Inferring Likely Mappings Between APIs using NLP

***Abstract*—API mapping across different platforms/ languages is highly desirable. Among other reasons, they promote machine handled migration from one API to another. This is very relevant in current context as developers increasingly release different versions of their application either to address a business requirement or to survive in competing market. Given a typical platform(/language) expose a large number of API's for developers to reuse, manually writing these mappings is prohibitively resource intensive and may result in manual error. In this paper we present a promising approach to automatically infer such mappings. In particular, unlike existing approaches in literature that rely on existence of manually ported (or at least functionally similar) software across source and target API's, we use textual description present in API documents to mappings. Furthermore, we also compare our results with these approaches and provide relevant discussions how our approach compliments them.**

I. Introduction

=¿ Discuss motivations of the problem

=¿ Discuss existing approaches to this problem and their limitations (e.g., "keyword-based filtering or rule-based approach") [in the current writing, you never explicitly talk about what these existing approaches are like]

=¿ Discuss challenges faced by these existing approaches (here you don't need to include those sentences in the end as in the current writing "To address this challenge..."

=¿ Start with your paragraph with "To address these challenges along with general challenges of applying NLP techniques on software related artifacdts [21], we propose ICON: ...."

=¿ Elaborate the techniques in your approach. For those techniques that address a specific challenge of the two challenges, state something like "To address the first challenge, we propose ...." Note that you should discuss other techniques in your approach besides those techniques for addressing the two challenges.

=¿ Contribution list

With ever increasing computing platforms finding there way to consumers, software developers increasingly release different versions of their application either to address a business requirement or to survive in competing market.

For example, a mobile software developers often release their applications for all the popular mobile platforms such as Android, iOS, and Windows. In context of desktop software, many well-known projects such as JUnit, Hibernate provide multiple versions in different languages, in an attempt to lure developer community to use these libraries across different languages.

Manually migrating a software from one platform(/language) to another is prohibitively time consuming and may be error prone. Recent researches partially alleviate the issue. As they require a programmer to manually describe how Application Programming Interfaces (APIs) of a source platform(/language) maps to API's of the target platform(/language).

Given a typical platform(/language) expose a large number of API's for developers to reuse, manually writing these mappings is prohibitively resource intensive and may result in manual error. Thus the aforementioned researches and tools typically support a limited subset of API's.

In the literature, there exists approaches that address the problem finding mapping between API's, leveraging static [27] and dynamic [8] analysis. However, these approach rely existence of manually ported (or at least functionally similar) software across source and target API's.

However, for two arbitrary libraries, there may not be implementations of the same project in these two libraries, and the APIs used in such projects may be limited. What if such mapping software pairs are not available. Moreover, does existence of such pair guarantee coverage of all API elements? More detail's about pro- and cons in related work section.

To address these shortcomings we propose a different approach for inferring likely mapping between API's. In particular, we propose the use of Natural Language processing of API documents to infer such mapping.

API documentation provides developers with useful information about class/interface hierarchies within the software. Additionally, API documents also provides information about how to use a particular method within a class by means of method descriptions. Method descriptions typically describe specifications in terms of the expectations of the method arguments and functionality of method in general.

We plan to leverage these method descriptions to find the likely mapping. However, it is not trivial to use natural langauge processing on API documents to infer likely mappings. In particular, our technique addresses the following technical challenges to infer likely mappings:

1) *Programming Keywords*. Method descriptions often contain programming keywords (e.g., `true`, `null`, `buffer`), which have a different meaning in the context of programs, in contrast to general linguistics. For instance, *"This method also returns false if path is null"*. In this sentence, words 'false' and 'null' are nouns in the context of object-oriented languages such

as Java and C#, whereas in general linguistics these words are adjectives. Thus, these keywords need to be handled differently.

2) *Semantic Equivalence*. A legal usage in natural language can be described in different words and semantic structures. For instance, consider the following two fragments that describe the similar functionality specification: *"Flushes this data output stream. This forces any buffered output bytes to be written out to the stream."* and *"Writes buffered data to the target stream and calls the flush method of the target stream."*. Thus, there is a need to identify the semantic equivalence of legal usage described in different ways. To address this challenge, we propose a new technique called *equivalence analysis* based on identified grammatical relationships (main nouns and verbs) of a sentence.
3) *Partial Functionality*. There are significant number of cases when the functionality in a source API method is broken across multiple methods in target API. For instance `Graphics.drawChar()` method in J2ME API is mapped to methods `Paint.setColor();Canvas.drawText()` in Android API.
   To address this challenge, we propose a lightweight `Type` analysis.

