JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2012; **00**:1–15
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Discovering Likely Mappings between APIs using Text Mining

Rahul Pandita[1], Raoul Praful Jetley[2], Sithu D Sudarsan[2], Tim Menzies[1], and Laurie Williams[1]

[1]*North Carolina State University, Raleigh, NC, USA*
[2] *ABB Corporate Research, Bangalore, India*

SUMMARY

Developers often release different versions of their applications to support various platform/programming-language application programming interfaces (APIs). To migrate an application written using one API (source) to another API (target), a developer must know how the methods in the source API map to the methods in the target API. Given a typical platform or language exposes a large number of API methods, manually writing API mappings is prohibitively resource-intensive and may be error prone. Recently, researchers proposed to automate the mapping process by mining API mappings from existing code-bases. However, these approaches require as input a manually ported (or at least functionally similar) code across source and target APIs. To address the shortcoming, this paper proposes TMAP: ***T**ext **M**ining based approach to discover likely **API** mappings* using the similarity in the textual description of the source and target API documents. To evaluate our approach, we used TMAP to discover API mappings for 15 classes across: 1) `Java` and `C#` API, and 2) `Java ME` and `Android` API. We compared the discovered mappings with state-of-the-art source code analysis based approaches: Rosetta and StaMiner. Our results indicate that TMAP on average found relevant mappings for 57% more methods compared to previous approaches. Furthermore, our results also indicate that TMAP on average found exact mappings for 6.5 more methods per class with a maximum of 21 additional exact mappings for a single class as compared to previous approaches. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: API documents, text mining

1. INTRODUCTION

Software is ubiquitous and of late people are increasingly interacting with software applications that run on variety of software platforms on daily basis. To retain existing users (and attract new users) across different platforms, developers are increasingly releasing different versions of their applications. For example, a typical mobile software developer often releases his/her applications on all the popular mobile platforms, such as Android, iOS, and Windows, which often involves rewriting applications in different languages. For instance, Java is preferred language for implementing Android applications and Objective-C for iOS application. In context of desktop software, many well-known projects, such as JUnit and Hibernate provide multiple versions in different programming languages, to attract developer community to use these libraries across those languages.

To assist developers with software migration there are existing language migration tools, such as Java2CSTranslator [8]. However, such tools require a programmer to manually input how methods

*Correspondence to: North Carolina State University, 890 Oval Drive Campus Box 8206, 3228 EB-II, Raleigh, NC 27695-8206. Email: rpandit@ncsu.edu

in a source language's Application Programming Interfaces (API) maps to the methods of the target language's API. Given a typical language (or platform) exposes a large number of API methods for developers to reuse, manually writing these mappings is prohibitively resource intensive and may be error prone.

The goal of this research is to support software developers in migrating an application from a source API to a target API by automatically discovering likely method mappings across APIs using text mining on the natural language API method descriptions.

Existing approaches address the problem of finding method mapping between APIs using static [25] and dynamic [5] analysis. Recently Nguyen et al. [14] further proposed to apply statistical language translation techniques to achieve language migration by mining large corpora of open source software repositories. However, these approaches require as an input manually ported (or at least functionally similar) software across source and target APIs. Since static analysis and mining approaches [14, 25] leverage source code analysis, accuracy of such approaches is dependent on the quality of the code under consideration. Likewise, accuracy of dynamic approaches [5] is dependent on the quality and completeness of test inputs to dynamically execute the API behavior comprehensively.

To address the shortcomings of existing program-analysis based approaches, we propose to use the natural language API method descriptions to discover the method mappings across APIs. Our intuition is: *since the API documents are targeted towards developers, there may be an overlap in the language used to describe similar concepts that can be leveraged.* In general, API documentation provides developers with useful information about class/interface hierarchies within the software. Additionally, API documents also provide information about how to use a particular method within a class by means of method descriptions. A method description typically outlines specifications in terms of the expectations of the method arguments and functionality of method in general.

This paper presents TMAP: An approach that leverages the natural language method descriptions to discover the likely method mapping between APIs. TMAP stands for ***T**ext **M**ining based approach to discover likely **API** method mappings*. In particular, TMAP proposes to create a vector space model [12, 19] of the target API method descriptions. TMAP then queries the vector space model of target API using automatically generated queries from the source API method descriptions. TMAP automates the query generation in source API using the concepts from text mining, such as emphasizing (or omitting) certain keywords over others and querying multiple facets (such as class description, package names, and method description). Since TMAP analyzes API documents in natural language, the proposed approach is reusable, independent of the programming language of the library.

We pose the following research question: *How accurately can the similarity in the language of API method descriptions be leveraged to discover likely API Mappings?* To answer our question, we apply TMAP to discover likely API mappings for 15 classes across: 1) `Java` and `C#` API; 2) `Java ME` [†] and `Android` API. We also compare the discovered mappings with two state-of-the-art static and dynamic analysis based approaches: Rosetta [5] and StaMiner [14].

This paper makes the following major contributions:

- A text mining based approach that effectively discovers mapping between source and target API.
- A prototype implementation of our approach based on extending the Apache Lucene [11]. An open source implementation of our prototype can be found at our website[‡].
- An evaluation of our approach on 5 classes in Java ME to Android API and 10 Classes Java to C# API. The evaluation results and artifacts are publicly available on the project website.

The rest of the paper is organized as follows. Section 2 presents the background on Text Mining. Section 3 presents a real world example that motivates our approach. Section 4 presents TMAP approach. Section 5 presents evaluation of TMAP. Section 6 presents a brief discussion and future work. Section 7 discusses the related work. Finally, Section 8 concludes the paper.

