Statistical Learning Approach for Mining API Usage Mappings for Code Migration

Anh Tuan Nguyen[1] anhnt@iastate.edu
Hoan Anh Nguyen[1] hoan@iastate.edu
Tung Thanh Nguyen[2] tung.nguyen@usu.edu
Tien N. Nguyen[1] tien@iastate.edu

[1]Electrical and Computer Engineering Department, Iowa State University, Ames, IA 50011, USA
[2]Computer Science Department, Utah State University, Logan, UT 84322, USA

ABSTRACT

The same software product nowadays could appear in multiple platforms and devices. To address business needs, software companies develop a software product in a programming language and then migrate it to another one. To support that process, semi-automatic migration tools have been proposed. However, they require users to manually define the mappings between the respective APIs of the libraries used in two languages. To reduce such manual effort, we introduce StaMiner, a novel data-driven approach that statistically learns the mappings between APIs from the corpus of the corresponding client code of the APIs in two languages Java and C#. Instead of using heuristics on the textual or structural similarity between APIs in two languages to map API methods and classes as in existing mining approaches, StaMiner is based on a statistical model that learns the mappings in such a corpus and provides mappings for APIs with all possible arities. Our empirical evaluation on several projects shows that StaMiner can detect API usage mappings with higher accuracy than a state-of-the-art approach. With the resulting API mappings mined by StaMiner, Java2CSharp, an existing migration tool, could achieve a higher level of accuracy.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

API Mappings; Code Migration; API Usages; Statistical Learning

1. INTRODUCTION

Software companies nowadays often originally develop software in one language and then migrate them to another language to address the business need that the same software is required to appear on multiple platforms and devices. The process of migrating software between languages is called *code migration*.

Toward helping developers in the process of code migration, there exist semi-automatic approaches and supporting tools [15, 42, 43, 44]. Those tools and methods require users to define the *migration rules* between the respective program constructs and the *mappings* between the corresponding Application Programming Interfaces (APIs) that are used in two languages. For example, a call to a superclass with super in Java needs to be translated into the call with base in C#. As an example for the APIs in libraries, to manipulate files and directories in a file system, developers use the APIs in the Java Development Kit library (JDK) such as java.io.File.mkdirs() and java.io.File.delete(). The same functionality can be achieved in C# with the APIs in System.IO such as System.IO.Directory.CreateDirectory(String) and System.IO.File.Delete(), respectively. Those mappings are called *API mappings* between languages. The existing tools and methods expect programmers to specify such API mappings. There are usually a large number of API mappings and many of them are newly introduced from time to time as well. Thus, existing tools can support only a subset of needed API mappings [39]. Importantly, it has been reported that as a result, the quality of the migrated code is reduced due to missing API mappings in such rule-based migration tools [8, 39].

Table 1: Examples of API Mappings from Java to C#

	API usages in Java	API usages in C#
1	copy.setFirstLineIndent(getFirstLineIndent())	copy.FirstLineIndent = FirstLineIndent
2	str.charAt(i)	str[i]
3	HtmlTags.ALIGN_CENTER. equalsIgnoreCase(alignment)	Util.EqualsIgnoreCase (HtmlTags.ALIGN_CENTER, alignment)
4	File.mkdirs()	System.IO.Directory.CreateDirectory(string)
5	HashMap<String,Integer>	Dictionary <string,int>
6	Vector.removeAllElements()	ArrayList.Clear()

To reduce manual effort in defining such rules of API mappings for code migration, there exist approaches that automatically *mine API mappings* from the corpus of the libraries' client code that already has two corresponding versions in two languages [30, 39]. In Twinning [30], users have to manually write the rules to specify the mappings as program transformations. MAM [39] represents API usages in the client code as a graph among API elements (i.e. API methods/classes). The two API usage graphs for the corresponding client code in two languages are matched against each other to determine the API mappings. The nodes (API methods/classes) in two usage graphs are mapped based on a set of *heuristics* such as the similarity of the nodes' names and/or the numbers and types of the parameters in method calls, as well as their returned types.

Those heuristics might not always hold because in general, a library in the target language could use a quite different paradigm with the APIs' names different than those in the library used in the source language. A method call in a library might be replaced by a field access or by multiple APIs in the target library. Even when it is replaced by a method call, the set of parameters might be different. Table 1 displays the corresponding API usages providing the same functionality that we found in the Java and C# versions of the project db4o [7]. In the first example, the pair of *getter* and *setter* functions becomes a *field access* and an *assignment*. In the second one, a call to charAt was migrated into an *array access* str[i]. In the third one, the *numbers of parameters* of the corresponding APIs are different, and an access to ALIGN_CENTER becomes an argument passed to an API call. The last three examples show that the names of the respective API classes/methods can be very different. Moreover, a usage with two APIs can be mapped into a usage with three APIs, e.g., File.new(string) and java.io.File.exists() in Java become FileInfo.new(string) and (System.IO.File.Exists(string) || System.IO.Directory.Exists(string)) in C# since java.io.File.exists() in Java can check if a file or directory exists, while such checking for a file or a directory in C# is achieved with two different APIs [39].

Importantly, in other cases, the API mappings of individual API classes/methods are not sufficient because an API usage as a whole needs to be migrated into another usage in the target library. In such cases, existing tools, which rely on one-to-one API mappings, will not work. Thus, providing mappings for entire API usages would help developers to understand and migrate the code themselves. The notion of *API usage mapping* encompasses the notion of *API mapping* since a usage could be a single API element.

In this paper, we propose StaMiner, a data-driven model that statistically learns the mappings between API usages from the corpus of the corresponding methods in the client code of the APIs used in two languages Java and C#. For a method, it builds a Groum [26], a graph representation for the API usages in the method. In a Groum, nodes represent method calls, field accesses, and control structures (i.e., if, while, for). Edges represent the control and data dependencies between the nodes. The nodes' labels are built from the names of the classes, methods, and control structures.

StaMiner extracts from the method's Groum the connected sub-groums representing the API (sub)usages with one or multiple objects. For each of those sub-groums, it extracts its nodes' labels to build a sequence of symbols. Those labels/symbols are called *usage symbols*, and a *sequence of such (usage) symbols* represents an API (sub)usage. Then, the sequences of symbols for all sub-groums are put together to form a sequence for the whole method, called the *sentence*. All the pairs of sentences for the corresponding methods in Java and C# are used as the input for a symbol-to-symbol alignment algorithm using Expectation-Maximization (EM) [18]. After that, StaMiner extends the resulting symbol-to-symbol alignments to compute the alignments for the sequences of symbols in two languages. The aligned sequences of symbols with their scores greater than a threshold represent the mappings of API usages.

Importantly, in StaMiner, building the corpus of respective methods of client code requires little effort of manually labeling. We built a tool to search for the respective methods in the pairs of corresponding implementations in Java and C#. With a conservative strategy, it chose only the methods with the same signatures in the classes with the same/similar names (with possible upper/lowercase differences) in the same/similar directory structures in both Java and C# versions. Such pairs of methods likely realize the same functions. We ran that tool on nine open-source projects with both Java and C# versions. We found 34,628 pairs of respective methods and used them as a corpus for mining. Other methods are not used since we are not sure if they correspond to each other.

Our empirical evaluation shows that StaMiner is able to mine a large number of API mappings with 87.1% accuracy including many *m*-to-*n* mappings. Comparing against the oracle of manually built single API mappings, it achieves higher precision (17.1% relative improvement) and recall (28.6%) than the state-of-the-art approach MAM [39]. We also evaluated how much benefit that the mined API

a) A Usage in Java

```
HashMap<String, Integer> myVocabIdxDict = new HashMap<String,
        Integer>();
myVocabIdxDict.put("alphabet", 1);
FileWriter  writer  = new FileWriter("VocabIdx.txt");
for  (String vocab: myVocabIdxDict.keySet()){
   writer.append(vocab + " " + myVocabIdxDict.get(vocab)+"\r\n");
}
writer.close();
```

b) A Usage in C#

```
Dictionary<string,  int> myVocabIdxDict = new Dictionary<string,
        int>();
myVocabIdxDict.Add("alphabet", 1);
StreamWriter writer = new StreamWriter("VocabIdx.txt");
List<string> keyList  = new List<string>(myVocabIdxDict.Keys);
foreach( string  vocab  in  keyList){
     int  idx;
     myVocabIdxDict.TryGetValue(vocab, out idx);
     writer.WriteLine(vocab + " " + idx);
}
writer.Close();
```

Figure 1: API Usage Mappings between Java and C#

mappings from StaMiner could help in supporting code migration. The result shows that with the single API mappings mined from StaMiner, Java2CSharp [15], an existing migration tool, produces code with 4.2% less compilation errors and 6.0% less defects than with the mappings from MAM. The contributions of this paper are

1. A statistical method for mining API mappings with any arities,

2. A tool to mine API usage mappings in Java and C#, and

3. An empirical evaluation to show StaMiner's higher accuracy and usefulness than the state-of-the-art approach MAM.

2. A MOTIVATING EXAMPLE

Figure 1 shows the corresponding client code in Java and C# listed on stackoverflow.com. The code is for the tasks of *reading the data* from a vocabulary containing pairs of words and indexes after *populating it*, and *then writing them line by line to a text file*. To do that, developers use the Application Programming Interface elements (**API elements**, APIs for short), which are the *classes*, *methods*, and *fields* provided by a **framework** or **library**. Those API elements are used in a specific order with specific data and/or control dependencies and control structures (i.e. condition/repetition) according the API specification. Such a usage of API elements is *used to achieve a programming task* and is called an **API usage**. For example, Figure 1a can be viewed as containing three (sub)usages. First, the vocabulary is instantiated and populated via HashMap.put. Second, the task of reading data from a vocabulary begins with a for iteration over all elements in the vocabulary using HashMap.keySet. Each element is then retrieved by the HashMap.get method of the vocabulary. Finally, the file writing usage starts with a FileWriter object instantiation, continues with a for loop where FileWriter.append is called on that object, and finally ends with the closing of the FileWriter object via FileWriter.close.