In summary, our approach leverages natural language description of API's to infer likely mapping thus facilitating cross API migration of applications. As our approach analyzes API documents in natural language, it can be reused independent of the programming language of the library. To the best of our knowledge, ours is the first approach that analyzes natural language API descriptions to infer likely mapping across API's.

Our paper makes the following major contributions:

- A technique that effectively infers mapping across source and target API.
- A prototype implementation of our approach based on extending the Stanford Parser [10], [22], which is a natural language parser to derive the grammatical structure of sentences. An open source implementation of our prototype can be found at our website[1].
- An evaluation of our approach on J2ME and Android API

The rest of the paper is organized as follows. Section II presents the background on code contracts as well as NLP. Section III presents an real world examples that motivate our approach. Section IV presents our approach. Section V presents evaluation of our approach. Section VI presents a brief discussion and future work. Section VII discusses related work. Finally, Section VIII concludes.

[1]http://research.csc.ncsu.edu/ase/projects/pint/

II. Background

A. Topic Modelling

Latent Drichilet Allocation (LDA) [1], [17] is an Information Retrieval (IR) model to fit a probabilistic model on the term occurrences in a corpus of documents. Given a corpus of n documents, a dictionary is created. The dictionary consists a list of all the unique terms (m) occurring in all n documents. Next, a term-to-document matrix M of size (m X n) is generated, where each row represents all the terms occurring the corpus and columns representing the all the documents in the corpus. Each cell of M ($M[i,j]$) contains the weight of the i^{th} term in m in the j^{th} document in n. In the simplest implementation `tf-idf` is used to calculate the weight of a term in a document. The `tf-idf` weight increases proportionally to frequency of the occurrence of a term in a document, however the weight is also offset by the frequency of occurrence the term in the corpus.

Next LDA transforms term-to-document matrix M into a topic-to-document matrix θ of size (k X n), where each row represents the topics occurring in the corpus and columns representing the all the documents in the corpus. This transformation is achieved by identifying latent variables (topics) in the documents. The parameter k is independently provided as an argument to the LDA algorithm. Each cell of θ ($\theta[i,j]$) contains the probability of the i^{th} topic in k occurring in the j^{th} document in n. Since $k << m$, LDA is a mapping of the documents from the term space m to the topic space k [17].

1) k, the total number of topics to be extracted from a corpus of documents.
2) n, the total number of gibbs iteration, where a single iteration involves a gibbs sampler sampling a topic for each term occurring in the corpus.
3) α, this parameter affects the distribution of topics across documents. A high α means that each document is likely to contain a mixture of most of the topics. Conversely, a low α means each document is likely to contain mixture of fewer or one topic.
4) β, this parameter affects the distribution of terms in each topic. A high β means that each topic is likely to contain a uniform distribution of terms. Conversely, a low β means terms are not uniformly distributed across topics.

Although, well suited for human communication, converting natural language into unambiguous specifications that can be processed and understood by computers is very difficult. Recently, a lot of exciting work has been carried out in the area of Natural Language Processing (NLP), with existing NLP techniques proving to be fairly accurate in highlighting grammatical structure of a natural language sentence. However, existing NLP techniques are still in the

processing phase and not in understanding phase. We briefly introduce the NLP techniques used in this work.

Parts Of Speech (POS) tagging [11], [12]. Also known as *'word tagging'*, *'grammatical tagging'* and *'word-sense disambiguation'*, these techniques aim to identify the part of speech (such as noun, verbs, etc.), a particular word in a sentence belongs to. The most commonly used technique is to train a classification parser over a previously known data set. Current state of the art approaches been have demonstrated to achieve 97% [22] accuracy in classifying POS tags for well written news articles.

Phrase and clause parsing. Also known as chunking, this technique divides a sentence into a constituent set of words (or phrases) that logically belong together (such as a Noun Phrase and Verb Phrase). Chunking thus further enhances the syntax of a sentence on top of POS tagging. Current state-of-the-art approaches can achieve around 90% [22] accuracy in classifying phrases and clauses over well written news articles.

Typed Dependencies [2], [3]. The Stanford typed dependencies representation is designed to provide a simple description of grammatical relationships directed towards non-linguistics experts to perform NLP related tasks. It provides a hierarchical structure for the dependencies with precise definitions of what each dependency means, thus facilitating machine based manipulation of natural language text.