[†] Java Platform Micro Edition
[‡] `https://sites.google.com/a/ncsu.edu/apisim/`

DOI: 10.1002/smr

2. BACKGROUND

Text mining is a broad research area, including but not limited to the techniques facilitating retrieval/manipulation of useful information from a large corpus to text. As opposed to traditional data mining, text mining analyzes free-form text distributed across documents (rather than data localized and maintained within a database). To analyze this data, text mining uses concepts from traditional data analytics, natural language processing, and data modeling. We next introduce the concepts from text mining that have been used in the presented approach.

Indexing [4, 12]: Indexing is the process of extracting text data from source documents and storing them in well-defined indexes. During the indexing process, a document (a sequence of text) is divided into its constituent units, known as tokens or terms, based on a well-defined criterion. A term is typically an individually identifiable unit of the document (such as a word) and relates to the individual terms stored in an index. Once the terms within each document are identified, they are added to the index, with the corresponding link to the document and associated term frequencies. Term frequency is the simple count of the occurrence of a term in a document. An optional pre-processing step further assists with indexing, such as removing stop-words.

Among various indexing strategies, the use of *inverted file indexing* [4] is well suited for large document collections. In the simplest form, an inverted file index provides a mapping of terms, such as words, to its locations in a text document. A document can be thought of as a collection of m terms. Typically a document is made up of a sequence of n unique terms such that $n <= m$. The number n is usually far less than m as most of the terms are repeated while forming a document. For instance, the term "the" is repeated many times in this paper. The set of unique terms within an index forms the "Term List" v of the index. If a pointer (say numeric location) is associated with each term in v to the location of that term in text document, the resultant data structure is a form of inverted file index. As the document collection grows, the number of documents matching a term in the index becomes sparser.

The index is oftentimes annotated with the information regarding the frequency of occurrence of each term in the document. The representation of a document as a vector of frequencies of terms is referred to as *vector space model* [4, 19].

TF-IDF [12]: Term -frequency inverse-document-frequency (`tf-idf`) is a numerical statistic that is intended to capture the importance of a term to a document in a corpus. Often used as a weighting metric in information retrieval and text mining, 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 count of the number of documents that contain the term.

Cosine Similarity [19]: In mathematics, Cosine similarity is a numerical statistic to measure the similarity between two vectors. Cosine similarity is defined as a dot product of magnitude of two vectors. In the context of text mining, the Cosine similarity is used to capture the similarity between two documents represented as term frequency vectors.

3. EXAMPLE

We next present an example to motivate our work and list the considerations for applying text mining techniques on API documents. The example is from the Java ME and the Android API. Both Java ME and Android use Java as the language of implementation and are targeted towards hand-held devices. Figure 1 shows the API method description of `drawString` method from `javax.microedition.lcdui.Graphics` class in Java ME API. Figure 2 shows the API method description of method `drawText` method from `android.graphics.Canvas` class in Android API.

Notice the overlap in the language of these two methods. A developer can easily conclude that the two methods offer similar functionality. However, Android API has more than 23,000 public methods. Manually going through each method description to find similar methods is prohibitively time consuming, supporting the need for automation. A naive solution to automate the task is to perform keyword-based search.

DOI: 10.1002/smr

javax.microedition.lcdui Class Graphics drawString

```
public void drawString(String str,int x,int y,int anchor)
```

Draws the specified String using the current font and color. The x,y position is the position of the anchor point. See anchor points.

Parameters

str - the String to be drawn
x - the x coordinate of the anchor point
y - the y coordinate of the anchor point
anchor - the anchor point for positioning the text

Throws

NullPointerException - if str is null
IllegalArgumentException - if anchor is not a legal value

See Also

`drawChars(char[], int, int, int, int, int)`

Figure 1. `drawString` API method description of `Graphics` class in the Java ME API

android.graphics Class Canvas drawText

```
public void drawText (String text, float x, float y, Paint paint)
```

Draw the text, with origin at (x,y), using the specified paint. The origin is interpreted based on the Align setting in the paint.

Parameters

text - The text to be drawn
x - The x-coordinate of the origin of the text being drawn
y - The y-coordinate of the origin of the text being drawn
paint - The paint used for the text (e.g. color, size, style)

Figure 2. `drawText` API method description of `Canvas` class in the Android API

To demonstrate the difficulties faced by keyword-based search in above example, we searched the Android API description with the keywords listed in Table I using the Apache Lucene [11] framework. Lucene is a high-performance, full-featured text search engine library written entirely in Java. The column "Query" describes the keywords we used to perform keyword-based search, The column "Hits" lists the number of matches found, and The column "Top-10" lists the rank of the first relevant method in top-ten results. For instance, when we searched for the class name "Graphics" in the Android API, we did not get any results. We also did not get any results for when we searched for method name using keywords "drawString".

When we searched the Android API using the words in the method signature as keywords, we got a 23,547 results (almost all methods in Android API). The high number of results is because of the confounding effects of keywords, such as "public". Most of the methods signatures have the keyword "public". Although the ranking mechanisms in Lucene did rearrange the results moving methods with most keyword matches first, we did not find a relevant method in top-ten results. Likewise, the words in method descriptions as a whole or in parts also did not yield better results. In the example, a combination of various attributes, such as method name split in camel case notation "draw String", keywords from both class and method description resulted in the Android API equivalent method `drawText` shown in Figure 2 in the top ten results.

The previous example demonstrates difficulties faced by simple keyword-based searches and text mining in general to discover likely method mappings. The TMAP approach in general addresses the following difficulties in applying text mining approaches on natural language API method descriptions:

1. *Confounding effects*. Certain method names have a confounding effects. For instance, `toString()`, `get()`, `set()` are too generic and tend to have similar descriptions across different method definitions. These generic methods often cause interference with the output

DOI: 10.1002/smr

#	Query	Hits	Top-10
1	Class Name: “Graphics”	0	-
2	Method Name: “drawString”	0	-
3	Method Signature	23547	-
4	Method Description: (Complete)	16820	-
5	Method Description: (summary sentences)	94230	-
6	Combination	1479	**3**

‘-’=No Match in Top-10 results.