To migrate code between two languages, one needs to implement a respective API usage in C# that *achieves the same programming task(s)* as the original API usage in Java. Such two respective API usages are called **an API usage mapping**. In a special case, if each respective API usage has a single API class or method, the mapping is called an **API mapping** (i.e., a class to a class or a method call to a method call). Currently, the existing automatic mining approaches for API (usage) mappings, which rely on the heuristics of name or structure similarity among API elements, face several challenges:

Firstly, the names of the respective API elements in two languages could be different. For example, C# utility library does not have a HashMap class as in Java, and one can use Dictionary instead. The classes have different names and their respective methods for adding an element to the collection have different names as well (HashMap.put versus Dictionary.Add). Similarly, to write texts to a file, one could use FileWriter.append in Java, and use StreamWriter.WriteLine in C#. In fact, the latest version of Java2CSharp [15] contains a manually written data set of 1,302 API mappings. 74% of them have different API names in Java and C# (with case sensitivity being ignored). Thus, an API mapping tool cannot be based solely on the similarity in names of API elements. Secondly, the numbers of parameters in corresponding methods are not necessarily the same. For example, the method HashMap.get(String) (line 5, Figure 1a) is used to retrieve a value given a key. To do that with C#'s Dictionary, one would use Dictionary.tryGetValue(String, int) (line 7, Figure 1b) with the second parameter being the reference to the index of the key if it is found in the collection (Dictionary does not have a get method). In fact, the authors of MAM [39] confirmed that a key source of its inaccuracy is the difference in names and parameters. Thirdly, an API element can be mapped to another one of a different kind (HashMap.keySet() became Dictionary.Keys).

Finally, sometimes, to achieve the same task, one API in a usage can become multiple APIs or vice versa. For example, to retrieve the set of keys in HashMap in Java, one could use HashMap.keySet (line 4, Figure 1a). However, in C#, since the field access Dictionary.Keys (line 4 of Figure 1b) returns a collection, the equivalent usage involves a List object encapsulating the collection. In fact, an API usage mapping is generally m-to-n where m might be different than n. For example, in Figure 1a, the task of *"writing to a text file"* in Java involves FileWriter.new, a for loop, FileWriter.append, and FileWriter.close. The respective usage in C# involves StreamWriter.new, List, a foreach loop, StreamWriter.WriteLine, and StreamWriter.close in Figure 1b. Currently, the existing rule-based migration tools support only one-to-one mappings. Without tool support in such general m-to-n mapping cases, developers would need to manually perform code migration. It would be beneficial to them if a tool could provide the mappings for entire API usages.

3. MINING OF API USAGE MAPPINGS

3.1 Formulation and Important Concepts

We propose StaMiner, a *statistical model* that learns the API usage mappings from the corpus of the respective client code of the APIs in two languages. Let us first present important concepts.

Definition 1 (**API Element**). *API elements are the classes, methods, and fields provided by a library or a framework.*

Definition 2 (**API Client Code**). *Client code of a library/framework is the code that uses API elements of that library/framework.*

Definition 3 (**API Usage**). *An API usage is a set of API elements in use in the client code (i.e., classes, method calls, field and array accesses, and operators), together with control units (i.e., condition/repetition) in a specific order, and with control and data flow dependencies among API elements.*

An API usage is an intended use to achieve a programming task. A usage could involve multiple libraries/frameworks. It contains the usage of the classes (via variables/ references), methods (via method calls), overloaded operators, or control structures (e.g. while, if), with specific orders of those elements and their dependencies. Figure 1a (lines 1-2, 4-5) shows the usage of retrieving data from

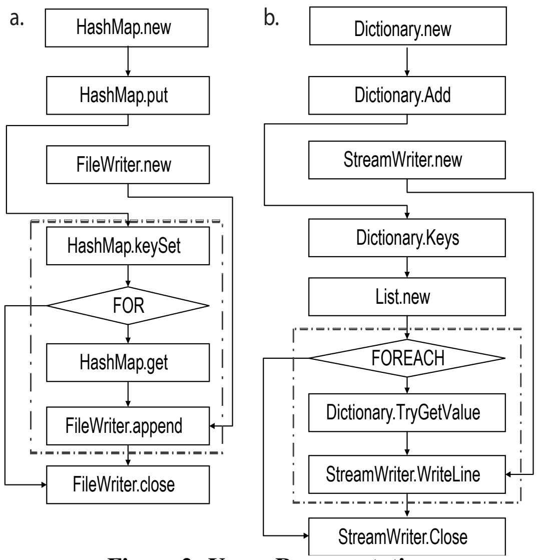

Figure 2: Usage Representation

a HashMap collection in JDK. A usage can be a composite one comprised of multiple sub-usages for multiple sub-tasks. Usages can interleave with others (Figure 1a). In our prior work [26], we developed a graph-based model, called *Groum* to represent API usages.

Definition 4 (**Groum**). *A Groum is a graph in which the nodes represent actions (i.e., method calls, overloaded operators, and field accesses) and control points (i.e., branching points of control structures such as if, while, for, etc.). The edges represent the control and data dependencies between the nodes. The labels of the nodes are built from the names of classes/methods/control structures.*

In our prior work [26], we showed that an API usage (even a composite or interleaving usage) can be represented via a connected subgraph in a Groum. Figures 2a and b illustrate the usages in Figures 1a and b in Java and C# as Groums. For clarity, we label the nodes with the simple names of classes and methods. In the implementation, the nodes' labels have fully qualified names and an action node for a method call also has its parameters' types.

In Figure 2a, the action nodes such as HashMap.new, HashMap.put, FileWriter.new, etc. represent the method calls. An edge connects two action nodes when there are data and control flow dependencies between two nodes. For example, HashMap.new is connected to HashMap.put because the former method call occurs before the latter method call for the initialized HashMap object to be used in the latter call. The control node FOR represents the for iteration. Note that HashMap.keySet in the condition is executed before the control point FOR, thus, its node comes before the node FOR. The edges from HashMap.keySet to FOR and from FOR to HashMap.get represent the control flows. The rectangle containing HashMap.get and FileWriter.append models the scope of the FOR loop. Moreover, if a method call is an argument of another call, e.g., m(n()), the node for the method call in the argument will be created before the node for the outside method call (i.e., the node for n comes before that of m). The rationale is that n is evaluated before m. The same semantics is applied to the Groum in Figure 2b for the usage in C#.