Named Entity Recognition [6]. Also known as *'entity identification'* and *'entity extraction'*, these techniques are a subtask of IE that aims to classify words in a sentence into predefined categories such as names, quantities, expression of times, etc. These techniques help in associating predefined semantic meaning to a word or a group of words (phrase), thus facilitating semantic processing of named entities.

Co-reference Resolution [13], [18]. Also known as *'anaphora resolution'*, these techniques aim to identify multiple expressions present across (or within) the sentences, that point out to the same thing or *'referant'*. These techniques are useful for extracting information; especially if the information encompasses many sentences in a document.

III. Examples

We now present a series progressing in complexity to demonstrate our approach. Our examples are from real Java Platform, Micro Edition (JME) formerly known as J2ME and Android API. Both JME and Android use Java as the language of implementation and targeted towards hand-held devices. However, our approach is independent of language of implementation and thus can be used for any similar source and target API.

First we parse the API documents of source and target API library and store it in intermediate representation for analysis. In particular, we extract class, interface and corresponding method descriptions from API documents. Second, we then create a term vector representation of each class description and method descriptions. Vector space model or term vector model is an algebraic model for representing text documents (and any objects, in general) as vectors of identifiers, such as, for example, index terms. In our case, each word is considered as a term barring the stop words such as (a, the, and ...).

Finally we use the term vector representation of a method description in a source API to query term vector representation of the the method description in target API. The results are ranked based on the Term Frequency-Inverse Document Frequency (TF-IDF) measure. TF-IDF is a numerical statistic which reflects how important a word is to a document in a collection or corpus. It is often used as a weighting factor in information retrieval and text mining.

We then collect the top 10 results from target API methods for each source API method. Figure **??** and Figure **??** shows example of trivially similar API methods. Our search results return the Android API method shown in Figure **??** as a first match for the JME method shown in Figure **??**. Notice the slightly different description for the methods in JME and Android API. Although, correct this is very trivial mapping. We next present a more complex example.

Figure **??** and Figure **??** shows example of non-trivially similar API methods. Although, the textual description is similar using text analytics (TF-IDF similarity using term frequency vector), there is difference in type of arguments.

Our approach first tries to align the arguments:

1) int is matched to float common knowledge
2) `Type` analysis for `Paint` is performed to get additional (previous) method invocations
3) `Type` analysis of `Anchor` is performed to get additional method invocations

IV. Approach

TBD

V. Evaluation

We conducted an evaluation to assess the effectiveness of our approach. In our evaluation, we address three main research questions:

- **RQ1**: What are the precision and recall of our approach in identifying API mappings?
- **RQ2**: How do the mappings inferred by our approach compare with the human written mappings?

A. Subjects

We used the API documents of the following two libraries as subjects for our evaluation.

J2ME. Java Platform, Micro Edition, or Java ME, is a Java platform designed for embedded systems (mobile devices are one kind of such systems). Target devices range from industrial controls to mobile phones (especially feature

phones) and set-top boxes. Java ME was formerly known as Java 2 Platform, Micro Edition (J2ME).

Android.Android is a Linux-based operating system designed primarily for touchscreen mobile devices such as smartphones and tablet computers.

VI. Discussion and Future work

VII. Related work

Language migration has been an active area of research [9], [15], [23], [24], with myriad techniques that have been proposed over time to achieve automation. However, most of these approaches focus on syntactical and structural differences across languages. For instance, Deursen et al. [23] proposed an approach to automatically infer objects in legacy code to effectively deal with differences between object-oriented and procedural languages.

However, El-Ramly et al. [5]'s experience points out that most of these approaches support only a subset of API's for migration. Another recently published survey by Robillard et al. [19] provides a detailed overview of techniques dealing with mining API mappings.

Among other works described in [19], Mining API Mapping (MAM) [27] is most directly related to our work. MAM mines API mapping relations across different languages for language migration, however there is a significant difference between the approach undertaken to achieve it, thus each techniques have its own pros and cons.

While MOM relies on existence of software that has been ported manually from a source to target API, our approach has no such requirement. MOM then applies a technique called "method alignment" that pairs the methods with similar functionality across implementation. These methods are then statically analyzed to detect mappings between source and target API. In contrast, our approach relies on simple text analytics and natural language processing of source and target API document to achieve the same. While our approach also uses simple static analysis, our techniques are relatively very lightweight. Furthermore, we demonstrate that our approach also infers equally good mappings if not better.

Gokhle et al. [8] is another work that is closely related to our approach. They improve upon the MOM project by removing the restriction of having software that has been ported manually from a source to target API. They however still require software that have similar functionality (if not exactly same) that use source and target API. In contrast our approach is independent of such requirement.