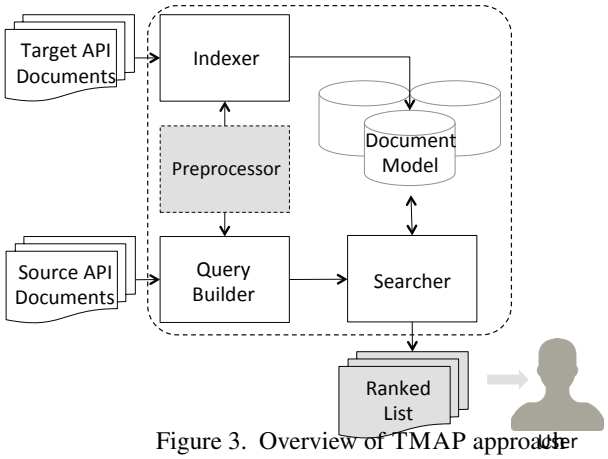

Figure 3. Overview of TMAP approach

of text mining based approaches. The problem is to automatically identify such method names to de-emphasize their importance in a query.

2. *Weights*. Not all terms in a method descriptions are equally important keywords. For instance, the term “zip” in the sentence “opens a zip file” is more important than the terms “opens” and “files” as the term emphasizes on the specific type of file. The problem is to automatically identify the importance of a term.
3. *Structure*. API documents are not flat contiguous text blobs. They have well defined structure, that is often shared by method descriptions. Ignoring the structure may cause ineffective queries negatively affecting the results. The problem is to effectively aggregate the results of querying individual API document elements (such as class description, class names, method names, method descriptions).

4. APPROACH

We next present our approach for discovering likely mappings of API methods across APIs. Figure 3 provides an overview of the TMAP approach. The TMAP approach consists of three major components: 1) an Indexer; 2) Query Builder; and 3) Searcher components.

The Indexer accepts the API documents of the target API and creates indexes (a vector space model) of these documents by extracting intermediate contents from the method descriptions. The Query Builder accepts the API documents of the source API and creates queries to be executed on the indexes (a vector space model). Finally, Searcher component executes the queries on the indexes and generates a ranked list of the API methods from target API documents as mapping results to be presented to the developers for confirmation. We next describe each component in detail.

4.1. Indexer

This component accepts the API method descriptions of the target API and creates indexes (a vector space model) of these documents. In particular, Indexer extracts the following fields from the API method descriptions:

DOI: 10.1002/smr

F1) Type Name: The name of enclosing class/interface of the method. For the method description of `drawString` shown in Figure 1, indexer extracts the type Name as "Graphics".

F2) Package Name: The package name of the enclosing type. For the method description shown in Figure 1, indexer extracts the package name as "javax.microedition.lcdui".

F3) Method Name: The name of the method. For the method description shown in Figure 1, indexer extracts the method name as "drawString".

F4) Class Description: The description of enclosing type. Class description is not shown in Figure 1 for the space considerations.

F5) Method Description: The description of the method. For the method description shown in Figure 1, indexer extracts the method description as all the text except method declaration in Line 1.

This step is required to extract the desired descriptive text from the API method descriptions. Getting structured descriptions facilitates searching on individual categories. This step allows TMAP to deal with the *structure* issue as presented in Section 3. Different API documents may have different styles of presenting information to developers. Such stylistic differences may also include the difference in the level of the details presented to developers. TMAP relies only on basic fields that are generally available for API methods across different presentation styles.

After extracting the desired information, extracted text is further preprocessed. The preprocessing steps are required to make the text amenable to text mining techniques that are used in the subsequent phases of the TMAP approach. In particular, TMAP performs the following basic prepossessing steps:

P1) Presentation Elements: A typical API method description is often interleaved with presentation elements for improved readability. For instance, `JavaDoc` provides a list of identifiers such as `@Code` and `@link`. These identifiers are automatically translated into presentation markup, such as links and fonts. Although such elements are part of the description text, these elements often cause noise in the text mining techniques to compute relevance based on the query. Therefore, this preprocessing step cleans the method descriptions to remove such elements. We use a static list of presentation elements to achieve cleaning in this step with relatively high accuracy.

P2) Split Package Notation: In method descriptions, the "." character is used as a separator character for package names like "`javax.microedition.lcdui`". We use regular expressions to split the package name into constituent words to facilitate search on individual words in the package name. For example, "`javax.microedition.lcdui`" is split into "javax microedition lcdui".

P3) Split CamelCase Notation: API method descriptions are often interleaved with programming identifiers, such as class names and method names. Oftentimes these identifiers use CamelCase notation. The CamelCase notation is the *de facto* mechanism used by programmers for combining phrases into a single word, such that each word in the phrase begins with a capital letter. Previous research [10] demonstrated the benefit of splitting the CamelCase word into its constituent phrase for automated code completion. TMAP splits such identifiers into constituent phrases to better facilitate searching. TMAP leverages the well-formed structure of CamelCase notations to encode a regular expression to achieve splitting with relatively high accuracy. For example, "`drawString`" is split into "draw String".

P4) Lowercase: This step involves converting the text description to lower case. The step is performed to normalize the text for making the keyword match case insensitive, further increasing the range of queries.

P5) Stemming: This steps transforms the words in the description to their base form. Stemming is very effective in extending the range of keyword based queries to match various operational forms of the words. For instance, "has", "have", and "had" are mapped to the stem "ha". We use the default implementation of the Lucene Porter stemmer for pre-processing.