Definition 5 (**Usage Symbol**). *A usage symbol (*symbol *for short) of a node in a Groum is the node's label.*

Definition 6 (**Usage Sequence**). *A usage sequence of a Groum is the sequence of the usage symbols that are extracted for the Groum's nodes following their corresponding order in the Groum.*

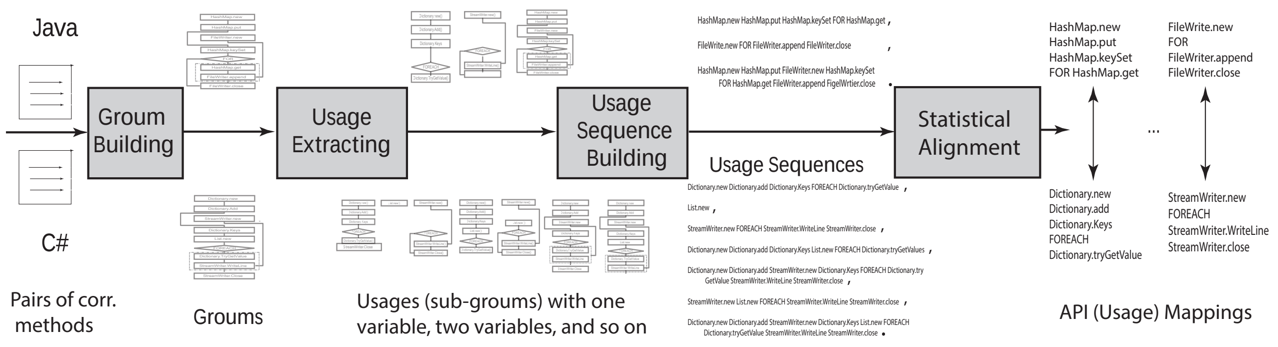
Java
C#
Pairs of corr. methods
Groum Building
Grooms
Usage Extracting
Usages (sub-groums) with one variable, two variables, and so on
Usage Sequence Building
HashMap.new HashMap.put HashMap.keySet FOR HashMap.get ,
FileWrite.new FOR FileWriter.append FileWriter.close ,
HashMap.new HashMap.put FileWriter.new HashMap.keySet FOR HashMap.get FileWriter.append FigelWrtier.close .
Usage Sequences
Dictionary.new Dictionary.add Dictionary.Keys FOREACH Dictionary.tryGetValue ,
List.new ,
StreamWriter.new FOREACH StreamWriter.WriteLine StreamWriter.close ,
Dictionary.new Dictionary.add Dictionary.Keys List.new FOREACH Dictionary.tryGetValues ,
Dictionary.new Dictionary.add StreamWriter.new Dictionary.Keys FOREACH Dictionary.try GetValue StreamWriter.WriteLine StreamWriter.close ,
StreamWriter.new List.new FOREACH StreamWriter.WriteLine StreamWriter.close ,
Dictionary.new Dictionary.add StreamWriter.new Dictionary.Keys List.new FOREACH Dictionary.tryGetValue StreamWriter.WriteLine StreamWriter.close .
Statistical Alignment
HashMap.new HashMap.put HashMap.keySet FOR HashMap.get
Dictionary.new Dictionary.add Dictionary.Keys FOREACH Dictionary.tryGetValue
...
FileWrite.new FOR FileWriter.append FileWriter.close
StreamWriter.new FOREACH StreamWriter.WriteLine StreamWriter.close
API (Usage) Mappings

Figure 3: StaMiner's Overview

For example, the usage sequence for the Groum in Figure 2a is HashMap.new HashMap.put FileWriter.new HashMap.keySet FOR HashMap.get FileWriter.append FileWriter.close.

If two nodes in a Groum do not have a specific order among them, we use their corresponding appearance order in the source code that is used to build the Groum, for their usage sequence.

Definition 7 (**Vocabulary**). *All distinct usage symbols extracted from the Groums of all methods in a corpus are collected into a vocabulary of our statistical learning model.*

Definition 8 (**API Usage Mapping**). *Given a corpus of the corresponding methods in two languages, the task of mining API usage mappings is to detect the corresponding usage sequences with highest probabilities in the corresponding code.*

3.2 Approach Overview

Figure 3 shows StaMiner's overview. Its input is the corpus of the client code using the APIs in the libraries/frameworks in two languages. Specifically, the input is the pairs of the corresponding methods in the client code of the libraries/frameworks (Section 1). For each method in a language, it builds a Groum G using our Groum building algorithm [26] to represent the entire API usage in the method. It then extracts from that method's Groum the *connected sub-groums* representing smaller API usages. In theory, we should extract sub-groums covering all possible paths at all sizes in G to represent for all possible usages. However, to reduce exponential complexity for better performance, we extract the usages that involve one variable, two variables, and so on. For each of those sub-groums, StaMiner collects its nodes' labels to build the usage sequence (Definition 6). The rationale for building usage sequences is the use of sequence-based alignment (Section 5). The usage sequences for all sub-groums are put together and separated by commas to form the *sequence for the usage of the entire method*, called a *sentence*. The usage sequences for sub-groums are placed in the sentence in the appearance order of the first variables in the code (Section 4).

All the pairs of sentences for the corresponding methods in the two libraries are then used as the input for a symbol-to-symbol alignment algorithm using Expectation-Maximization (EM) [18]. After that, StaMiner extends the resulting symbol-to-symbol alignments to compute the alignments for the usage sequences between the two languages. The aligned usage sequences represent the mappings of API usages. Finally, the aligned usage sequences with their scores higher than a threshold are kept and presented in the sequence form.

4. USAGE SEQUENCE BUILDING

In this section, we describe how StaMiner extracts the sub-Groums from a method's Groum for the usages involving one and multiple objects, and builds the usage sequences for them.

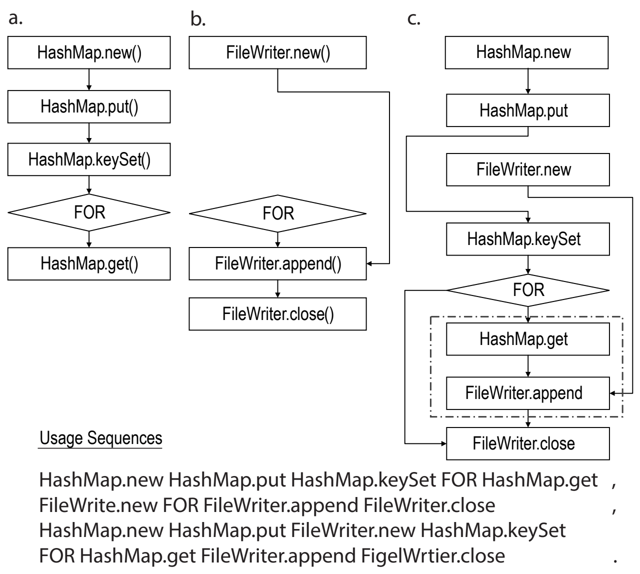

Figure 4: Extracting Usages and Building Usage Sequences

First, StaMiner analyzes the method and its Groum G_M. Since statically analyzing source code to extract API usages, it collects the variables in the method that represent the objects in the usages. In the case of cascading method calls (e.g., m1().m2()), it also creates a temporary variable to represent the object returned by the former method call (e.g., the value returned by m1()). Second, it collects into a set T all the nodes according to their order in the Groum G_M. If two nodes do not have an order, it uses their appearance order in the source code that is used to build the Groum. Third, it starts to extract the sub-Groums with one variable, two variables, and so on, and then builds the usage sequences for them. To do that, it performs standard program slicing to collect a set of k variable nodes and the related control nodes in G_M having data and control flow dependencies to form a sub-groum with k variables. The rationale for the requirement of dependencies is that if the objects or control units do not have dependencies, they should not be in the same usage. Fourth, for each sub-groum with k variables, based on the set T, StaMiner produces the usage sequences from the nodes' labels according to their order in G_M. Finally, all usage sequences, which are separated by ",", are collected into a sentence that ends with a period. The punctuation marks are used to mark the boundaries of sequences and sentences, and to allow us to remove the alignments of the invalid sequences crossing such boundaries. They are also used as the pivots to improve the alignment process. Note that other punctuation marks could be used in place of commas and periods.

```
1 Dictionary.new Dictionary.add Dictionary.Keys FOREACH Dictionary.
        tryGetValue ,
2 List.new ,
3 StreamWriter.new FOREACH StreamWriter.WriteLine StreamWriter.close ,
4 Dictionary.new Dictionary.add Dictionary.Keys List.new FOREACH
        Dictionary.tryGetValues ,
5 Dictionary.new Dictionary.add StreamWriter.new Dictionary.Keys
        FOREACH Dictionary.tryGetValue StreamWriter.WriteLine
        StreamWriter.close ,
6 StreamWriter.new List.new FOREACH StreamWriter.WriteLine
        StreamWriter.close ,
7 Dictionary.new Dictionary.add StreamWriter.new Dictionary.Keys List.new
        FOREACH Dictionary.tryGetValue StreamWriter.WriteLine
        StreamWriter.close .
```

Figure 5: Usage Sequences for C# code in Figure 1b

```
1  function EM_Algorithm(Training corpus T)
2    Initialize $\lambda, \pi, \tau$ uniformly
3    repeat
4      reset all counts to 0
5      for each pair (s,t) in T:
6        m = length(s), l = length(t)
7        count(m,l)++, count(l)++
8        for i = 1..m, j = 0..l:
9          u = $s_i$, v = $t_j$
10         p(i,j) = $\pi$(j,i,m,l) * $\tau$(u,v)
11         $\delta$(i,j) = p(i,j)/$\sum_{j=0}^{l}$p(i,j)
12         count(j,i,m,l) += $\delta$(i,j), count(i,m,l) += $\delta$(i,j)
13         count(u,v) += $\delta$(i,j), count(v) += $\delta$(i,j)
14     for all m,l:
15       $\lambda$(m,l) = count(m,l)/count(l)
16     for all j,i,m,l:
17       $\pi$(j,i,m,l) = count(j,i,m,l)/count(i,m,l)
18     for all u,v:
19       $\tau$(u,v) = count(u,v)/count(v)
20   until convergence
21 return ($\lambda, \pi, \tau$)
```

Figure 6: Expectation-Maximization Algorithm [3]

Figure 4 shows the illustration for the usage sequences built for the Groum in Figure 2a. Figure 4a shows the nodes in the Groum of Figure 2a that are involved in the usage of a single object of type HashMap (corresponding to the variable myVocabIdxDict in Figure 1a). Figure 4b corresponds to the nodes in the Groum of Figure 2a that are involved in the usage of the FileWriter object writer in Figure 1a. Figure 4c is the same as the Groum in Figure 2a since it corresponds the usage involving two variables (myVocabIdxDict and writer in Figure 1a). The labels are collected as described earlier to form a usage sequence. The sequences for those usages are put together in the order of the numbers of references and then the appearance order of the first reference/variable in the code.

