given Java code fragment containing API usage(s), and statistically translates it into a new API usage with a sequence of the C# APIs that are likely to be needed by developers in migrating the given code. JV2CS works in two phases. In the first phase, it learns from code corpora the single mappings for APIs in Java and C#. In the second phase, it uses the learned single API mappings to translate a given API usage in Java into the respective one in C# using a statistical machine translation (SMT) tool.

II. Related Work

To support code migration, several approaches have been proposed to mine API mappings. MAM [1] uses API Transformation Graphs, and compares APIs via similar names and calling structures. Aura [4] uses call dependency and text similarity to identify change rules. Rosetta [2] needs pairs of functionally-equivalent applications. The above approaches are limited to mine only 1-to-1 API mappings. StaMiner [3] mines API mappings by maximizing the likelihoods of observing the mappings between API pairs from a parallel corpus of client code. It can mine many-to-many API mappings. However, it is a mining approach and cannot generate a C# API usage sequence for a given Java code. In general, none of them can generate new usages. Phrase-based SMT was used to migrate from Java to C# [5], but does not work well for API usages.

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.17

a) A usage in Java

```
1 HashMap dict = new HashMap();
2 dict.put("A", 1);
3 FileWriter writer = new FileWriter("Vocabulary.txt");
4 for (String vocab: dict.keySet()) {
5     writer.append(vocab + " " + dict.get(vocab)+"\r\n");
6 }
7 writer.close();
```

b) The corresponding usage in C#

```
1 Dictionary myVocabIdxDict = new Dictionary();
2 myVocabIdxDict.Add("A", 1);
3 StreamWriter writer = new StreamWriter("Vocabulary.txt");
4 foreach(string vocab in myVocabIdxDict.Keys) {
5     int idx;
6     myVocabIdxDict.TryGetValue(vocab, out idx);
7     writer.WriteLine(vocab + " " + idx);
8 }
9 writer.Close();
```

Fig. 1: Corresponding API Usages between Java and C# [3]

III. Illustrating Example

Fig. 1 shows an example of corresponding code in Java and C# found on StackOverflow. The code is for the tasks of *reading the data from a vocabulary* of pairs of words (lines 4-6) and indexes after *populating it* (lines 1-2), and *then writing them line by line to a file* (lines 3-7). To do that, developers use the Application Programming Interface elements (*API elements*, APIs for short), which are the *classes*, *methods*, and *fields*. Such a usage with API elements is *used to achieve a programming task* and is called an *API usage*. To migrate the code, one needs to implement a respective API usage in C# that *achieves the same programming task(s)* as the original API usage in Java. If each respective API usage has a single API class or method/field, the mapping is called a (single) *API mapping* (e.g., a class to a class or a method to a method). For example, FileWriter $\leftrightarrow$ StreamWriter, HashMap.put $\leftrightarrow$ Dictionary.add, etc.

Although the entire code in Fig. 1a) might not appear exactly elsewhere, the sub-usages for the tasks of reading the content of a HashMap (lines 1, 4–5), or writing to a new file (lines 3, 4–7) could occur frequently in other projects due to the intention of the designers of the software library. Each API element has a specific role in a usage and its relations to other elements are always well-defined. For example, HashMap.keySet is used first to get the key set and then HashMap.get is applied to each key to obtain the element. With such well-defined functions/roles, an API element regularly occurs with the surrounding API elements, thus, its well-defined relations to those APIs *repeat* in several usages. Therefore, we aim to have a representation that is capable of characterizing an API via its surrounding APIs (rather using name similarity), and capturing the *co-occurring among APIs in the usages* (*e.g.,* getting the key set and then obtaining the element).

IV. Approach

A. Overview

Our tool is based on the following key ideas. First, we characterize an API element by its usage(s) in the context(s) of surrounding, co-occurring APIs (rather than by its names). Let us use the word *usage relations* to denote co-occurring relations among APIs in API usages. For example, each of the three APIs ArrayList.Iterator(), Iterator.hasNext(), and Iterator.next() has its role in an API usage involving a list traversal as explained. We do not aim to detect the role of each API. Instead, we aim to learn the usage relations with a model that maximizes the likelihood of observing a certain API given the surrounding context consisting of other API elements in an usage.

