Mining API Mapping for Language Migration

Hao Zhong[1], Suresh Thummalapenta[4], Tao Xie[4], Lu Zhang[2,3], Qing Wang[1]
[1]Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing, 100190, China
[2]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China
[3]Institute of Software, School of Electronics Engineering and Computer Science, Peking University, China
[4]Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA
zhonghao@itechs.iscas.ac.cn, {sthumma,xie}@csc.ncsu.edu, zhanglu@sei.pku.edu.cn, wq@itechs.iscas.ac.cn

ABSTRACT

In the long history of programming languages, researchers and practitioners created various languages such as C# and Java. These languages have their own benefits and thus coexist in various projects. To develop projects in multiple languages, programmers need various manipulations such as translating projects from one language into other languages. Automating this translation process can reduce substantial manual effort. However, it is challenging to map APIs of different languages correctly in such automation. In this paper, we propose a novel approach that mines API mapping automatically. For two languages, our approach needs a set of projects with versions in the two languages. From each project, our approach analyzes its two versions and mines API mapping of the two languages based on API usages exhibited in the two versions. The mined API mapping describes mapping relations of APIs in different languages and thus aids language migration. Based on our approach, we implemented a tool named MAM (**M**ining **A**PI **M**apping) and conducted an evaluation of our approach based on the tool. The results show that our approach mines various API mapping between Java and C# with reasonable accuracies. The results also show that mined API mapping reduces 54.4% compilation errors in translated projects of the CSharp2Java tool as mined API mapping contains mapping relations that do not exist in language migration tools such as CSharp2Java.

1. INTRODUCTION

In the history of software programming, researchers and practitioners created various programming languages. In particular, the HOPL[1] website lists 8512 different languages. Due to various considerations such as attracting programmers with various backgrounds and achieving better performances on particular platforms, a project may be implemented as versions of multiple languages. For example, many well-known projects such as Lucene[2], Db4o[3], and WordNet[4] all provide versions of multiple languages. For those open source projects, even if a project does not provide any versions of multiple languages, outside programmers may implement versions of particular languages for the project. For example, although WordNet officially does not provide a C# version, Simpson and Crowe developed WordNet.Net[5] for C# programmers. Another example is iText[6]. Although it provides only a Java version, Kazuya developed iText.Net[7] for C# programmers. In fact, as pointed out by Jones [6], about 1/3 of existing software projects have versions of multiple languages.

To reduce the effort of programming, a natural way to develop a version of a new language for a project is to translate from an existing version of the project. To help programmers conduct such migration, researchers proposed various approaches [3, 7, 16]. However, language migration is still quite high risk even with the help of those proposed approaches. For example, Terekhov and Verhoef [10] stated that at least three companies went bankrupt and another company lost 50 million dollars all because of failed language migration projects.

One main challenge of language migration is to map APIs from one language into other languages automatically. It is non-trivial to build API mapping as libraries typically provide huge APIs and many mapping relations of APIs are not straightforward. Consequently, existing approaches [3, 7, 16] support only a subset of APIs or even ignore the mapping relations of APIs. Such a limitation causes many compilation errors in migrated projects and limits their applications in practice. In this paper, we propose an approach that mines API mapping from API client code automatically. As the mined API mapping describes mapping relations of APIs, our approach can improve existing language migration approaches. This paper makes the following main contributions:

- We propose the first approach that mines API mapping of different languages from existing client code automatically. The mined API mapping describes mapping relations of APIs provided by different languages and can aid language migration.
- We implemented a tool named MAM based on our approach and conducted an experiment on ten projects with both Java and C# versions. The results show that our approach mines various mapping relations with reasonable accuracies.
- We further conducted an experiment that migrate five project from Java to C# using the mined API mapping. The results

[1]`http://hopl.murdoch.edu.au/`
[2]`http://lucene.apache.org/`

[3]`http://www.db4o.com/`
[4]`http://wordnet.princeton.edu/`
[5]`http://opensource.ebswift.com/WordNet.Net/`
[6]`http://www.lowagie.com/iText/`
[7]`http://www.ujihara.jp/iTextdotNET/en/`

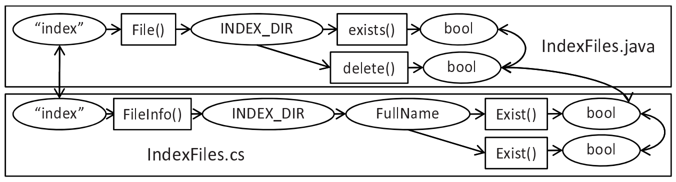


Figure 1: API methods connected by inputs and outputs

show that the mined API mapping reduces the numbers of compile in migrated projects.

The remainder of our paper is as follows. Section 2 illustrates our approach using an example. Section 3 presents definitions. Section 4 presents our approach. Section 5 presents our evaluations. Section 6 discusses issues of our approach. Section 7 presents the related work. Section 8 concludes.