The Groum of the corresponding method in C# is processed to create the usage sequences. The usage sequences in the sentence for the corresponding method in C# in Figure 2b are shown in Figure 5.

5. STATISTICAL ALIGNMENT

This section presents how we apply statistical alignment on the extracted usage sequences. We use IBM Model 2 [3], a model for symbol-to-symbol aligning. Let us present it and then an extension to derive the alignments of the usage sequences in Java and C#.

5.1 Symbol-to-Symbol Alignment

The input of this process is the collection of pairs of sentences for the methods in the source and target languages, respectively. Assume that L_S and L_T are two sets of sequences in two languages, and $s = s_1 s_2 ... s_m$ in L_S and $t = t_1 t_2 ... t_l$ in L_T. The goal of IBM Model 2 is to compute the probability $P(s|t)$, that is, the probability that s is the corresponding of t given the observable t. To do that, IBM Model 2 considers s to be generated with respect to t by the following generative process. First, a length m for s is chosen with probability $P(m|t)$. For each position i, it chooses a symbol $t_j \in t$ and generates a symbol s_i based on t_j. In this case, it considers s_i to be aligned with t_j. Such alignment is denoted by an alignment variable $a_i = j$. The symbol s_i can also be generated without considering any symbol in t. In this case, s_i is considered to be aligned with a special symbol null. The vector $a = (a_1, a_2, ... a_m)$ with the value of a_i within $0..l$ is called *an alignment* of s and t ($a_i = 0$ means no alignment in t for a_i). To practically compute $P(s,t)$, IBM Model 2 makes the following assumptions:

1. The choice of length m of s is dependent on only the length l of t, i.e., $P(m|t) = \lambda(m,l)$;

2. The choice of the alignment $a_i = j$ depends on only the position i and the two lengths m and l, i.e., $P(a_i|i,m,t) = \pi(j,i,m,l)$;

3. The choice of symbol $s_i = u$ of s depends on only the aligned symbol $t_j = v$, i.e., $P(s_i|t_{a_i}, i, m, t) = \tau(u,v)$.

With those independent choices, the model computes:

$$P(s,a|t) = \lambda(m,l) \prod_{i=1}^{m} (\pi(a_i,i,m,l).\tau(s_i,t_{a_i})) \quad (1)$$

Thus, $P(s|t)$ is computed by summing over all alignments:

$$\begin{aligned} P(s|t) &= \sum_a P(s,a|t) = \sum_{a_1=0}^{l} ... \sum_{a_m=0}^{l} P(s,(a_1,a_2,...a_m)|t) \\ &= \sum_{a_1=0}^{l} ... \sum_{a_m=0}^{l} \lambda(m,l) \prod_{i=1}^{m} \pi(a_i,i,m,l).\tau(s_i,t_{a_i}) \\ &= \lambda(m,l).\prod_{i=1}^{m} \sum_{j=0}^{l} \pi(j,i,m,l).\tau(s_i,t_j) \text{ [3]} \quad (2) \end{aligned}$$

The model considers (λ, π, τ) as its parameters, which are statistically learned via an Expectation-Maximization algorithm as follows.

Expectation-Maximization Algorithm [3]. Figure 6 illustrates the training algorithm for the alignment. The algorithm takes as input a training corpus T and returns the model parameters (λ, π, τ). T contains the pairs (s,t) of the sentences for the corresponding methods. It first randomly initializes the parameters (line 2). Then, it performs a loop for training (lines 3–20) in which each iteration has two steps. In the expectation step (lines 4–13), it uses the existing (estimated) model parameters (λ, π, τ) to infer the expected alignments for all pairs of sequences (s,t) in the corpus. Then, in the maximization step (lines 14–19), the counting values collected from those alignments are used to re-estimate the parameters.

Specifically, at an iteration, with the current parameters, for a given pair of sequences (s,t), the probability that s_i is aligned to t_j is proportional to $p(i,j) = \pi(j,i,m,l).\tau(s_i,t_j)$ (line 10). Thus, the probability that $a_i = j$ is $\delta(i,j) = p(i,j)/\sum_{j=0}^{l} p(i,j)$ (line 11). $\delta(i,j)$ is the expected number of the alignments of a symbol u at position i of s (with the length m) to a symbol v at position j of t (with the length l). Thus, it is added to the counting values $\text{count}(j,i,m,l)$, $\text{count}(i,m,l)$, $\text{count}(u,v)$, and $\text{count}(v)$, which are the occurrence counts on the alignments. For example, $\text{count}(j,i,m,l)$ is for the aligned sequences of the lengths m and l in which the positions i and j are aligned with the current parameters. Similar meaning is for other counting values. After counting for all sequence pairs in the corpus, those values are used to re-estimate the parameters:

$$\lambda \simeq \frac{\text{count}(m,l)}{\text{count}(l)}, \tau(u,v) \simeq \frac{\text{count}(u,v)}{\text{count}(v)}, \pi(j,i,m,l) \simeq \frac{\text{count}(j,i,m,l)}{\text{count}(i,m,l)}$$

Then, $P(s,a|t)$ is computed by (1) and $P(s|t)$ is by (2).

Table 2: Subject Systems

Project	Java			C#			M.Meth
	Ver	File	Meth	Ver	File	Meth	
Antlr (AN) [2]	3.5.0	226	3,303	3.5.0	223	2,718	1,380
db4o (DB) [7]	7.2	1,771	11,379	7.2	1,302	10,930	8,377
fpml (FP) [9]	1.7	138	1,347	1.7	140	1,342	506
Itext (IT) [14]	5.3.5	500	6,185	5.3.5	462	3,592	2,979
JGit (JG) [16]	2.3	1,008	9,411	2.3	1,078	9,494	6,010
JTS (JT) [17]	1.13	449	3,673	1.13	422	2,812	2,010
Lucene (LC) [21]	2.4.0	526	5,007	2.4.0	540	6,331	4,515
Neodatis (ND) [25]	1.9.6	950	6,516	1.9b-6	946	7,438	4,399
POI [32]	3.8.0	880	8,646	1.2.5	962	5,912	4,452

5.2 Sequence-to-Sequence Alignment

For sequence-to-sequence alignment, we adopted an extension of symbol-to-symbol alignment called *phrase-based model* [18, 19]. The extended model expands the surrounding symbols of the aligned symbols to get larger aligned sequences, i.e., phrases. The steps for computing the sequence-to-sequence alignments include:

1. The model adds the pairs of symbols that were aligned by the symbol-based alignment model (Section 5.1) into a *sequence mapping table* with their mapping probabilities;

2. It collects all sequence pairs that are consistent with the symbol alignment, i.e., the sequence alignment has to contain all alignments for all covered symbols. A sequence pair (s,t) is consistent with a symbol alignment a, if all symbols s_1, ..., s_k in s that have alignment points in a have these with symbols t_1, ..., t_k in t and vice versa [18]. A sequence pair is required to include at least one alignment point.

3. It iterates over all target sequences to find the ones closest to source sequences, and adds those sequence pairs and mapping probabilities to the sequence mapping table.

Finally, we perform a filtering step. The sequence mapping table contains all pairs of sequences with their scores. The pairs with scores higher than a threshold are kept. Moreover, the pairs with both Java and C# usage sequences that are included in other usage sequences with higher scores are not reported because the pairs with enclosing sequences are already reported. In contrast, all non-inclusion usage sequence pairs are kept. The sequences containing only control structures (i.e., for, if) are discarded.

6. EMPIRICAL EVALUATION

Based on the aforementioned algorithms, we have built our API usage mining tool, StaMiner. For the phrase-based alignment, we adapted Phrasal [4], a popular toolkit for statistical machine translation. Then, we conducted an empirical evaluation on

1. StaMiner's accuracy in mining *API usage mappings*;
2. Its accuracy in mining *API mappings* in comparison with the state-of-the-art approach MAM [39]; and
3. The usefulness of its mined API mappings in code migration.

All experiments were conducted on a computer with AMD Phenom II X4 965 3.0GHz, 8GB RAM, and Linux Mint.