Second, despite that the respective APIs in C# might have different names, since they can be used to achieve the same functionality, each of them would have the same/similar role in the corresponding C# code. For example, List.GetEnumerator() is for obtaining an iterator; IEnumerator.MoveNext() is for checking; and IEnumerator.Current is for retrieving the next element. Thus, we rely on such similar structures in the roles of APIs to derive API mappings. For example, the usage relation (checking before retrieving the next elelement) between Iterator.hasNext() and Iterator.next() is the same as the corresponding APIs in C# IEnumerator.MoveNext() and IEnumerator.Current. If we can learn the usage relations among APIs (i.e., characterizing an API via its surrounding APIs), when we know some of the corresponding APIs in the two languages (e.g., Iterator.hasNext() and IEnumerator.MoveNext()), we could train a model to derive other mappings based on its relations with other APIs.

B. Word2Vec Vector Representation

The API elements in API usages have high repetiveness/regularities because they have to be used in specific ways. This gave us a suggestion to represent API elements in usages with Word2Vec [7], an advanced technology in natural language processing (NLP). Mikolov *et al.* [8] introduces Word2Vec to efficiently learn the vector representations of words in a continuous space from large amounts of text data. The key goal of Word2Vec is to represent a word by learning how it is used from its surrounding words. They introduce two Word2Vec models, named Continuous Bag-of-Words (CBOW) and Skip-gram models. Details can be found in [7].

C. Using Word2Vec for API Usages

It has been shown in NLP that CBOW is able to learn to represent a word by the usage of the word in its surrounding words. The model can characterize a word via its surrounding context consisting of the words right before and after itself. Such characterization has two folds. First, the words being used in a similar context tend to be mapped into nearby locations along some dimension(s) in the projected continuous space [9]. We expect that relevant APIs that are used in similar usage contexts of surrounding APIs will be projected into the locations in the vector space that are closer than the vectors for other API elements with less similar contexts. We define that *two APIs with similar usage contexts as having similar sets of surrounding API elements in their API usages*.

Second, the regularities are observed as constant vector offsets between pairs of words sharing a particular relationship [7]. Via the visualization with PCA [10] and vector computation,

one can observe the following syntactic relations: base/comparative, base/superlative, singular/plural, base/past tense, etc. [8]. For example, $V(better) - V(good) \approx V(faster) - V(fast)$, where V is Word2Vec and the minus sign denotes vector substraction operation. Several types of semantic relationships are observed: country-capital city, city-state, famous person's name-profession, company's name-famous products [8]. For example, for (state,capital): $VFrance) - V(Paris) \approx V(Italy) - V(Rome)$.

For API usages, APIs are often used in certain orders with several semantic dependencies and relations among them. For example, in Fig. 1a), line 3, the return value of FileWriter.new is assigned to the variable declaration FileWriter#var. FileWriter.new must be used to instantiate the object before we can call FileWriter.append. FileWriter.close is then used to close the file. The call to Dictionary.get is used as a parameter for FileWriter.append. As another example of the semantic relations on a data structure, an Iterator can be obtained from a list via ArrayList.iterator and then used to traverse the list with Iterator.hasNext and Iterator.next in a while loop. Those relations among APIs are parts of API usages and occur regularly in source code. We expect to observe such usage relations among APIs via vector offsetting as in NLP. For example, the relation "*avoid adding duplicate elements to a collection*" is expected to be captured via: V(Set.contains) - V(Set.add) $\approx$ V(Map.containsKey) - V(Map.put).

V. Tool Implementation

A. Building Distributed Vectors

Implementation speaking, we process a large Java code corpus to build the sentences for the CBOW Word2Vec model. For each method in a training Java project, we parse the code and have as much type resolution as possible. We then traverse the AST and collect the API elements (classes and method calls), along with the types of their parameters, and the control units used in the usages (while, for, if, etc.). Each method is considered as a sentence consisting of a sequence of API elements, types, and control units. All the sentences for Java projects are used to train Word2Vec model to build the vectors for Java APIs. Such vectors are called API2VEC or API embeddings [6]. For example, in Fig. 1a), we build the sequence:

```
HashMap.var HashMap.new
HashMap.put
FileWriter.var FileWriter.new String
for String.var HashMap.keySet
FileWriter.append String HashMap.get String
FileWriter.close
```

The entire sequence is used for training the CBOW model. Let us assume that the current API is HashMap.keySet, which is used for the output layer. If the context window n = 4, we use for the input layer 4 elements before it (i.e., FileWriter.new, String, for, String.var) and 4 elements after it (i.e., FileWriter.append String HashMap.get String). For the training process, each element in every sentence is considered as the current one. Details on training are given in [7]. After training, the output of the hidden layer gives us the API2VEC vector for the current API. The vector representations for .NET APIs in C# are constructed in the same manner from a corpus of C# code.