After preprocessing, TMAP next creates indexes for the API method descriptions. An index is collection of documents where each document is made up of values organized into well defined fields. TMAP considers every method description as an individual document and uses the following major fields (as previously described): 1) combination of package and class name; 2)

DOI: 10.1002/smr

java.util Interface Iterator **hasNext**

`boolean hasNext()`

Returns `true` if the iteration has more elements. (In other words, returns true if `next()` would return an element rather than throwing an exception.)

Returns:

`true` if the iteration has more elements

Figure 4. API Method Description of `hasNext` method in `Iterator Interface` from Java API

Type Name: `java util iter`
Type Description: `iter over collect iter take enumer java collect framework`
Method Name: `ha next`
Method Description: `return true iter ha more element other word return true next would return element rather than throw except`

Figure 5. Query based on API method description of `hasNext` method in `Iterator Interface` from Java API

class description; 3) method signature; 4) method name; and 5) method description. The values of these fields are the text after preprocessing. TMAP uses a vector space model representation of the documents for each field. Vector space model or term vector model is an algebraic model for representing text documents (and any objects, in general) as vectors of identifiers and their frequency of occurrence. In case of TMAP, each word is considered as a term except the stop words, such as "a", "the", and "and".

4.2. Query Builder

This component accepts the API method descriptions of the source API and creates queries for method descriptions. These queries are executed on the target API index to retrieve an ordered list of relevant API methods. In particular, Query Builder uses the same preprocessing steps followed by Indexer (listed in Section 4.1). After extracting the desired descriptive text from the API method descriptions, this component systematically creates search queries to search for different fields in Indexes. Keywords for searching in "Type Name", "Type Description", "Method Name", and "Method Description" fields are derived from their equivalents in the extracted descriptive text.

For instance, consider the method description shown in Figure 4 and equivalent query in shown in Figure 5. The Keywords for "Type Name" are derived from preprocessing "java.util.Iterator" resulting in "`java util iter`". Notice the package notation is split into individual words and "Iterator" is further transformed to lower case and its stem "iter". Likewise, keywords for field "Method Name" is derived by preprocessing "hasNext", which is first split into "has Next" and then transformed using stemming into "`ha next`" (*ha* being stem of word *has*).

For generating keywords to query the "Type Description" field we consider following heuristic: *Heuristic H1: the first paragraph or the first five sentences of the type description (whichever is shorter) provides reasonable keywords for searching equivalent class in target API.*

Likewise, for generating the keywords to query the "Method Description" field, we consider the following heuristic: *Heuristic H2: the first paragraph or the first two sentences of the method description (whichever is shorter) provides reasonable keywords for searching equivalent method in target API.*

TMAP uses these heuristics to improve the performance of searching infrastructure that tends to be inversely proportional to the complexity and length of the query. Using all the descriptive text as keywords often results in a verbose query. As the number of keywords in a query increases the effectiveness of the query decreases. A large number of keywords have higher probability of matching large number of documents in comparison to a query with fewer keywords. In contrast, we observed that the document writers tend to describe the general overview of class and method

DOI: 10.1002/smr

description in the first few sentences followed by implementation and design specific details. We thus focused on the words in these overview sentences to create queries instead of using entire descriptive text.

Weights for terms. As mentioned in Section 3, all terms in a method description are not equally important keywords. TMAP further enhances the query by quantifying the importance of a term in the method description and use that as the weight of the corresponding keywords in the query. In particular, we propose to use tf-idf [12] as a means to quantify importance of a term. For each term in the method description TMAP calculates the number of times the term occurs in that method description as $freq_{mtd}$. TMAP also calculates the maximum frequency of any term in the document as $freq_{MAX}$. TMAP then calculates the number of documents in the corpus that contains the term as $freq_{doc}$. TMAP finally calculates the tf-idf score of the term (as listed [12]) as:

$$\texttt{tf-idf} = (0.5 \ + \frac{0.5 \ X \ freq_{mtd}}{freq_{MAX}}) \ * log(1 \ + \ \frac{total_{mtd}}{freq_{doc}})$$

The calculated tf-idf values of terms are normalized to a range of 0 to 1 (both 0 and 1 inclusive) for each document. The normalized tf-idf sore of the top-k terms is then used as the weights for the corresponding keywords occurring in the query.

For the API method description shown in Figure 4, TMAP calculates "iter", "ha" (Lemma of "has"), and "element" as most the important terms with normalized tf-idf scores of 1.0, 1.0, and 0,6 respectively. We augment the query shown in Figure 5 with the computed weights for the keywords respectively.

4.3. Searcher

The searcher component accepts the query from Query Builder component and queries the index generated by Indexer component. The results are then ranked and presented to the end user for review. The searcher is realized as follows. First all the documents that match the keywords and clauses in a query are returned. Then, the returned documents are ranked using the cosine similarity [19] of the terms in query and the terms in returned documents. In mathematics, Cosine similarity is a numerical statistic to measure the similarity between two vectors. In information theory [12], cosine similarity is the standard statistic to rank relevant documents.

4.4. Implementation

We implemented a prototype version of the TMAP approach. We first manually download the HTML version of API documents of libraries. We then implemented a parser for extracting the requisite text from these documents using Jsoup[§], which is a java library for working with HTML documents. In particular our prototype implementation parses: 1) Oracle's Javadoc style; 2) Android style documentation; and 3) Microsoft's MSDN documentation.

We next implemented the indexing, query building, and searching infrastructure using the Apache Lucene [11]. Lucene is a high-performance, full-featured text search engine library written entirely in Java. Our prototype implementation and evaluation subjects are publicly available on the project website[¶].