For comparison, we used the same data set as in the prior research, MAM [39], on mining API mappings (a couple of subject projects are not available anymore, thus, we replaced them with the new projects). They are open-source projects which were originally developed in Java and then ported to C# (Table 2). They are well-established systems with long histories and both of their Java and C# versions have been in use. Columns Java.Ver and C#.Ver show the corresponding versions in the two languages. We confirmed the corresponding versions by reading the associated documentation and code in each project. Columns File and Meth show the numbers of files and methods in each version for each language.

First, we built the *corpus* of the pairs of corresponding client code methods in the two languages. That corpus is used as the *input* of our tool. To build it, we followed the same procedure as described in Section 4.1. of the MAM paper [39]: basically, an automated tool was built to search for the same method signatures of the classes with similar/same names (with possible uppercase/lowercase differences) in similar directory structures in two languages. Such pairs of methods in the client code are likely to perform the same functionality. Note that this procedure was aimed to align client code methods that formed the input corpus for our tool. We manually verified a randomly selected sample of 500 resulting pairs of methods (1.5%). We followed the sampling procedure to have a statistical confidence level of 95% on the corresponding method pairs. All of the sampled pairs are correct. Since in a subject project, the corresponding versions include different supporting and utility packages in two languages, there are methods in both Java and C# versions that do not have the corresponding ones, hence, they were not used. Column M.Meth shows the number of aligned methods among two versions.

6.1 Accuracy in Mining API Usage Mappings

We conducted our first experiment to evaluate StaMiner's accuracy in mining the mappings of API usages from the client code in the subject projects. We ran StaMiner on all projects in Table 2. We used the default threshold $\Delta = 0.5$ in Phrasal [4] for the scores of sequence pairs/mappings in the sequence mapping table (SMT). A score of a mapping (s,t) is computed as the ratio $n(s,t)/(n(s)+1)$ where $n(s,t)$ is the number of mapping occurrences (s,t) and $n(s)$ is the number of occurrences of the sequence s. The mappings in SMT that have scores greater than $\Delta = 0.5$ were included in the result. The rationale is that if s is observed, t is observed as the corresponding sequence of s in more than 50% of the time.

Table 3 shows our result. The column Tot shows the total number of API usage mappings in the SMT after filtering. For the resulting mappings, we examined the Java usage sequences and counted the mappings that have different numbers of API elements and control units (e.g., for, while, if): one (column **1**), two to three (**2-3**), four to seven (**4-7**), and eight or more (**8+**). We also counted the mappings having different numbers of variables involved in the usages (columns **1, 2, 3**, and **4+** for one, two, three, and four or more variables). Note that, the numbers of variables involved in the usages are different from those of API elements because the API elements also include static method calls, and because a variable can be involved with multiple API elements (classes/methods).

We measured StaMiner's accuracy on mining API usage mappings based on two metrics. The first one is correctness. We followed the same procedure as in MAM [39] to randomly select 50 mined API usage mappings for each project. The number of the selected mappings in each group (a group of sequence size of 1, 2-3, 4-7, or 8+ API elements) is proportional to the size of that group. We manually examined each mined API usage mapping and classified it as correct or incorrect by inspecting API client code and API documentation. A mapping is considered as correct if all API elements and control units in both source and target usage sequences are correct and support the same task. Column Corr. shows the percentage of correct mappings over the number of checked mappings.

Our second metric is called *edit distance ratio (EDR)*. This metric measures the number of API elements that a user must delete/add in order to transform a resulting usage into a correct one according to API documentation. It is computed as EDR = $\frac{\sum_{pairs} EditDistance(s_R,s_T)}{\sum_{pairs} length(s_T)}$, where $EditDistance(s_R,s_T)$ is the *editing distance* between each

Table 3: Accuracy in Mining API Usage Mappings

Proj		# of Java APIs				# of Java Vars				Corr.	EDR
	Total	1	2-3	4-7	8+	1	2	3	4+		
AN	1,202	26	18	5	1	45	3	1	1	86.0%	7.7%
DB	19,562	5	22	15	8	24	15	9	2	92.0%	4.1%
FP	913	32	14	2	2	44	6	0	0	90.0%	6.2%
IT	9,918	36	12	2	0	45	3	1	1	90.0%	9.2%
JG	32,621	32	13	4	1	41	6	1	2	86.0%	7.9%
JT	1,139	24	17	9	0	40	6	4	0	86.0%	7.8%
LU	16,206	22	19	8	1	41	6	2	1	86.0%	6.5%
NE	11,864	22	13	12	3	34	3	5	8	94.0%	2.4%
PO	7,400	31	14	5	0	44	5	0	1	78.0%	10.6%
All	100,825	220	137	62	16	357	51	18	9	**87.1%**	7.3%

pair of the *correct sequence* s_R and the resulting sequence s_T in C#; and the denominator is the total numbers of symbols of all resulting sequences in C#. The smaller an EDR value is, the closer the resulting sequence to the correct one. There are also 27 *incorrect usage sequences* s_R in Java for which EDR was not defined/computed. A usage sequence is incorrect if it does not represent a correct API usage. Row All lists the total result after *duplicate mappings* across all projects were removed. Thus, the numbers of mappings at row All might be smaller than the total numbers of mappings at all the rows.

As seen, StaMiner was able to mine a large number of API usage mappings with a relatively high accuracy (87.1%). On average, users have to edit 7.3% of the API elements in the resulting API usage mappings to correct them. Accuracy can be as high as 94%, and EDR can be as low as 2.4%. Many of the resulting API usage mappings involve a single API class/method (220 out of 435 mappings) and a single variable in Java (357 out of 435) [1]. However, there are many other resulting mappings involving multiple API classes/methods (215 out of 435) and multiple variables (78 out of 435).

6.2 Characteristics of API Usage Mappings

In this experiment, we aimed to study the characteristics of the mappings between API usages in Java and C#, e.g., how many of them are one-to-one or many-to-many, how many of them involve multiple variables, etc. The answers to those questions would be an indicator for the need of StaMiner in mining API usage mappings.

Arities of API Usage Mappings. Among 435 API usage mappings that we checked (note that 15 duplicate ones across projects were removed), we kept 379 correct API usage mappings for this study (i.e., 87.1% of them). The incorrect mappings were not used in this experiment. Among 379 correct API usage mappings, many of them (216/379 = 57%) are 1-to-1 API mappings with classes, methods, or fields. 133 out of 379 are *n*-to-*n* (*n*>1) API mappings, i.e., they can be broken into 1-to-1 mappings. However, 30 out of 379 (7.9%) are general *n*-to-*m* mappings with $m \neq n$. That is, there are API usages in Java that can be mapped to the equivalent ones in C# with different numbers of API elements. These results show that 1-to-1 API mappings are more popular, however, general API usage mappings are also important. Thus, StaMiner is useful since it is able to mine 1-to-1 API mappings as well as the mappings of entire API usages (i.e., general *m*-to-*n* mappings with $m \neq n$). Moreover, this result calls for the new rule-based code migration tools that can operate on such general API usage mappings. The state-of-the-art migration tools operate only with rules for 1-to-1 API mappings [15]. Currently, even without such migration tools yet, our mined *m*-to-*n* API usage mappings would assist developers in manually migrating their code for those usages.

[1]Among 450 mappings that we checked, there are 15 duplicate mappings across all nine projects.

Table 4: Different Numbers of Variables Involved in 379 Correct API Usage Mappings

		No of Vars in C# Usages			
		1	2	3	4+
No of	1	297	5	0	0
Vars	2	12	37	1	0
in Java	3	2	5	11	0
Usages	4+	0	0	3	6

In addition, the majority of API usage mappings involves the usages with the sizes between 1-7 (366/379=96.5%). There are nine cases in all the projects that have usages involving 8-12 API elements and control units (for, if, etc.). We examined them and found that those usages are the ones for traversal on complex tree data structures and database tables. For example, the following usage with length of 7 in Neodatis is used to iterate over a BTree data structure to check for a certain key and to retrieve the parent's node(s):

IBTreeNode.getNbKeys _IF_ AbstractBTreeIterator.getCurrentNode _WHILE_ AbstractBTreeIterator.getCurrentNode AbstractBTreeIterator.getChildNode IBTreeNode.getParent in Java becomes

IBTreeNode.GetNbKeys _IF_ AbstractBTreeIterator.currentNode _WHILE_ AbstractBTreeIterator.currentNode AbstractBTreeIterator.childNode IBTreeNode.GetParent in C#.