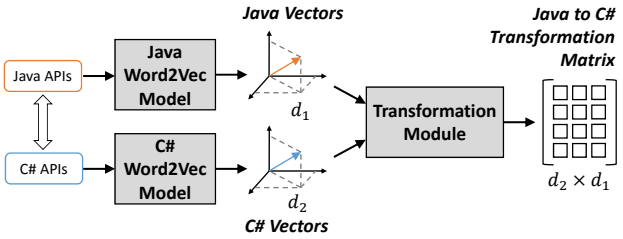

Fig. 2: Training for Transformation Model [6]

B. Transformation Module

After building the API2VEC vectors for all the APIs in the training data, we learn the mappings between the two vector spaces representing the API elements in Java and those in C#.

We develop a mapping technique, API2API [6] that learns the transformation between the two vector spaces for the APIs in the two languages as follows. Fig. 2 shows how we train the transformation model. First, we collect the single mappings between JDK in Java and .NET in C# (in the current implementation, we used a collection of API mappings that was provided as part of a code migration tool, Java2CSharp [11]). For example, FileReader in JDK is mapped to StreamReader in .NET. We then use the trained Word2Vec models for JDK and .NET to build the vectors for each pair of APIs. The pairs of vectors of the respective APIs are used to derive the transformation matrix from Java to C# as follows.

Transformation Matrix. Let us have a set of API pairs and their associated vector representations $\{j_i, c_i\}, i = 1..n$ where j_i is a vector in the Java vector space with d_1 dimensions and c_i is the corresponding vector in the C# vector space with d_2 dimensions. We need to find a transformation matrix T such that $T \times j_i$ approximates c_i. We learn the matrix T with the dimensions $d_2 \times d_1$ by minizing the Least Square Errors [12]:

$$\min_W \sum_{i=1}^{n} ||T \times j_i - c_i||^2$$

The training process is done with stochastic gradient descent. To avoid overfitting, we need to have the number of training pairs equal to or higher than the numbers of dimensions of the vector spaces (in our experiment, the optimal number of dimensions is 200–300).

For prediction, for a given API in Java j, we compute $c = T \times j$. The API in C# whose vector is closest to c via cosine similarity will be the top result. Candidates are ranked using the cosine similarity measures. For all JDK APIs in its vocabulary, we use the computed matrix to compute their corresponding single mappings in .NET in C#. That is, we have $\{j_i, c_i\}, i = 1..|V|$ with V is the vocabulary of JDK APIs.

Note that we need only a small training set of single API mappings (Fig. 2), which helps us to train the transformation model to learn the transformation matrix. The rationale is that the corresponding APIs in JDK and .NET are projected into the locations with similar geometric arrangements in the vector spaces. The transformation matrix enables us to obtain the mappings for all the JDK APIs in the vocabulary.