2. EXAMPLE

In this section, we use an example to illustrate challenges of mining API mapping. Suppose that a programmer need to migrate the following code snippet from Java to C# using a translation tool.

```
1  File file = new File("test");
2    if(file.exists()){...}
```

The input of the code snippet is a string that denotes the name of a file or a directory. The output of the code snippet is a boolean value that denotes whether the file or the directory exists. To achieve this functionality, the code snippet declares a local variable whose type is `java.io.File` and calls `exists()` of the local variable. Here, as `exists()` is called through `file`, we call `file` as a receiver of `exists()`. To translate this code snippet, a translation tool needs to know mapping relations of API class to translate the variable `file` to C#. In addition, a translation tool needs to know how to call API methods to use the variable and the input ("`test`") to produce the exactly same output.

As APIs are often large in size, a translation tool often supports only a subset of mapping relations of used APIs. Thus, a translation tool fails to translate the preceding code snippet correctly if the translation tool does not support the preceding two API methods. Some translation tools provide extensions for the programmer to add customized rules for translation. To write a customized rule, a programmer needs to know the mapping relation of APIs. Otherwise, the programmer may choose to learn the mapping relation of APIs from existing samples.

Many projects such as Lucene provide both C# versions and Java versions, and these projects provide various such samples. However, it is not straightforward to learn those mapping relations of APIs. The programmer needs to find Java source files that implement the same functionality with the same input and output. For this example, the programmer may find `IndexFiles.java` in the Java version of Lucene useful as the file satisfies the preceding criteria. The programmer then finds the corresponding source file `IndexFiles.cs` from the C# version of Lucene. The two files are as follows:

```
                    IndexFiles.java
3 public class IndexFiles {
4   static final File INDEX_DIR = new File("index");
5   public static void main(String[] args) {
      ...
6     if (INDEX_DIR.exists()) {...}
      ...
7       INDEX_DIR.delete();
    }
  }
                    IndexFiles.cs
8 class IndexFiles{
9   internal static readonly System.IO.FileInfo INDEX_DIR
          = new System.IO.FileInfo("index");
10   public static void  Main(System.String[] args){
     ...
11    bool tmpBool;
12    if (System.IO.File.Exists(INDEX_DIR.FullName))
13      tmpBool = true;
14   else
15      tmpBool = System.IO.Directory
                        .Exists(INDEX_DIR.FullName);
     ...
   }
 }
```

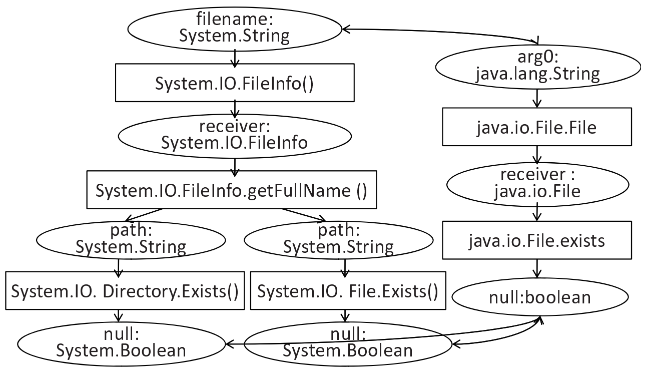


Figure 2: API mapping

It is time-consuming to find out the two files as the two versions of Lucene have more than 1,000 classes totally. Even after the programmer successfully finds out the two aligned files, it is still non-trivial to learn mapping relations of APIs. The programmer first needs to map inputs of the three code snippets. In particular, by comparing Line 1 with Line 4, the programmer knows that `index` is the name for a file or a directory. Consequently, the programmer analyzes how `index` in Line 4 (Java code) and `index` in Line 9 (C# code) are used in the two source files for API mapping of interest so that boolean values are produced for the functionality. To achieve so, the programmer chooses to analyze inputs and outputs of each API method. Figure 1 shows API methods connected by inputs and outputs for the preceding two files. In particular, a box denotes an API method, and an ellipse denotes either an input or an output. The strategy of analyzing inputs and outputs helps find out an API method like `System.IO.Directory.Exists()`. This API method is called in an assignment statement, whereas three other related API methods are called in infix expressions of if-statements. If the programmer relies on call-site structures only, the programmer can miss the API method as its call-site structure is quite different. To match outputs, the programmer must know the mapping relations of classes. For this example, the programmer should know the mapping relation between `boolean` in Java and `System.Boolean`. After the inputs and outputs are mapped, the programmer needs to further match functionalities. For this example, the `delete()` branch of Figure 1 implements a different functionality as indicated by its name, and thus this branch is not mapped with other branches.

The programmer learns mapping relations of APIs from the preceding analysis. However, the analysis is not accurate enough. In particular, Figure 1 does not consider parameters and fails to provide useful information if two API methods have different parameter orders. For this example, as shown in Line 6, the input of `java.io.File.exist()` is a variable, but the inputs of `System.IO.Directory.Exist()` and `System.IO.File.Exist()` are both their parameters as shown in Line 12 and Line 15. The preceding analysis does not consider parameters yet.