Numbers of Involved Variables. Table 4 shows the numbers of API usage mappings in which the usages in Java and the corresponding ones in C# involve certain numbers of variables. As seen, a large percentage of usage mappings involve the usages with a single object in both languages (297/379 = 78.4%). These include 216 mappings of a single API element. That is, there are 81 mappings of a single object but with multiple API elements and control units. There are 351 out of 379 (92.61%) mappings that have the same numbers of variables in two languages. An example of cases with different numbers of variables is when a temporary variable is used such as the variable keyList in line 4 of Figure 1b. Among 28 mappings whose usages have different numbers of variables in the two languages, there are 17 of them with the usages having one-to-two or two-to-one variable mappings when being migrated from Java to C#. For example, in db4o, StaMiner found that

Iterator4.moveNext _WHILE_ Iterator4.current in Java becomes

IEnumerator.MoveNext _WHILE_ ExpectingVisitor.Visit in C#.

In this case, instead of using an Iterator object in Java, the developers use two C# objects of IEnumerator and ExpectingVisitor to traverse a data structure in db4o. There are also 7 of them with the usages having 3-to-1 or 3-to-2 variable mappings. Moreover, there are a few mappings of the usages with four variables (Table 4), e.g.,

Collection.iterator Iterator.hasNext _FOR_ Iterator.next Geometry.getCoordinates GeometryFactory.createPolygon Geometry.getUserData in Java becomes

Collection.Iterator Iterator.HasNext _FOREACH_ Geometry.Coordinates GeometryFactory.CreatePolygon Geometry.UserData in C#.

Examples. StaMiner is able to mine the mappings in which an API element is migrated to another one of *a different kind* (e.g., the mappings in Table 1). As another example, in the POI project:

System.getProperty("line.separator") in Java becomes

Environment.NewLine in C#

because in C#, developers retrieved the system's code for a line separator by reading the NewLine property of an Environment object, while in Java, getProperty is used. StaMiner can also handle the cases where the names of APIs or the methods' signatures are different. For example, in the Antlr project, StaMiner found a usage:

Map.get (String) in Java becomes

IDictionary.TryGetValue(String,int) in C#.

Table 5: Time Efficiency

Proj	M.Meth	Avg. Groum Size		Max. Groum Size		Time
		Java	C#	Java	C#	(mins)
Antlr	1,380	9.0	8.6	232	149	23
db4o	8,377	6.3	5.9	146	132	62
fpml	506	8.0	7.6	136	102	4
Itext	2,979	12.0	10.4	710	467	27
JGit	6,010	14.2	13.7	270	217	86
JTS	2,010	7.3	6.8	92	92	18
Lucene	4,515	13.5	13.1	331	310	56
Neodatis	4,399	9.5	9.7	383	447	55
POI	4,452	9.6	8.7	299	552	43

6.3 Time Efficiency

We also measured StaMiner's mining time. In Table 5, the column M.Meth shows the number of the mapped methods in the client code. The columns under Avg.Groum Size and Max.Groum Size show the average and maximum sizes, respectively, in terms of nodes of the Groums built for the methods in the subject projects in Java and C#. The column Time shows the running time in minutes.

As seen, StaMiner was able to run on the large projects with up to 8,377 methods. A graph for each method in Java has the average and maximum sizes of up to 14.2 and 710 nodes, respectively. The respective sizes for the methods in C# are 13.7 and 552 nodes. Generally, for all methods in a project, there are up to 78,270 nodes for all Groums. StaMiner's running time is within a couple of hours for the large projects (e.g., 62 minutes for db4o with 8,377 methods and 86 minutes for JGit), and about a few minutes for the smaller projects (e.g., 4 minutes for fpml with 506 methods).

6.4 Accuracy Comparison in Mining Single API Mappings

We conducted another experiment to evaluate how well StaMiner performs in mining API mappings in comparison with the state-of-the-art API mapping mining tool, MAM [39]. Note that, although StaMiner can mine the mappings for entire API usages, in this study, we aimed to compare its accuracy on single (1-to-1) API mappings.

6.4.1 Experimental Setting and Measurement

We re-implemented MAM based on the description of its paper [39]. We then compared the resulting API mappings mined from both StaMiner and MAM against a mapping data set consisting of the *manually written API mappings that are provided as an external data file of Java2CSharp* [15], an existing semi-automatic migration tool. That is, Java2CSharp has a set of manually written API mappings of APIs in Java and C#. Details of the file format can be found in [15, 39]. The mapping files are associated with several packages defined by J2SE APIs and 2 packages defined by JUnit. We used the most up-to-date data files in Java2CSharp. We used the same packages that were used in MAM paper for the comparison [39]. However, the latest data files in Java2CSharp no longer contain the API mappings for the library java.text. Thus, we removed that package in our evaluation. The mappings for the remaining 8 packages (column package in Table 6) in the mapping files were used as the oracle (golden standard) in this experiment.

To compare the results mined from MAM and StaMiner, we ran each of them on the subject projects (Table 2) to produce the output API mappings. After running MAM, we followed the procedure described in its paper [39] (page 202) to produce the resulting API mappings. Finally, as described in their paper, we ignored the API mappings in the oracle that do not have call sites in the subject projects because both approaches rely on the call sites to mine the mappings. At the end, we had 343 mappings in the oracle. After running StaMiner, we processed the sequence mapping table as follows. For comparison, all single API mappings whose scores are higher than Δ were kept. For each API method/class, we chose its resulting mapping in C# with the highest score. The sequences that had more than one symbol or did not belong in those packages in the oracle were discarded in this experiment. The mappings of method calls and field accesses were kept for the resulting mappings of API methods and fields. The class names extracted from the action nodes of the mapped ones were used to form the resulting mappings for API classes. Then, we compared those resulting mappings for API classes/methods against the oracle.

We used the traditional precision, recall, and F-score for evaluation. They are defined as follows. $Precision = TP/(TP+FP)$, $Recall = TP/(TP+FN)$, and $F\text{-}score = 2*(Precision*Recall)/(Precision+Recall)$ where
TP is true positive count, which is the number of the API mappings that are in both the mined API mappings and the oracle. FP is false positive count, which is the number of the API mappings that exist in the mined mappings but not in the oracle. We manually checked false positive mappings. If they are true mappings, we consider them as newly found ones and removed them from the false positive set. FN is false negative count, which is the number of API mappings that are in the oracle but not in the mined API mapping set.

6.4.2 Results

Table 6 shows the result on the mappings for API classes and methods. The columns Pre, Rec, and F-score display precision, recall, and F-score values for each package, respectively. The row All shows the result for all packages. As seen, for all packages, StaMiner achieves reasonably high precision (74.4% for classes and 77.5% for methods), recall (72.9% for classes and 65.6% for methods), and F-score (73.6% for classes and 71.1% for methods).

Compared to MAM, for individual packages, precision, recall, and F-score values for both API classes and methods are higher than those of MAM. Specifically, for all packages, StaMiner has *higher recall* in API class mappings (16.2% in absolute percentage and **28.6%** relatively) and in API method mappings (5.3% in absolute percentage and 8.8% relatively). For all packages, StaMiner also achieves *higher precision* than MAM in API class mappings (9.7% in absolute percentage and **15.0%** relatively) and in API method mappings (11.3% in absolute percentage and **17.1%** relatively).

Generally, for all packages, StaMiner has higher F-scores than MAM: 13.2% in absolute percentage and **21.9%** relatively for API class mappings, and 8.0% in absolute percentage and **12.7%** relatively for API method mappings. For individual packages, it improves over MAM with up to **22.2%** and **10.7%** in absolute percentages in F-score in mining API classes and methods, respectively.

6.4.3 Newly Found API Mappings

Interestingly, we also found that StaMiner correctly detected a total of **120 new API mappings** of API classes and methods that were not manually written in the latest mapping files in Java2CSharp. Some cases with different syntactic types and names are listed in Table 7. MAM can only detect 25 new mappings. MAM did not detect the cases in Table 7 and Table 1 because the API class/method mappings do not have similar names or do not have the same kind (e.g., a method call becomes an array access). Those newly found mappings are correct and could be added to complement the data files of Java2CSharp. Detailed results can be found on [41].

The method calls with the same names but with different numbers and data types of parameters were a source for MAM's inaccuracy. StaMiner is able to handle those cases. However, it could miss the cases where the mappings did not occur frequently enough.