Zheng at al. [26] mine search results of Web search engines such as google to recommend related APIs of different libraries. In particular, they propose heuristics to formulate keywords using the name of the method in the source API, and the name of target API to query web search engine. **TODO** to see if their techniques can be adapted.

NLP NLP techniques are increasingly applied in the software engineering domain. NLP techniques have been shown to be useful in requirements engineering [7], [20], [21], usability of API documents [4], [16], and other areas [14], [25], [28], [29]. We next describe most relevant approaches.

Zhong et al. [28] employ NLP and ML techniques to infer resource specifications from API documents. Their approach uses machine learning to automatically classify such rules in a single API library. In contrast, our approach infers API map- ping relations across different languages for language migration, whereas the previous approaches mine API properties of a single language to detect defects or to assist programming.. Furthermore, the performance of the preceding ML-based approaches is dependent on the quality of the training sets used for ML. In contrast, our approach is independent of such training set and thus can be easily extended to target respective problems addressed by these approaches.

VIII. Conclusion

References

[1] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[2] M. C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *Proc. LREC*, 2006.

[3] M. C. de Marneffe and C. D. Manning. The stanford typed dependencies representation. In *Workshop COLING*, 2008.

[4] U. Dekel and J. D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.

[5] M. El-Ramly, R. Eltayeb, and H. Alla. An experiment in automatic conversion of legacy Java programs to C#. In *Proc. IEEE CSA*, pages 1037–1045, 2006.

[6] J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proc, 43nd ACL*, 2005.

[7] V. Gervasi and D. Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transactions Software Engineering Methodologies*, 14:277–330, 2005.

[8] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proc. 35nd ICSE*, 2013.

[9] A. E. Hassan and R. C. Holt. A lightweight approach for migrating web frameworks. *Inf. Softw. Technol.*, 47(8):521–532, Jun 2005.

[10] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proc. 41st ACL*, pages 423–430, 2003.

[11] D. Klein and D. Manning, Christopher. Accurate unlexicalized parsing. In *Proc. 41st Meeting of the Association for Computational Linguistics*, pages 423 – 430, 2003.

[12] D. Klein and D. Manning, Christopher. Fast exact inference with a factored model for natural language parsing. In *Proc. 15th NIPS*, pages 3 – 10, 2003.

[13] H. Lee, Y. Peirsman, A. Chang, N. Chambers, M. Surdeanu, and D. Jurafsky. Stanford's multi-pass sieve coreference resolution system. In *Proc. CoNLL-2011 Shared Task*, 2011.

[14] G. Little and R. C. Miller. Keyword programming in Java. In *Proc. 22nd ASE*, pages 84–93, 2007.

[15] M. Mossienko. Automated Cobol to Java recycling. In *Proc. 7th CSMR*, pages 40–, 2003.

[16] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *Proc. 34th ICSE*, 2012.

[17] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proc. 35th ICSE*, pages 522–531, 2013.

[18] K. Raghunathan, H. Lee, S. Rangarajan, N. Chambers, M. Surdeanu, D. Jurafsky, and C. D. Manning. A multi-pass sieve for coreference resolution. In *Proc. EMNLP*, 2010.

[19] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Trans. on Software Engineering*, 39(5):613–637, 2013.

[20] A. Sinha, A. M. Paradkar, P. Kumanan, and B. Boguraev. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *Proc. DSN*, pages 327–336, 2009.

[21] A. Sinha, S. M. SuttonJr., and A. Paradkar. Text2test: Automated inspection of natural language use cases. In *Proc. ICST*, pages 155–164, 2010.

[22] The Stanford Natural Language Processing Group, 1999. http://nlp.stanford.edu/.

[23] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. 21st ICSE*, pages 246–255, 1999.

[24] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. on Software Engineering*, 14(8):1207–1228, 1988.

[25] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *Proc. 20th ACM SIGSOFT FSE*, pages 12:1–12:11, 2012.

[26] W. Zheng, Q. Zhang, and M. Lyu. Cross-library API recommendation using web search engines. In *Proc. 13th ESEC/FSE*, pages 480–483, 2011.

[27] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proc. 32nd ICSE*, pages 195–204, 2010.

[28] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *Proc. 24th ASE*, pages 307–318, 2009.

[29] H. Zhou, F. Chen, and H. Yang. Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology. In *Proc. 8th QSIC*, pages 225 –234, 2008.