In this paper, we propose a novel approach that mines API mapping automatically. Figure 2 show the mined mapping relation of APIs from the preceding two source files. In this figure, a box denotes an API method. Here, our approach uses `getFullName()` to denote the field access of `FullName`. An ellipse denotes a parameter, a variable, or a return value. Each ellipse is named as "*name:type*". Here, our approach uses "variable" for accesses of variables, "null" for return values, and parameter names for parameters.

The mined API mapping has matched inputs, outputs, and how inputs and outputs are connected. Consequently, with the mined API mapping, a translation tool can automatically translate the preceding code snippet into C# as follows:

```
16  FileInfo file = new FileInfo("test");
17  if(System.IO.File.Exist(file.FullName)||
        System.IO.Directory.Exists(file.FullName)){...}
```

In summary, for this example, we find that a programmer need to take tedious and error-prone efforts to find and to analyze source files from two versions of a project for API mapping. We next present our approach to mine API mapping automatically.

3. DEFINITIONS

We next present definitions of the terms used in the rest of the paper.

API: An Application Programming Interface (API) [8] is a set of classes and methods provided by frameworks or libraries.

API library: An API library refers to a framework or library that provides reusable API classes and methods.

Client code: Client code refers to the application code that reuses or extends API classes and methods provided by API libraries. The definitions of API library and Client code are relative to each other. For example, Lucene[8] uses J2SE[9] as an API library, whereas Nutch[10] uses Lucene as an API library. Therefore, we consider Lucene as Client code and API library for the J2SE API library and Nutch, respectively. In general, for Client code, source files of API libraries are often not available.

Mapping relation: A mapping relation refers to a replaceable relation among entities such as API classes or methods defined by two different languages. For example, consider two languages L_1 and L_2, and two entities e_1 and e_2 in languages L_1 and L_2, respectively. We define a mapping relation between the entities e_1 and e_2, if e_1 of the Language L_1 can be translated to e_2 of the Language L_2 without introducing new defects in the translated entity. (@Hao: Should this be something like both entities should have the same behavior).

Mapping relation of API classes: We define a mapping relation between two API classes c_1 and c_2 of languages L_1 and L_2, respectively, if the API class c_1 of L_1 is translated to the API class c_2 of L_2 without introducing new defects in the translated code. Our mapping relation of API classes is many-to-many (@Hao: Please check whether it is many-to-many). For example, the `java.util.ArrayList` class of Java is mapped to either `System.Collections.ArrayList` **or** `System.Collections.Generic.List` of C#. On the other hand, the `java.lang.System` class of Java is mapped to `System.DataTime` **and** `System.Environment` of C# based on how Client code uses these classes. For example, `java.lang.System` is mapped with `System.DataTime` when client code uses the two classes to get the current time and is mapped with `System.Environment` when client code uses the two classes to get environment settings. (@Hao: Could you please revise this sentence. I haven't got what this means?).

[8] ???

[9] http://java.sun.com/j2se/1.5.0/

[10] http://lucene.apache.org/nutch/

Furthermore, mapped API classes can have different functionalities in different languages. For example, `java.lang.String` of Java is mapped to `System.String` of C#. However, `System.String` has an API method `insert`, which does not exist in `java.lang.String`.

Mapping relation of API methods: We define a mapping relation between two API methods m_1 and m_2 of languages L_1 and L_2, respectively, if m_1 is translated to m_2 without introducing defects in the translated code.

Both the mapping relations of API classes and methods are required for achieving language translation. In particular, mapping relation of API classes is required to translate variables such as `file` in Figure **??**. Similarly, mapping relation of API methods is required to translate API methods such as `exist()` in Figure **??**. When an API method is translated from one language to another, the translated method accepts the same parameters (both variables and constants) and implement the same functionality as the original method.

Merged API method: A merged API method of L_1 refers to an API method that is created by merging two other API methods of L_1. (@Hao: From my understanding, i think that merged API method is within the language. Is that correct?) For example, consider two API methods m_1 and m_2 defined in classes C_1 and C_2 of L_1, respectively, with the following signatures:

m_1 `signature:` o_1 C_1.m_1(inp_1^1, inp_2^1, ..., inp_m^1)
m_2 `signature:` o_2 C_2.m_2(inp_1^2, inp_2^2, ..., inp_n^2)

We merge methods m_1 and m_2 to create a new merged API method m_{new} if the output o_1 of m_1 is used either as a receiver object or a parameter for m_2 (i.e., $o_1 == C_2$ or $o_1 == inp_i^2$) in Client code. The signature of the new merged API method m_{new} is shown below:

m_{new} `signature:` o_2 m_{new}(inp_1^1, inp_2^1, ..., inp_m^1, inp_1^2, inp_2^2, ..., inp_n^2)

We next present an example for a merged API method using the illustrative code example shown in Section 2. (@Suresh: Insert a figure for the code sample and refer it from here). Consider the `file` variable, which is a return variable for the constructor and a receiver object for the `exist()` method. As the output of one API method is passed as receiver object of another API method, we can combine these two methods to create a new merged API method m_{new}. Figure 2 shows the m_{new} method `boolean File.exists(string)`. The m_{new} method accepts a `string` parameter that represents a file name and returns a boolean value that describes whether a file exists or not.