Table 6: Accuracy Comparison with MAM [39]

Package	API Class Mappings						API Method Mappings					
	Precision		Recall		F-score		Precision		Recall		F-score	
	MAM	StaMiner	MAM	StaMiner	MAM	StaMiner	MAM	StaMiner	MAM	StaMiner	MAM	StaMiner
java.io	66.9%	70.0%	54.5%	63.6%	60.1%	66.6%	60.0%	70.0%	54.5%	64.0%	57.1%	66.9%
java.lang	63.4%	82.5%	60.5%	76.7%	61.9%	79.5%	71.7%	86.7%	69.6%	76.5%	70.6%	81.3%
java.math	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%	66.7%	66.7%	66.7%	66.7%	66.7%	66.7%
java.net	100.0%	100.0%	50.0%	50.0%	66.7%	66.7%	100.0%	100.0%	33.3%	33.3%	50.0%	50.0%
java.sql	100.0%	100.0%	50.0%	50.0%	66.7%	66.7%	100.0%	100.0%	50.0%	50.0%	66.7%	66.7%
java.util	61.5%	64.7%	51.6%	71.0%	56.1%	67.7%	54.4%	63.0%	53.1%	54.8%	53.7%	58.6%
jUnit	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
orw.w3c	60.0%	80.0%	75.0%	100.0%	66.7%	88.9%	76.9%	84.6%	41.6%	45.8%	54.0%	59.4%
All	64.7%	74.4%	56.7%	72.9%	60.4%	73.6%	66.2%	77.5%	60.3%	65.6%	63.1%	71.1%

Table 7: Some newly found API mappings from Java to C# that were not in Java2CSharp's manually written mapping data files

	API in Java	Corresponding API in C#
1	java.util.Locale.getDefault()	System.Globalization.CultureInfo. CurrentCulture
2	java.lang.Class.getSuperclass()	System.Type.BaseType
3	java.nio.charset.Charset.forName(String)	Sharpen.Extensions.GetEncoding(string)
4	java.util.Collections.unmodifiableList (ArrayList)	System.Collections.ReadOnly (ArrayList)
5	java.lang.Math.random()	System.Random.NextDouble()
6	java.lang.Number.floatValue()	System.IConvertible.ToSingle(...)

Table 8: Subject Systems for Code Migration

Project	Java			C#			M.Meth
	Ver	File	Meth	Ver	File	Meth	
rasp (RA)	1.0.1	315	1,819	1.0.1	556	1,893	739
llrp (LL)	1.0.0.6	255	3,833	1.0.0.5	222	975	545
SimMetrics (SI)	1.6.2	106	581	1.5	65	327	143
aligner (AL)	1.0	41	232	0.2	19	51	30
fit (FI)	1.1	59	461	1.1	39	285	254

Table 9: Quality of Migrated Code

Proj	Compilation Errors				Defects			
	MF	MAM	StaMiner	Improv	MF	MAM	StaMiner	Improv
RA	191	176	166	13.1%	123	101	88	28.5%
LL	238	235	235	1.3%	114	114	114	0.0%
SI	13	9	9	30.7%	0	0	0	0
AL	25	25	25	0.0%	0	0	0	0
FI	99	97	84	15.2%	0	0	0	0
All	566	542	519	**8.3%**	237	215	202	**14.8%**

6.5 Usefulness in Code Migration

We conducted another experiment to evaluate StaMiner's usefulness in assisting an existing migration tool, Java2CSharp [15]. We used the API mappings mined from StaMiner in addition to the mapping data files in Java2CSharp in this experiment, and then compared to the case of the addition of MAM's result to Java2CSharp's data files. We ran Java2CSharp with the complemented mapping data files on 5 new subject projects listed in Table 8, which were also used in MAM paper [39]. Those 5 subject projects were used only for code migration, while the other ones in Table 2 were used for mining API mappings. The process to collect the data and the notations of Table 8 are the same as those for Table 2. We then compared the quality of the migration code when different mapping data files were used. The respective C# code of the subject projects used for migration is considered as the oracle (i.e., correctly migrated code). Let us call the respective C# files in the oracle the *reference set*.

We used the same procedure and metrics as in MAM paper [39] to evaluate the quality of the migrated code. The two metrics are the numbers of compilation errors and defects in the migrated code. Defects refer to semantic errors after code was compiled. There are compilation errors and defects that are not relevant to APIs. Thus, we manually inspected the migrated code for the new five subject projects to count the API-related defects. API-related defects and compilation errors occur when Java2CSharp did not migrate API elements in Java to its respective ones in C#. To save effort, for each project, we chose to inspect the top five largest files in the resulting C# files that have the corresponding files in the C# version of the project. We carefully inspected those resulting C# files and their differences with the reference files to detect API-related defects and compilation errors. A typical API-related error occurs when Java2CSharp did not migrate an API in Java to C#, e.g., the resulting C# code still contains java.lang.Math.random() in Java.

Table 9 shows the results. The columns under Error and Defect display the numbers of compilation errors and API-related defects found, respectively. Column MF lists the result when only the mapping file of Java2CSharp was used. Column MAM shows the result when the combination of the mapping file of Java2CSharp and the one produced from MAM's result was used. Column StaMiner lists the result when the combination of the Java2CSharp's mapping file and the one created from StaMiner's result was used. Column Improv shows relative improvement in percentages over MF.

With newly added API mappings, StaMiner is able to assist Java2CSharp to **reduce 47 compilation errors** (i.e., **8.3%** relatively) and **35 defects** (i.e., **14.8%** relatively). In comparison to MAM, StaMiner is able to improve over MAM by **further reducing 23 compilation errors** (i.e., **4.2%** relatively) and **13 defects** (i.e., **6.0%** relatively). The projects SimMetrics and fit used the libraries that are just partially supported by the mapping files in Java2CSharp and thus, StaMiner is able to reduce 4+15 compilation errors. Project aligner has all of its needed APIs supported by Java2CSharp, thus, StaMiner did not reduce the numbers of errors and defects. It did not improve much in the case of the llrp project since it uses jdom and log4j libraries, and our training data in Table 2 contains no usage of the APIs in those libraries. In contrast, StaMiner performed better than MAM in the rasp project given the same subject projects for mining. Overall, StaMiner helps Java2CSharp reduce 13 more defects and 23 compilation errors than MAM in the migrated code.

6.6 Limitations and Discussion

We examined the incorrect cases, and determined the limitations and different sources of inaccuracy in StaMiner. First, it is a data-driven approach, which depends on the given data for mining. It is not able to mine the mappings for the APIs that do not appear in the given data. That is a source of inaccuracy in the results in Tables 6 and 9. The MAM approach faces the same issue. However, with the support by an automatic tool and minimum manual effort in collecting 34,628 respective methods in 9 projects in Table 2, StaMiner is able to statistically learn a large number of correct API usage mappings, which were useful in assisting a rule-based migration tool to reduce compilation errors and defects (see Section 6.5). Thus, it provides a good starting set of API mappings for code migration, and developers do not have to manually start from scratch. Moreover, with the same given data in our evaluation, it achieved higher levels of mining accuracy and usefulness than MAM.

Second, our light-weight technique to find the data dependencies also causes inaccuracy. Thus, a usage was broken down into smaller ones due to missing dependencies, leading to incorrect alignment from Java sequences to C# ones. Third, unresolved types lead to Groums' nodes with incorrect labels. Incorrectly resolved types in polymorphic calls occur due to our static analysis. Thus, they cause incorrect alignments. Fourth, there is also room for improvement in sequence-to-sequence alignment. During expanding from the already-mapped symbols, StaMiner tends to include an additional irrelevant or duplicate symbol to create an incorrect API usage. We plan to improve the extension algorithm and add a post-processing phase to handle irrelevant symbols. Fifth, we use only sequence alignment, thus, we have to build sequences from graphs. A drawback is that some different subgraphs might have the same sequence. We are investigating statistical graph-based alignment. Sixth, the current results are in the sequence form only, but resulting graphs can be created since we store traces between sequences and subgrpums. Finally, StaMiner is currently limited to only Java-to-C# due to our Groum representation for OO languages and due to the limitations of sequence alignment. However, StaMiner can be generalized for new languages if one could define the representations for the usages or for the abstraction level that needs the mappings.

6.7 Threats to Validity

Our subject projects might not be representative, but for comparison, we used the same data set as in MAM [39] and replaced a couple of them since they are not publicly available any more. Thanks to the authors of MAM, the links to their subject projects are available in MAM website. However, the tool itself is not available. Our re-implementation follows its description in the paper. We currently compared only with MAM, and will compare StaMiner with other tools. In our last experiment, we used Java2CSharp, and will use other tools. In our experiments, we manually verified a sample set of respective methods and the API usage mappings and defects in the migrated code. Thus, human errors and sampling can affect our results. For comparison in migration, we used the same 5 subject projects as in MAM [39]. They might not be representative and the result could be affected. For example, they might use or not use certain APIs that does not appear in the given data. However, for comparison, both StaMiner and MAM mined on the same data set, and Java2CSharp with newly API mappings was run on them. Another threat is that we assume client code has correct API usages. We also plan to study the impact of the threshold on the result.

7. RELATED WORK

API Migration. Several approaches have been proposed to mine API mapping rules to support code migration and to adapt API evolution. Zhong *et al.* [39] mine API mappings via API Transformation Graphs. It relies on the heuristics of textual and structural similarities between APIs. Nita and Notkin [30]'s twinning method allows users to specify the changes that migrate a program to using new APIs. In Aura, Wu *et al.* [36] combine call dependency and text similarity analysis to identify change rules for one-replaced-by-many and many-replaced-by-one methods. HiMa [23] is a history-based method to match pairs of framework revisions and aggregate revision-level rules to obtain framework-evolution rules. Although Aura and HiMa are aimed to help migration between different versions of the same library or framework, they could be used for code migration between languages. However, despite their differences in details, they all share the principle that two corresponding APIs and the usages in two languages have textually similar names, and the calling structures and parameters are similar. Unlike those deterministic methods, StaMiner is a data-driven, statistical approach.