5. EVALUATION

We conducted an evaluation to assess the effectiveness of TMAP. In our evaluation, we address following research questions:

- **RQ1**: What is the effectiveness of TMAP in leveraging the similarity in the language of API method descriptions to discover likely API Mappings?

[§] http://jsoup.org/
[¶] https://sites.google.com/a/ncsu.edu/apisim/

DOI: 10.1002/smr

- **RQ2**: How do the mappings discovered by TMAP compare with the mappings discovered by existing program-analysis based approaches?

5.1. Subjects

We evaluated TMAP using a snapshot of the publicly available API documents of Java, C#, Android, and Java ME downloaded in January 2015.

Java Platform Micro Edition, or Java ME, is a Java platform designed for embedded systems (such as mobile devices). Target devices range from industrial controls to mobile phones (especially feature phones) and set-top boxes. Android is a linux-based operating system designed primarily for touchscreen mobile devices, such as smartphones and tablet computers.

Java and C# are general-purpose programming languages from Oracle and Microsoft respectively. Java applications are typically compiled to bytecode that run on any Java Virtual Machine (JVM) irrespective of underlying computer architecture. Likewise, C# is compiled into intermediate representation that run on Microsoft's common language infrastructure.

Particularly we used the API documents of the following library pairs as subjects for our evaluation.

Java ME (to Android) API: For our evaluation we considered the methods in the following Java ME `types` as the source API methods to discover mapping methods in Android API: `Alert`, `Canvas`, `Command`, `Graphics`, and `Font` Classes in `javax.microedition.lcdui` package.

The listed `types` provides methods for supporting graphics related functionality in Java ME. Furthermore, Rosetta approach by Gokhale et al. [5] reports the mapping for methods in these `types` along with seven others (twelve `types` in total) as a part of their evaluation, thus allowing a comparison with dynamic-analysis based approaches. Rosetta approach requires a user to manually execute functionally similar applications using source and target API with identical (or near identical) inputs and collect execution traces. Finally Rosetta analyses the collected execution traces to infer method mappings. We focused our evaluation on the listed five types which first three authors perceived as frequently used types among the twelve types reported by Rosetta. In the future, we plan to evaluate TMAP approach on all the twelve reported types.

Java (to C#) API: For our evaluation we considered the methods in the following Java `types` as the source API methods to discover mapping methods in C# API: 1) `File`, `Reader`, and `Writer` in `java.io` package; 2) `Calendar`, `Iterator`, `HashMap`, and `ArrayList` in `java.util` package; and 3) `Connection`, `ResultSet`, and `Statement` classes in `java.sql` package.

The `types` in `java.io` provide the API methods for accessing and manipulating the file system. Types in `java.util` provide API methods for miscellaneous utilities, such as text manipulation, collections frameworks and other data structures. Types in `java.sql` provide the API methods for accessing and processing data stored in databases. We selected these particular packages in Java programming languages because Nguyen et al. [14] in their work (StaMiner) for statistical language migration find mappings for the `types` in these packages. Their mapping results allow comparison of TMAP with static-analysis based approach. Although, Nguyen et al. [14] report on all the classes in these packages, due to the amount of effort, we focused our analysis on the listed types which first three authors perceived as frequently used types in their respective packages.

5.2. Evaluation Setup

We first downloaded the publicly available API documents from the respective websites of the subject APIs. We then cleaned and extracted the desired fields as described in the Section 4.1. We then indexed the extracted text into the Lucene indexes. We created a separate index for every API type: Java ME, Android, Java, and C#.

For every `type` (class/interface) under consideration (as listed in Section 5.1), we extracted the publicly listed methods from API documents. For a given `type`, we only consider the methods that are listed in the public API. We only consider the methods explicitly declared or overridden by a type and ignore the inherited methods. We then use TMAP to create the queries form the descriptions of the considered methods as described in Section 4.2. Finally, we execute the formulated queries on the index and collect results We only consider top-10 results for each query.

DOI: 10.1002/smr

Previous approaches [1, 5] also only consider the top-10 results suggested by their approach for evaluation.

The top-10 matches found by the TMAP are then analyzed/reviewed manually to determine the effectiveness of the matched results. For a given method in the source API, a match is characterized by a class and a method within that class that is determined to be the corresponding implementation in the target API. Authors next annotated each match as 'relevant' and/or 'exact' based on the following acceptance criteria:

1. **Relevant**: If the target method in the top-10 list can be used to implement the same (or similar) functionality as the source method, we classify the result as relevant.
   OR
   The target method is reported by the previous approaches [5, 14] as a mapping.
2. **Exact**: If the target method is a relevant method and the target method accurately captures the functionality of the source method, and implements the same feature/function, the resultant match is classified as an 'exact' match.
   OR
   The target method is reported by the previous approaches [5, 14] as a mapping.

For example, `getInt` method in `java.sql.ResultSet` type has an exact match in the C# method `GetInt32` from `system.data.sqlclient.SqlDataReader` type, since both methods provide the same functionality of extracting the 32-bit signed integer value stored in a specified column. In contrast, `getClob` method in `java.sql.ResultSet` type does not have an exact corresponding method in C#. The closest functionality available is the C# method `GetValues` in `system.data.sqlclient.SqlDataReader` type. Thus the method `GetValues` is marked as relevant, but not an exact match.

We then calculate coverage (Cov) as the ratio of the number of methods in a type that TMAP found *at-least* one **relevant** mapping to the total number of source methods in that type. We also calculate the Δ_{Cov} as increase in the Cov in comparison to results reported by previous approaches [5, 14] as : $\Delta_{Cov} = TMAP_{cov} - Prev_{cov}$. High value of Δ_{Cov} indicates the effectiveness of TMAP in finding API method mappings. Finally we measure the common methods between the exact mappings suggested by TMAP for a source method with the mappings suggested by previous approaches. We then calculate, the number of new mappings a the number of exact mappings sans the common mappings.