A merged API method can be further merged with other API methods or other merged API methods. For simplicity, we use API method to refer to both API method and merged API method in the rest of the paper.

4. APPROACH

Figure 3 shows the overview of our approach. To mine API mapping of two languages, our approach needs a set of projects as data sources. Each project has at least two versions of the two languages under consideration. As mined API mapping describes mapping relations of APIs for the two languages, it is useful to language migration between the two languages. In particular, the first step of our approach is to align client code from versions of two languages so that the aligned source files implement similar functionalities (Section 4.1). The second step is to mine mapping relations of API classes (Section 4.3). The final step is to mine mapping relations of API methods (Section 4.3).

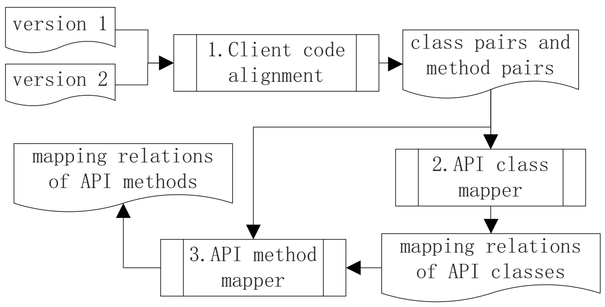

version 1
version 2
1.Client code alignment
class pairs and method pairs
2.API class mapper
mapping relations of API classes
3.API method mapper
mapping relations of API methods

Figure 3: Overview of our approach

4.1 Aligning Client Code

Given two versions of a project, the first step of our approach is to align classes and methods of the two versions. Two aligned classes or methods implement a similar functionality. As they implement a similar functionality, APIs used by them may be replaceable. For two versions of a project, our approach use name similarities of two entities to align classes and methods of the two versions. Here, we treat entity names (class/method names) defined by the two versions of the project and entity names of third-party libraries used by the two versions of the project differently. The former often comes from the same programmer or the same team, or programmers may refer to existing versions for naming entities such as classes, methods, and variables. As a result, for the former, using name similarity is often reliable to distinguish their functionalities. For two entities, our approach chooses Simmetrics[11] to calculate similarities between their names.

Algorithm 1 shows how our approach aligns client code classes. The first step of the algorithm is to find candidate class pairs by names. For two sets of classes (C and C'), the algorithm returns candidate pairs of classes (M) with the maximum similarity and the similarity must be greater than a predefined threshold. Some project may have many classes with the same name, so M may contain more than one pair for a class in a version. To align those classes, the algorithm uses package names of these classes to refine M and returns only one pair with the maximum similarity[12].

In each aligned class pair, our approach further aligns methods within the class pair. The algorithm for methods is similar as the algorithm for classes but relies on other criteria such parameter numbers and names to refine candidate method pairs. These candidates may contain more than one method pair due to overloading.

For the example shown in Section 2, our approach correctly aligns the class named `IndexFiles` and the method named `main` in Java to the class named `IndexFiles` and the method named as `Main` in that their names are quite similar.

4.2 Mapping API classes

The second step of our approach is to mine mapping relations of API classes. As defined in Section 3, mapping relations of API classes are used to translate variables. Consequently, our approach mines mapping relations of API classes based on how aligned client code declares variables (*i.e.*, fields of aligned classes, local variables of aligned methods, and parameters of aligned methods). In particular, for each aligned class pair $\langle c_1, c_2\rangle$, our approach analyzes each field pair $\langle f_1, f_2\rangle$) and considers $\langle f_1.type, f_2.type\rangle$ as one mined mapping relation of API classes when the similarity between $f_1.name$ and $f_2.name$ is greater than a threshold. Similarly, for each aligned method pair $\langle m_1, m_2\rangle$, our approach analyzes each local variable pair $\langle var_1, var_2\rangle$ and considers $\langle var_1.type, var_2.type\rangle$ as one mined mapping relation of API classes when the similarity between $var_1.name$ and $var_2.name$ is greater than a threshold. Also, our approach analyzes each parameter pair $\langle para_1, para_2\rangle$ of m_1 and m_2, and our approach considers $\langle para_1.type, para_2.type\rangle$ as one mined mapping relation of API classes when the similarity between $para_1.name$ and $para_2.name$ is greater than a threshold. Here, the thresholds are the same as Section 4.1 in that they are all for names of client code.

For the example shown in Section 2, our approach mines the mapping relation between `java.io.File` and `System.IO.FileInfo` based on the matched fields of Line 4 and Line 9. The mapping relation of API classes helps translate the variable declared in Line 1 to the variable declared in Line 16.

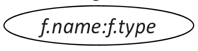
4.3 Mapping API methods

The final step of our approach is to mine mapping relations of API methods. This step has two major sub-steps. First, our approach builds a graph for each client code method. Second, our approach compares the two graphs of each paired client code methods for mapping relations of API methods.