Adaptation. As software libraries evolve, client code is required to be adapted. Chow and Notkin [5]'s approach require users' annotations. SemDiff [6] mines API usage changes from the evolution of library itself and other client code. CatchUp [11] records refactorings and then replays them to update client code. Diff-CatchUp [37] recognizes API changes and suggests API replacements based on framework examples. Tansey and Tilevich's approach [33] infers generalized transformation rules from given examples so that application developers use the inferred rules for refactoring. *spdiff* [1] identifies common changes made in a set of files. API developers could use *spdiff* to extract a generic patch and apply it to other client code. SmPL [31, 20] is a domain-specific source transformation language that captures textual patches with a semantic change.

Language Migration. To translate SmallTalk to C, Yasumatsu and Doi [38] create replacement classes realizing the same functions as SmallTalk classes as part of an execution model. Mossienko [24] and Sneed [48] perform code migration from COBOL to Java via transformation rules on syntactic units and via automated language transformation. To migrate a Web application using Active Server Pages to Netscape Server Pages, Hassan and Holt [10] proposed a structure-based approach based on Water Transformations. Van Duersen and Kuipers [34]'s method identifies the potentials to form classes by semi-automatically restructuring existing data structures in the code. Trudel *et al.* [49] migrate C code to Eiffel, taking advantage of type safety and contracts. In Waters [35]'s paradigm, a program is analyzed to obtain a language-independent procedure for a re-implementation in the target language. El-Ramly *et al.* [8] reported that existing rule-based migration tools support only a subset of APIs for code migration. Our prior work uses statistical machine translation to migrate Java to C# code using lexical tokens [27] and syntactic/semantic annotations [28]. In comparison, StaMiner mines the API mappings and used them in a deterministic migration tool, Java2CSharp. Our preliminary result appears in a poster [29].

Example code searching is useful in language migration. Strathcona [12] extracts the *structural context* of the code under editing and finds its relevant examples. MAPO [40] mines and indexes API usage patterns and recommends code examples. Hindle *et al.* [13] used *n*-gram statistical model [22] to support code suggestion.

8. CONCLUSIONS

We introduce StaMiner, a novel data-driven approach that statistically learns the mappings between API usages from the corpus of the corresponding client code of the APIs in two languages Java and C#. Instead of using heuristics on the textual or structural similarity between APIs in two languages as in existing mining approaches, StaMiner is based on a statistical model that learns the mappings in such a corpus and provides mappings for entire API usages. Our empirical evaluation on several projects shows that StaMiner can detect API usage mappings with high accuracy than the state-of-the-art approach MAM. With the resulting API mappings mined by StaMiner, Java2CSharp, an existing migration tool, produces code with higher quality in code migration.

9. ACKNOWLEDGMENTS

This project is funded in part by US National Science Foundation grants: CCF-1018600, CNS-1223828, CCF-1349153, CCF-1320578, and CCF-1413927.

10. REFERENCES

[1] J. Andersen and J. Lawall. Generic patch inference. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering* (ASE '08), pages 337–346. IEEE CS, 2008.

[2] Antlr. https://github.com/antlr/.

[3] P. F. Brown, V. J. Della Pietra, S. A. Della Pietra, and R. L. Mercer. The mathematics of statistical machine translation: parameter estimation. *Comput. Linguist.*, 19(2):263–311, MIT Press, June 1993.

[4] 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 *Proceedings of the NAACL HLT 2010 Demonstration Session*, pages 9–12, Association for Computational Linguistics, 2010.

[5] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the 1996 International Conference on Software Maintenance* (ICSM '96), page 359, IEEE CS, 1996.

[6] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering* (ICSE '08), pages 481–490. ACM, 2008.

[7] db4o. http://sourceforge.net/projects/db4o/.

[8] M. El-Ramly, R. Eltayeb, and H. Alla. An experiment in automatic conversion of legacy Java programs to C#. *ACS/IEEE International Conference on Computer Systems and Applications*, pages 1037–1045. IEEE CS, 2006.

[9] fpml. http://sourceforge.net/projects/fpml-toolkit/.

[10] A. E. Hassan and R. C. Holt. A lightweight approach for migrating web frameworks. *Information and Software Technology*, 47(8):521–532, June 2005.

[11] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering* (ICSE '05), pages 274–283. ACM Press, 2005.

[12] R. Holmes and G.C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering* (ICSE '05), pages 117–125. ACM Press, 2005.

[13] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12), pages 837–847. IEEE Press, 2012.

[14] iText. http://sourceforge.net/projects/itext/.

[15] Java2CSharp. http://j2cstranslator.wiki.sourceforge.net/.

[16] JGit. https://github.com/eclipse/jgit/.

[17] JTS. http://sourceforge.net/projects/jts-topo-suite/.

[18] P. Koehn. *Statistical Machine Translation*. The Cambridge Press, 2010.

[19] P. Koehn, F. J. Och, and D. Marcu. Statistical phrase-based translation. In *Proc. of the Conf. of North American Chapter of the Association for Computational Linguistics on Human Language Technology*, NAACL '03, pages 48–54, 2003.

[20] J. L. Lawall, G. Muller, and N. Palix. Enforcing the use of API functions in Linux code. In *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software* (ACP4IS '09), pages 7–12, ACM, 2009.

[21] Lucene. http://lucene.apache.org/.

[22] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*. MIT Press, USA, 1999.

[23] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12), pages 353–363. IEEE, 2012.

[24] M. Mossienko. Automated Cobol to Java recycling. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering* (CSMR '03). IEEE CS, 2003.

[25] NeoDatis. http://sourceforge.net/projects/neodatis-odb/.

[26] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* (ESEC/FSE '09), pages 383–392. ACM, 2009.

[27] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Lexical Statistical Machine Translation for Language Migration. In *Proceedings of the 9th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* (ESEC/FSE '13), NIER Track. ACM, 2013.

[28] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Migrating Code with Statistical Machine Translation. In *Proceedings of the 36th International Conference on Software Engineering* (ICSE '14), Demo Track, pages 544–547. ACM, 2014.

[29] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, T. N. Nguyen. Statistical Learning of API Mappings for Language Migration. In *Proceedings of the 36th International Conference on Software Engineering* (ICSE '14). ACM, 2014.

[30] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (ICSE '10), pages 205–214. ACM, 2010.

[31] Y. Padioleau, J. L. Lawall, and G. Muller. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers. *Electronic Notes Theoretical Computer Science*, 166:47–62, 2007.

[32] POI. http://poi.apache.org/.

[33] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications* (OOPSLA '08), pages 295–312. ACM, 2008.

[34] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st international conference on Software engineering* (ICSE '99), pages 246–255. ACM, 1999.

[35] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. Softw Eng.*, 14(8):1207–1228, 1988.

[36] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd International Conference on Software Engineering* (ICSE '10), pages 325–334. ACM, 2010.

[37] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Trans. Softw. Eng.*, 33(12):818–836, 2007.

[38] K. Yasumatsu and N. Doi. SPiCE: A system for translating Smalltalk programs into a C environment. *IEEE Trans. Softw. Eng.*, 21(11):902–912, Nov. 1995.

[39] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang.

Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (ICSE ’10), pages 195–204. ACM, 2010.

[40] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European Conference on Object-Oriented Programming* (ECOOP ’09), pages 318–343. Springer-Verlag, 2009.

[41] StaMiner. http://home.engineering.iastate.edu/~anhnt/Research/StaMiner/.

[42] Sharpen. http://community.versant.com/Projects/html/projectspaces/db4o_product_design/sharpen.html.

[43] DMS. http://www.semdesigns.com/Products/DMS/DMSToolkit.html.

[44] Java to C# Converter. http://www.tangiblesoftwaresolutions.com/Product_Details/Java_to_CSharp_Converter.html.

[45] XES. http://www.euclideanspace.com/software/language/xes/userGuide/convert/javaToCSharp/.

[46] Octopus.Net Translator. http://www.remotesoft.com/octopus/.

[47] Microsoft Java Language Conversion Assistant. http://support.microsoft.com/kb/819018.

[48] H. M. Sneed. Migrating from COBOL to Java. In *Proceedings of the 26th IEEE International Conference on Software Maintenance* (ICSM ’10), pages 1–7, IEEE CS, 2010.

[49] M. Trudel, C.A. Furia, M. Nordio, B. Meyer, M. Oriol. C to O-O Translation: Beyond the Easy Stuff. In *Proceedings of the 19th IEEE Working Conference on Reverse Engineering* (WCRE ’12), pages 19-28. IEEE CS, 2012.