5.3. Results

We next describe our evaluation results to demonstrate the effectiveness of TMAP in leveraging natural language API descriptions to discover method mappings across APIs.

5.3.1. RQ1: Effectiveness of TMAP Table II presents our evaluation results for answering RQ1. The columns 'API' lists the name of source API under 'Source' and target API under 'Target'. The column 'Type' lists the class or interface in source API under consideration for finding mappings in target API. The column 'No. Methods' lists the number of methods in the class or interface under consideration. The columns 'Relevant' lists the number of methods for which at least one relevant mapping is reported. The sub-column 'Prev.' reports relevancy numbers by previous approaches. The previous approach for comparison of Java ME-Android mappings is Rosetta [5]. The previous approach for comparison of Java-C# mappings is StaMiner [14]. The sub-column 'TMAP' reports relevancy numbers by TMAP (at least one relevant method in top-ten results). The column 'Exact' lists the number of methods for which a exact mapping is found. The sub-column 'Prev.' reports exact numbers by previous approaches. Since previous approaches do not make a distinction between exact and relevant, we report same values for both columns. The sub-columns 'TMAP' reports exact numbers by TMAP (at least one exact method mapping in top-ten results). Column 'Δ_{Cov}' lists the ratio of increase in the number of methods for which a relevant mapping was found by TMAP to the total number of methods in the type.

DOI: 10.1002/smr

Table II. Evaluation Results

S No.	API Source	API Target	Type	No. Methods	Relevant Prev	Relevant TMAP	Exact Prev	Exact TMAP	Δ_C
1	Java ME	Android	javax.microedition.lcdui.Alert	16	3	15	3	7	0
2	Java ME	Android	javax.microedition.lcdui.Canvas	22	5	18	5	10	0
3	Java ME	Android	javax.microedition.lcdui.Command	6	3	3	3	0	0
4	Java ME	Android	javax.microedition.lcdui.Graphics	39	18	36	18	29	0
5	Java ME	Android	javax.microedition.lcdui.Font	16	3	15	3	8	0
6	Java	C#	java.io.File	54	15	37	15	26	0
7	Java	C#	java.io.Reader	10	1	8	1	6	0
8	Java	C#	java.io.Writer	10	2	10	2	10	0
9	Java	C#	java.util.Calendar*	47	0	11	0	5	0
10	Java	C#	java.util.Iterator*	3	0	3	0	1	1
11	Java	C#	java.util.HashMap	17	5	9	5	5	0
12	Java	C#	java.util.ArrayList	28	6	22	6	15	0
13	Java	C#	java.sql.Connection	52	1	28	1	13	0
14	Java	C#	java.sql.ResultSet	187	10	146	10	31	0
15	Java	C#	java.sql.Statement	42	1	21	1	5	0
Total				549	73	382	73	171	0.5

*=Previous approach reported a manually constructed class as

Prev= previous approach; Previous approach for Java ME-Android mappings is Rosetta [5]; Previous approach for Java-C#

Our evaluation results indicate the TMAP on average finds relevant mappings for 57% (Column 'Δ_{Cov}') more methods. For the Java ME-Android mappings TMAP performs best for `Alert` and `Font` classes from `javax.microedition.lcdui` package in Java ME API with 75% increase in number of methods for which a relevant mapping was found in Android API. For the Java-C# mappings TMAP performs best for `Iterator` interface from `java.util` package in Java API finding a relevant method in C# API for all the methods. Previous approach StaMiner reports a manually constructed wrapper type as a mappings instead. Furthermore, our results also indicate that TMAP found on average exact mappings for 6.5 $((171-73)/15)$ more methods per type with a maximum of 21 additional exact mappings for a `java.sql.ResultSet` type as compared to previous approaches. We next describe the cases where TMAP did not find any relevant mapping.

A major cause for inadequate performance of TMAP is lack of one-to-one mapping between methods in source and target API. Often times functionality of a method in a source API is broken down into multiple functions in the target API or vice versa. Although, TMAP reports some of the relevant methods, exact mapping may involve a sequence of method calls in target API which is the limitation of TMAP. In the future, we plan to investigate techniques to deal with such cases.

Another cause of inadequate performance of TMAP is inconsistent use of terminology across different APIs. For instance, TMAP did not find any additional relevant mapping for methods in `Command` class in Java ME API. In Java ME API, 'command' is used to refer the user interface construct 'button'. In Android API, 'command' is used in more conventional sense of the term. This inconsistent use of terminology causes TMAP to return irrelevant results. When we manually replaced the term 'command' with 'button' in the generated queries, we observed a relevant method appeared in top ten results for every method in the `Command` class in Java ME API. However, we refrain from including such modifications to stay true to TMAP approach for evaluation. In the future, we plan to investigate techniques to automatically suggest alternate keywords.

5.3.2. RQ2: Quality of discovered mappings To answer RQ2, we compared the exact mappings discovered by TMAP with the mappings discovered by previous approaches. In Table II the previous approach for comparison of the Java ME-Android mappings is Rosetta [5] and the previous approach for comparison of Java-C# mappings is StaMiner [14]. Our results show that out of 171

Prepared using smrauth.cls
DOI: 10.1002/smr

discovered exact mappings only 22 are in common with previous approaches. We next discuss some of the implications of the results.

Before carrying out this evaluation, we expected that the mappings discovered by TMAP would significantly overlap with the mappings discovered by the Rosetta and StaMiner, as these approaches infer mappings from actual source code. Thus, these mappings can be considered as the representative of how developers are actually migrating software. However, the results suggest a low overlap. We manually investigated the possible TMAP specific implications of the observed mismatch.