API transformation graph. An API transformation graph (ATG) of a client code method (m) is a directed graph ($G\langle N_{data}, N_m, E\rangle$). N_{data} is a set of the fields of m's declaring class (F), the local variable of m (V), parameters of m (P_1), parameters of methods called by $m(P_2)$, and return values of all methods (R). N_m is a set of methods called by m. E is a set of directed edges. An edge ($d_1 \rightarrow d_2$) from a datum ($d_1 \in N_{data}$) to a datum ($d_2 \in N_{data}$) denotes the data dependency from d_1 to d_2. An edge ($d_1 \rightarrow m_1$) from a datum ($d_1 \in N_{data}$) to a method ($m_1 \in N_m$) denotes d_1 is a parameter or a receiver of m_1. An edge ($m_1 \rightarrow d_1$) from a method ($m_1 \in N_m$) to a datum ($d_1 \in N_{data}$) denotes d_1 is the return value of m_1.

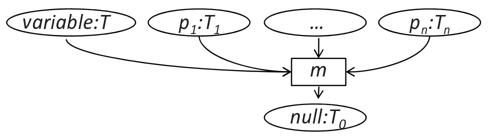
We propose ATG for two considerations. One is for mining mapping relations of method inputs correctly. The inputs of two API methods that satisfy all criteria of mapping relations may have different orders and positions. For example, `java.math.BigDecimal.multiply(BigDecimal)` has a receiver (v_1) and a parameter (p_1); `System.Decimal.Multiply(Decimal, Decimal)` has two parameters (p_2 and p_3). Here, v_1 is mapped with p_2, and p_1 is mapped with p_3. As ATG describes inputs of API methods, our approach is able to deal with the problem of mapping inputs. The other is for mining mapping relations of merging API methods. As ATG describes data dependencies among inputs and outputs, our approach is able to mine mapping relations for merged API methods as shown in Figure 2.

Building API transformation graphs. The first sub-step builds an ATG for each method. The graph contains details such as inputs and outputs for each client code method. In particular, for each method, our approach first builds subgraph for its variables, API methods, and field accesses according to the rules as follows:

(1) $f \in F \cup V \cup P_1 \Rightarrow$ 

Our approach adds used variables of a client code method as these variables are useful to analyze data dependencies among API methods.

(2) $T_0\ m(T_1 p_1, \ldots, T_n p_n) \Rightarrow$



[11]http://sourceforge.net/projects/simmetrics/
[12]For C#, we refer to namespace names for package names.

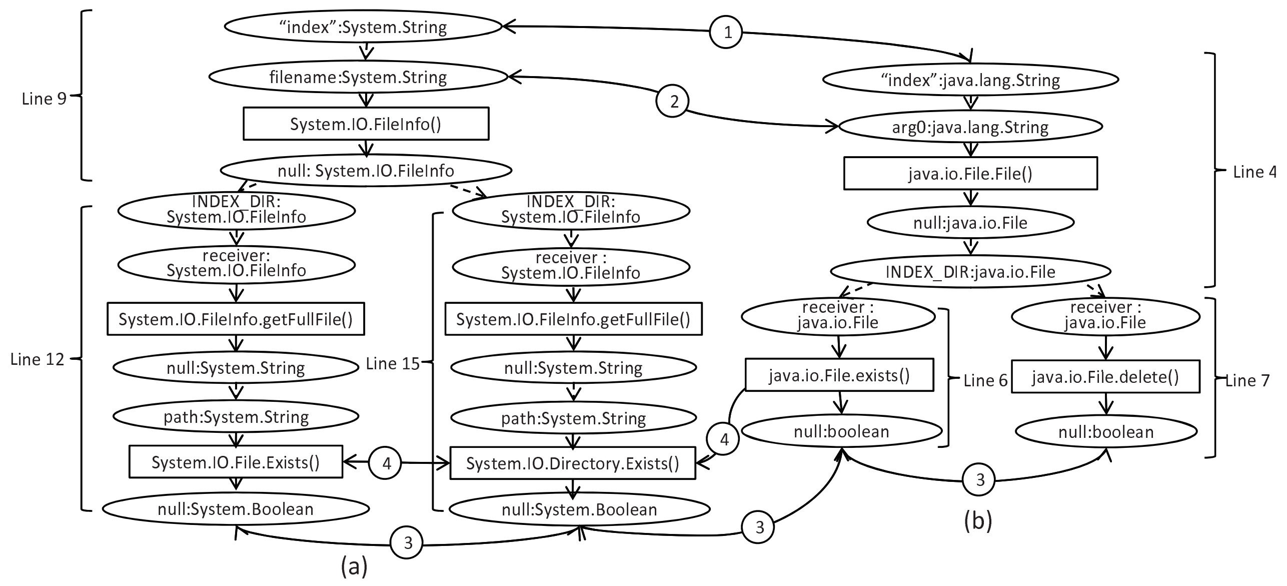


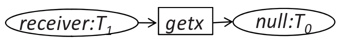
Figure 4: Built ATGs and the main steps of comparing ATGs