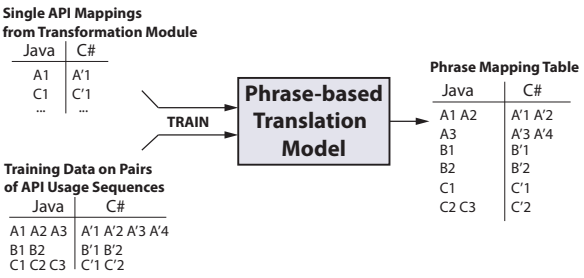

Fig. 3: Training for Phrase-based Translation Model

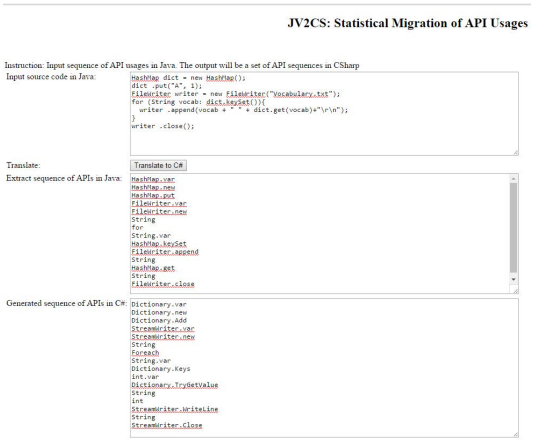

Fig. 4: API Usage Migration with JV2CS

C. API Usage Migration with JV2CS

After using the transformation matrix to obtain all the single mappings for all the APIs in Java and .NET in the vocabularies, we use those mappings from API2API to translate a given API usage from Java code into the corresponding API usage in C#. For this, we treat the sequences of API usages as sentences and use the phrase-based, statistical machine translation tool Phrasal [13] with our mined single API mappings produced by API2API in the first phase. Fig. 3 shows the input and output for the training process for Phrasal. The input is two training datasets. The first one is the set of single API mappings. The second one is the set of mappings between the API sequences representing API usages in the two languages. To prepare for the second dataset, we reuse the dataset in our existing work (Table 2 in [3]) in which we have the mappings of 34,628 Java methods and the respective methods in C#. We parse each pair of methods and build the API sequences. The pairs of corresponding sequences for API usages in both languages are used as the second input to train Phrasal translation model.

Given a Java code fragment, we first build the API sequence. The Phrasal model processes the source sentence *s* from left to right and performs translation. The probability for a candidate sentence is gradually computed along the translation process. The sentence, *i.e.,* the API sequence in C# with the highest probability is presented (Fig. 4). Details on phrase-based statistical machine translation can be found in [14].

We conducted an experiment with the aforementioned dataset of 9 projects with 34,628 pairs of respective methods in Java and C#. We used API sequences in the methods of a project for testing and those in the remaining 8 projects for training. We repeated the process for each project, and compared the result against the real sequences in the manually-migrated C# code in the dataset. We computed precision and recall of translated sequences while considering the orders of APIs. We computed the longest common subsequence (LCS) of a resulting sequence and its reference sequence in the oracle. Precision and recall are computed as: $Precision = \frac{|LCS|}{|Result|}$, $Recall = \frac{|LCS|}{|Reference|}$. They are accumulatively computed for all selected sequences. Our result shows that for a given sequence of Java APIs, the *first resulting .NET API sequence covers from 7.6–8.9 out of 10 needed .NET APIs, and from 6.6–8.6 out of 10 generated .NET API elements are in the correct order in that API sequence*.

VI. Conclusion

We introduce JV2CS, a tool to generate a sequence of C# API elements and related control units that are needed for the migration of a given Java code fragment. To characterize an API with its context consisting of surrounding APIs in its usages, we take advantage of Word2Vec model to project the APIs of Java JDK and C# .NET into corresponding continuous vector spaces. We then used learned single API mappings to translate using a phrase-based machine translation tool.

VII. Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

References

[1] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," in ICSE'10, ACM. 2010.
[2] A. Gokhale, V. Ganapathy, and Y. Padmanaban, "Inferring likely mappings between APIs," in ICSE'13, IEEE CS, 2013.
[3] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical Learning Approach for Mining API Usage Mappings for Code Migration," in ASE'14, ACM. 2014.
[4] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: A hybrid approach to identify framework evolution," in ICSE'10. ACM, 2010.
[5] A.T. Nguyen, T.T. Nguyen, and T.N. Nguyen, "Divide-and-Conquer Approach for Multi-phase Statistical Migration for Source Code", in ASE'15. IEEE CS, 2015.
[6] T.D. Nguyen, A.T. Nguyen, H.D. Phan, T.N. Nguyen, "Exploring API Embedding for API Usages and Applications", in ICSE'17. IEEE, 2017.
[7] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in NIPS'13, pp. 3111–3119.
[8] T. Mikolov, W. T. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in NAACL-HLT-2013. Association for Computational Linguistics, 2013.
[9] L. Deng and D. Yu, "Deep learning: Methods and applications," *Found. Trends Signal Process.*, vol. 7, pp. 197–387, Jun. 2014.
[10] I. Jolliffe, *Principal Component Analysis*. USA: Springer-Verlag, 1986.
[11] "Java2CSharp," http://sourceforge.net/projects/j2cstranslator/.
[12] T. Mikolov, Q. V. Le, and I. Sutskever, "Exploiting similarities among languages for machine translation." *CoRR*, vol. abs/1309.4168, 2013.
[13] D. Cer, M. Galley, D. Jurafsky, and C. D. Manning, "Phrasal: A toolkit for statistical machine translation with facilities for extraction and incorporation of arbitrary model features," in HLT-DEMO '10.
[14] P. Koehn, *Statistical Machine Translation*, 1st ed. New York, NY, USA: Cambridge University Press, 2010.