The results (matches found) comprise of methods from different classes in the target API, reflecting that often there are multiple ways to solve a problem, or to implement a feature using an API. Further, choice of using multiple APIs gives a developer the flexibility to use different approaches when porting an application from one platform to another.

When more than one match is found for a given source method, the results in TMAP are ranked according to the similarity score, with the more relevant (or exact matches) ranked higher. The ranked set of results helps the developer use the best suited or most appropriate target method in their implementation. This approach is different from earlier approaches [5, 14] that focused on only exact matches between different classes using the number of similar methods as a basis.

We also contacted the authors of Rosetta approach [5] to report the difference in the mappings. Specifically, we inquired that Rosetta does not report any method form `AlertDialog` class in the Android API as a possible mapping for `Alert.setString` method in Java ME API. Rosetta reports method sequence `Paint.setAlpha CompundButton. setChecked` as one of the likely mappings. In contrast, TMAP discovers `AlertDialog.setTitle` method in Android API as a likely mapping.

The lead author of Rosetta approach responded that they restricted the output of Rosetta to sequences of length up to 2 when inferring mappings (i.e., A→p;q, or A→p). Furthermore, authors count a reported method sequence as a valid mapping if the reported sequence implements some of the functionality of a source method on the target platform. With regards to our query, Rosetta authors observed that in many of the traces, setting a `string` first involves setting the attributes of the `Paint` (with is used to draw the text), followed by a call to `setText` method, which led them to believe that the sequence `Paint.setAlpha CompundButton. setChecked` could be a likely mapping, if at least in part. Although, author did confirm that technically `AlertDialog.setTitle` method in Android API as a likely mapping.

The exchange with Rosetta approach's lead author points out that the mappings discovered by TMAP generally point to the closest aggregate API in contrast to the individual smaller API calls that achieve the same functionality. Furthermore, the exchange also demonstrates the reliance of the code analysis approaches on the quality of the code under analysis. In contrast, TMAP relies on the quality of the API method descriptions.

6. LIMITATION AND FUTURE WORK

We next describe the limitations and future work of TMAP, followed by discussions on threats to validity.

A key limitation of the presented work is its reliance on the human developer to confirm or refute mappings. In the future, we plan to extend the TMAP infrastructure to achieve an end-to-end automation. Particularly, we plan on using the program-analysis techniques, such as type-analysis proposed in existing approaches [14, 26].

Sometimes the functionality achieved by a method call in a source API, is achieved by a sequence of method calls in the destination API and vice versa. Although TMAP may return individual methods as relevant, TMAP does not provide explicit sequences of method calls as relevant suggestions. In the future, we plan to extend the current text mining infrastructure to provide method sequences as relevant suggestions when applicable. In particular, we plan to leverage the NLP techniques, such as specification inference [16] to identify method sequences.

DOI: 10.1002/smr

From an implementation perspective, TMAP does not take into account the API fields, which limits TMAPs ability in reporting mappings involving API fields. However, disregarding API fields is a limitation of the current implementation and in future iterations of TMAP implementations we plan to include API fields in the indexes as well.

Finally, TMAP operates under the assumption of the availability of the API documents. Thus TMAP is not applicable in situations where API documents are of low quality or are unavailable altogether. In the future, we plan to extend TMAP infrastructure to workaround such situations by integrating with existing source-code-mining based approaches. Specifically we plan to leverage techniques like code summarisation [20] and IR based approaches like Exoa [9].

Threats to Validity: The primary threat to external validity is the representativeness of our experimental subjects to the real world software. To address this threat we chose real world API pairs: 1) Java ME and Android APIs are two Java based platforms to develop mobile applications; 2) Java and C# APIs are the top object-oriented programming language APIs used for generic application development. The threat can be further minimized by evaluating TMAP on more APIs from different domains.

Our chief threat to internal validity is the accuracy of TMAP in identifying API mappings. To minimize this threat we compared the mappings inferred by TMAP with the mappings provided by previous approaches. We thank Gokhale et al. [5] for sharing with us the Java ME and Android API mappings inferred by their Rosetta approach. We also thank Nguyen et al. [14] for making their mappings publicly available. Furthermore, authors did manually identify some of the mappings that could not be compared to previous work. Thus, human errors may affect our results. To minimize the effect, each annotation was independently agreed upon by two authors.

7. RELATED WORK

Language migration is an active area of research [5,7,13,14,22,23,25], 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. [22] 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. [3] suggest that most of these approaches support only a subset of API's for migration. A recently published survey by Robillard et al. [18] provides a detailed overview of techniques dealing with mining API mappings.

Among other works described in [18], Mining API Mapping across different languages [25] (MAM) is most directly related to our work. MAM takes into input a software (S) written in source language and manually ported version of S (S') written in target language. MAM then applies a technique called "method alignment" that pairs the methods with similar functionality across S and S'. These methods are then statically analyzed to detect mappings between source and target language API. Recently, Nguyen et al. [14, 15] proposed StaMiner, an approach that applies statistical-machine-translation based techniques to achieve language migration. They consider source code as a sequence of lexical tokens (lexemes) and apply a phrase-based statistical-machine-translation model on the lexemes of those tokens to achieve migration. While MAM and StaMiner require as input software that has been manually ported from a source to target API (both S and S'), our approach is independent of such requirement. In contrast, TMAP relies on text mining of source and target API method descriptions (that are typically publicly available) to discover likely mappings.