Algorithm 1: Align Classes Algorithm

```
Data: $C$ is the classes of a language; $C'$ is the classes of another
      language
Result: $P$ is aligned pairs of classes
begin
    $M \leftarrow findCandidateClassPairs(C, C')$
    while $M.size > 0$ do
        if $M.size > 1$ then
            $M \leftarrow refineByPackageNames(M)$
        if $M.size == 1$ then
            $P.add(M)$
            $C.remove(M[0].c)$
            $C'.remove(M[0].c')$
        $M \leftarrow findCandidateClassPairs(C, C')$
end
```

Our approach adds API methods called by a client code method to the built graph. For each API method m, T is the declaring class of m. Our approach does not add a receiver nodes for static methods, and our approach does not add parameter nodes for methods without parameters.

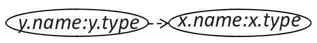
(3) $f.x, f \in F \cup V \Rightarrow$



As Java often uses getters and setters whereas C# often use field accesses, our approach treats field accesses as a special type of method calls. In the preceding sub-graph, f is a variable whose type is T_1, and T_0 is the type of $f.x$.

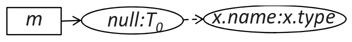
Our approach further connects the built subgraphs through data dependencies among built sub-graphs. In particular, our approach analyzes source files of a client code method statement by statement and adds edges according to the rules as follows:

(4) $x = y, x \in F \cup V \wedge y \in F \cup V \Rightarrow$



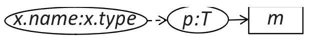
Our approach adds an edge from y to x if y is assigned to x. The edge denotes the data dependency from y to x.

(5) $x = m(), x \in F \cup V \Rightarrow$



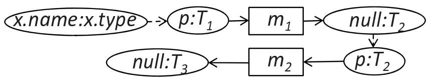
Our approach adds an edge from m to x if m's return value is assigned to x. The edge denotes the data dependency from m's return value to x.

(6) $m(x) \Rightarrow$



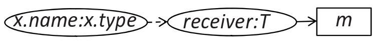
Our approach adds an edge from x to m's parameter node if x is a parameter of m. The edge denotes the data dependency from x to m's parameter.

(7) $m_2(m_1(x)) \Rightarrow$



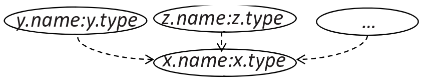
Our approach adds an edge from m_1's return value node to m_2's parameter node if m_1 is used as a parameter of m_2. The edge denotes the data dependency from m_1's return value node to m_1's parameter.

(8) $x.m() \Rightarrow$



Our approach adds an edge from x to m if x is an variable that is related to m. The edge denotes the data dependency from x to m's parameter.

(9) $x = y \ binop \ z \ binop \ \ldots, binop \in \{+, -, *, /\} \Rightarrow$



Our approach adds edges from y, z, and others to x if these variables are connected by bin operations and the return value is assigned to x. The edge denotes the data dependency from y, z, and other variables to x.

For the two source files shown in Section 2, Figure 4 (a) shows the partial ATG of `IndexFiles.cs`, and Figure 4 (b) shows the

Algorithm 2: ATG Comparison Algorithm

```
Data: $G$ is the ATG of a method ($m$); $G'$ is the ATG of $m$'s mapped
      method.
Result: $S$ is a set of mapping relations for API methods
begin
    $P \leftarrow findVarPairs(m, m')$
    for Pair p in P do
        $SM \leftarrow G.nextMethods(p.sharp)$
        $JM \leftarrow G.nextMethods(p.java)$
        $\Delta S = mapping(SM, JM)$
        while $\Delta S \neq \phi | \Delta SM \neq \phi | \Delta JM \neq \phi$ do
            $S.addAll(\Delta S)$
            for Method sm in SM do
                if $sm.isMapped$ then
                    $SM.replace(sm, sm.nextMethod())$
                else
                    $SM.replace(sm, sm.mergeNextMethod())$
            for Method jm in JM do
                if $jm.isMapped$ then
                    $JM.replace(jm, jm.nextMethod())$
                else
                    $JM.replace(jm, Jm.mergeNextMethod())$
            $\Delta S = mapping(SM, JM)$
end
```

partial ATG of `IndexFiles.java`. Figure 4 also lists corresponding line numbers of each sub-graphs. In particular, for Line 4 and Line 9, our approach applies Rule 2 and Rule 6 to build corresponding sub-graphs. For Line 6 and Line 7, our approach applies Rule 2 and Rule 8 to build corresponding sub-graphs. For Line 12 and Line 15, our approach applies Rule 2, Rule 3, and Rule 6 to build corresponding sub-graphs.

Comparing ATGs for mapping relations of API methods. The second sub-step compares the each pair of built ATGs for mapping relations of API methods. As shown in Figure 2, two mapped API methods have (1) the same functionality, (2) the mapping relations of inputs, and (3) the mapping relations of outputs. As two mapped API methods satisfy the preceding three criteria, they are replaceable in client code and thus are useful to aid language migration.

Algorithm 2 shows the main steps of comparing ATGs for mining mapping relations of API methods. For each method pair $\langle m, m' \rangle$, our algorithm first finds matched variable pairs and matched constant pairs of F, V, and P_1 of m and m'. For two variables, our algorithm considers them matched when the two variables have similar names. For constants, our algorithm considers them matched when the two constants have exactly the same value. For each variable pair and each constant pair, our algorithm then finds its connected methods and tries to map these methods. If a method is mapped, our algorithm replaces the method with its next connected method. If a method is not mapped, our algorithm merges the next connected method to this method. As our algorithm merges some methods, both sm and jm may contain more than one methods. In other word, both sm and jm may be two connected sub-graphs (G_{sm} and G_{jm}). Our algorithm considers sm and jm mapped when they satisfy the following criteria:

inputs: (1) the total number of sm's receiver and parameters equal the total number of jm's receiver and parameters; (2) each receiver and each parameter are mapped.

Our algorithm considers two inputs matched if the two inputs connect to two matched variables/constants or the two inputs connect to two outputs of matched API methods.

functionalities: sm and jm have a similar functionality, and our algorithm requires at least one pair of methods with similar names.

Our algorithm uses the maximum similarity of method names between G_{sm} and G_{jm} as the measure for their functionalities.

outputs: sm and jm have mapped outputs.

The outputs of API methods have no names, so our algorithm considers two outputs as matched when the types of the two outputs are matched. Our algorithm continues until S, SM, and JM do not change anymore.

For example, the numbers within circles of Figure 4 show the main steps to mine the mapping relations of API methods as shown in Figure 2. The main steps are as follows:

S1: mapping parameters, fields, local variables, and constants. For the two graphs of each method pair, this step maps variables such as parameters, fields, and local variables by names and maps constants by values. For the preceding example, this step maps two constants as shown by the first red arrow of Figure 4 since the two constants have the same values as `index`.

S2: mapping inputs of API methods. For each variable pair, this step traces to the first connected two API methods and tries to map all the parameters and the receivers of the two API method. For the preceding example, this step maps the parameter named as `filename` to the parameter named as `arg0` as they are of the same type and they connect to the mapped constants.

S3: mapping outputs of API methods. After inputs are mapped, this step further maps outputs of API methods. If it fails to map outputs, this step further merges the next API method and tries to map output of merged API methods. For the preceding example, as the output of `System.IO.FileInfo()` is not mapped to the output of `java.io.File.File()`, this step further merges the next API methods until it finds a match. The arrows marked as "3"" of Figure 4 shows the matched outputs. These outputs are of matched API classes.

S4: mapping functionalities. After inputs and outputs are both mapped, this step further maps functionalities of those merged methods. Give two merged methods with mapped inputs and outputs, this step use the maximum similarity of method names as the measure for their functionalities. In the preceding example, this step maps the two merged methods shown in Figure 4 (a) to the merged methods of the `java.io.File.exist()` as the three merged methods all contain a method named as "exist".

After comparing the graph shown in Figure 4 (a) with the graph shown in Figure 4 (b), our approach mines the mapping relation as shown in Figure 2 by merging variables between API methods.

Project	Source	Java version		C# version	
		#C	*#M*	*#C*	*#M*
neodatis	SourceForge	1298	9040	464	3983
db4o	SourceForge	3047	17449	3051	15430
numerics4j	SourceForge	145	973	87	515
fpml	SourceForge	143	879	144	1103
PDFClown	SourceForge	297	2239	290	1393
OpenFSM	SourceForge	35	179	36	140
binaryNotes	SourceForge	178	1590	197	1047
lucene	Apache	1298	9040	464	3015
logging	Apache	196	1572	308	1474
hibernate	hibernate	3211	25798	856	2538
rasp	SourceForge	320	1819	557	1893
llrp	SourceForge	257	3833	222	978
simmetrics	SourceForge	107	581	63	325
aligner	SourceForge	41	232	18	50
fit	SourceForge	95	461	43	281
Total		11668	75685	6900	34165

Table 1: Subjects

Project	Java version		C# version		Aligned	
	%C	*%M*	*%C*	*%M*	*#C*	*#M*
neodatis	44.7%	54.8%	100.0%	93.6%	408	3728
db4o	87.8%	65.5%	87.6%	74.1%	2674	11433
numerics4j	57.2%	48.6%	95.4%	89.9%	75	174
fpml	93.7%	70.5%	93.5%	56.2%	134	620
PDFClown	86.5%	51.0%	88.6%	82.1%	257	1143
OpenFSM	97.1%	72.1%	94.4%	92.1%	34	129
binaryNotes	98.9%	61.1%	89.3%	92.7%	176	971
lucene	34.9%	26.6%	97.6%	79.8%	453	2406
logging	91.8%	18.1%	58.4%	19.3%	180	285
hibernate	26.4%	1.2%	99.1%	12.6%	848	319
Total	44.9%	28.0%	75%	62.1%	5239	21208

Table 2: Aligned client code

5. EXPERIMENTS

We implemented a tool named MAM based on our approach and conducted two experiments on the tool. Our experiments focus on two research questions as follows:

1. How effective can our approach mine various mapping relations of APIs (Section 5.1)?
2. How much benefit can the mined mapping relations of APIs offer in aiding language migration (Section 5.2)?