Gokhle et al.'s [5] approach Rosetta addresses the limitations of MAM to infer method mappings. In particular, Rosetta relaxes the constraint of having software that has been manually ported from a source to target API. Instead, they use functionally similar software in source and target API. For instance, they use two different 'tic-tac-toe' game applications in Java ME and Android API not necessarily manually ported. Rosetta approach then requires user to manually execute these applications under identical (or near identical) inputs and collect execution traces. Finally Rosetta analyses the collected execution traces to infer method mappings. In contrast, TMAP is independent

DOI: 10.1002/smr

of both the requirements: 1) to have functionally similar applications, 2) to manually execute such applications using similar inputs.

Furthermore, from an infrastructure perspective, TMAP is independent of the programming language or API under consideration. In contrast, program analysis based approaches like [5, 14, 25] may need significant efforts for adding support to additional APIs and programming language.

Zheng et al. [24] mine search results of a web search engine, such as Google to recommend related APIs of different libraries. In particular, they propose heuristics to formulate keywords using the name of the source API, and the name of target API to query a web search engine. For instance, to search for an equivalent class in C# for the `HashMap` class in Java, a user may manually enter "HashMap C#" in a web search engine. The results are computed one by one and candidates are ranked by relevance, mainly according to their frequency of the appearance of keywords in the query. However, authors provides only preliminary results and queries proposed are of a coarse grain. Furthermore, the results are susceptible to influence by the webpages returned by a web search engine. In contrast, TMAP is independent of the web search engine results.

Information retrieval techniques [1, 6, 9, 17] are also being increasingly used in Code Search. We next describe some relevant approaches. Chatterjee et al.'s [1] approach Sniff annotates the source code with API document descriptions. Sniff then performs additional type analysis on the source code to rank relevant code snippets. Grechanik et al.'s [6] approach Exemplar uses the text in API documents to construct a set of keywords associated with an API call. Exemplar then uses the keywords list to facilitate query expansion to achieve code search. However, these approaches are targeted towards code search in one API. In contrast, TMAP discovers method mappings across multiple APIs.

Text analysis [2, 10, 16, 26, 27] of API documentation is increasing being used to infer interesting properties from software engineering perspective. For instance, Zhong et al. [26] employ natural language processing (NLP) and machine learning (ML) techniques to infer resource specifications (rules governing the usage of resources) from API documents. Treude et al. [21] also leverage rule-based NLP approaches to infer action (programming actions described in documentation) oriented properties from an API document. In contrast, TMAP uses text mining a comparatively lightweight-approach instead of sophisticated NLP techniques used in these approaches. Furthermore, TMAP discovers API mapping relations across different API for language migration, whereas the previous approaches mine properties of one API.

8. CONCLUSION

API mapping across different platforms/languages mappings facilitate machine-based migration of an application from one API to another. Thus tool assisted discovery of such mappings is highly desirable. In this paper, we presented TMAP: a lightweight text-mining based approach to infer API mappings. TMAP compliments existing mapping inference techniques by leveraging natural language descriptions in API documents instead of relying on source code. We used TMAP approach to discover API mappings for 15 types across: 1) `Java` and `C#` API, 2) `Java ME` and `Android` API. We demonstrated the effectiveness of TMAP by comparing the discovered mappings with state-of-the-art code analysis based approaches. Our results indicate that TMAP on average found relevant mappings for 57% more methods compared to previous approaches. Furthermore, our results also indicate that TMAP found on average exact mappings for 6.5 more methods per type with a maximum of 21 additional exact mappings for a single type as compared to previous approaches.

ACKNOWLEDGEMENT

This work is funded by the USA National Security Agency (NSA) Science of Security Lablet. Any opinions expressed in this report are those of the author(s) and do not necessarily reflect the views of the NSA. We also thank the Realsearch research group for providing helpful feedback on this work.

DOI: 10.1002/smr

REFERENCES

1. S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *Proc. of 12th FASE*, pages 385–400. 2009.
2. U. Dekel and J. D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.
3. 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.
4. W. Frakes. Introduction to information storage and retrieval systems. *Space*, 14:10, 1992.
5. A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proc. 35nd ICSE*, 2013.
6. M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *Proc. 32nd ICSE*, volume 1, pages 475–484, 2010.
7. A. E. Hassan and R. C. Holt. A lightweight approach for migrating web frameworks. *Inf. Softw. Technol.*, 47(8):521–532, Jun 2005.
8. Java 2 CSharp Translator for Eclipse. http://sourceforge.net/projects/j2cstranslator/.
9. J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Towards an intelligent code search engine. In *Proc. AAAI*, 2010.
10. G. Little and R. C. Miller. Keyword programming in Java. In *Proc. 22nd ASE*, pages 84–93, 2007.
11. Apache Lucene Core. http://lucene.apache.org/core/.
12. C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*, volume 1. Cambridge University Press, 2008.
13. M. Mossienko. Automated Cobol to Java recycling. In *Proc. 7th CSMR*, pages 40–, 2003.
14. 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 *Proc. 29th ASE*, pages 457–468, 2014.
15. A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 544–547. ACM, 2014.
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. S. P. Reiss. Semantics-based code search. In *Proc. 31st ICSE*, pages 243–253, 2009.
18. 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.
19. A. Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.
20. G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proc. of 19th ICPC*, pages 71–80, 2011.
21. C. Treude, M. Robillard, and B. Dagenais. Extracting development tasks to navigate software documentation.
22. A. Van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. 21st ICSE*, pages 246–255, 1999.
23. R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. on Software Engineering*, 14(8):1207–1228, 1988.
24. W. Zheng, Q. Zhang, and M. Lyu. Cross-library API recommendation using web search engines. In *Proc. 13th ESEC/FSE*, pages 480–483, 2011.
25. 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.
26. 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.
27. 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.

DOI: 10.1002/smr