We choose fifteen open source projects that have both Java versions and C# versions as the subjects of our experiments, and Table 1 show these subjects. Column "Project" lists names of subjects. Column "Source" lists sources of these subjects. These subjects come from famous open source societies such as SourgeForge[13], Apache[14], and hibernate[15]. Column "Java version" and Column "C# version" list the two versions from each subject. All these versions are the latest versions. For the two columns, sub-column "*#C*" lists numbers of classes. Sub-column "*#M*" list numbers of methods. We notice that Java versions are much larger than C# versions totally. We investigate these projects and find two factors as follows. One is that Java versions of some projects are more update-to-data. For example, the latest Java version of *numericas4j* is 1.3 whereas the latest C# version is 1.2. The other factor is that some projects are migrating from Java into C# in progress. For example, the website[16] of *neodatis* says that *neodatis* is a project in Java and is being ported to .Net. This observation further confirms the usefulness of our approach as our approach aids migrating from one language to other languages. Totally, these projects have 18568 classes and 109850 methods.

All the experiments were conducted on a PC with an Intel Qual CPU @ 2.83GHz and 1.98M memory running the Windows XP operation system.

5.1 Mining API mapping

To evaluate the first research question, we use ten projects from Table 1 as the subjects for mining API mapping.

Aligning client code. We first use our approach to align client code based on name similarities. The threshold is set to 0.6. We choose a relatively low threshold so that our approach does not loose any useful client code.

Table 2 shows the results of this step. For column "Aligned", sub-column "*# C*" lists numbers of aligned classes. Sub-column "*# M*" lists numbers of aligned methods. For each project of Column "C# version" and Column "Java version", sub-column "*%C*" lists the percent of the aligned classes to total classes of corresponding versions. Sub-column "*%M*" lists the percent of the aligned methods to total methods of corresponding versions. We find that the results of Table 2 fall into three categories. This first category includes *db4o*, *fpml*, *PDFClown*, *OpenFSM*, and *binaryNotes*. Our approach achieves relatively high percents for both Java versions and C# versions. In each of the five project, "*%M*" is relatively smaller than "*%C*" for two factors. One is that methods of those unaligned classes cannot be aligned and thus are counted as unaligned. The other is that Java versions usually have many getters and setters and these getters and setters often do not have corresponding methods in C# versions. The second category includes *neodatis*, *numerics4j*, and *lucene*. Our approach aligns C# version well but does not aligns Java versions so well. We find that *neodatis* and *lucene* are migrating from Java to C# in progress and the Java version of *numerics4j* is more update-to-date than its C# version. As a result, some Java classes or methods do not have corresponding implementations in C# versions in these projects and thus are left unmapped. The third category includes *logging* and *hibernate*. Our approach does not align classes and methods of the two projects well. Although both the two projects seem to be migrated from existing Java versions, the programmers of the two projects do not refer to names of existing Java versions for naming entities. For each of the two projects, the percent of aligned classes is relatively high, and the percent of aligned methods is relatively low. We find that even if our approach aligns a wrong class pair, our approach does not aligns methods within the wrong pair as the method names of a wrong pair are quite different. The result suggests that the low threshold may not introduce many negative impacts on the results of API mapping.

For all these projects, our approach does not align all classes and all methods. Besides the intuitive factor of naming entities, one factor lies in that one functionality may be implemented as a single class in one version and is implemented as several classes in the other version. Another factor lies in that a Java version and a C# version may have quite different functionalities. We further discuss this issue in Section 6.

In summary, our approach aligns most classes and methods of eight projects listed in Table 2 using simple name similarities. The result confirms that many programmers refer to existing versions to name entities of a version under development.

Mining API mapping. We then use our approach to mine mapping relations of API classes and API methods.

Table 3 shows the results of this step. For Column "Class" and Column "Method", sub-column "*Num.*" lists numbers of mined mapping relations. The numbers of mined API mapping are largely proportional to the sizes of projects as shown in Table 1 except *logging* and *hibernate*. As classes and methods of the two projects are not quite well aligned, our approach does not mine many mapping relations of APIs from the two projects. For the remaining projects, our approach mines many mapping relations of API classes and API methods. Sub-column "*Acc.*" lists accuracies of mined API mapping. For mined API mapping from each project, we manually inspect top thirty mined mapping relations of APIs, and these accuracies are percents of correct mapping relations. We find that our approach achieves high accuracies except *hibernate*. Although our approach does not align *logging* quite well either, the accuracies of API mapping from *logging* are still relatively high. To mine API mapping of classes, our approach requires names of classes, methods, and variables are all similar. To mine API mapping of methods, our approach requires two built API transformation graphs are similar. The two requirements are strict. As a result, if the first step does not align client code quite well, our approach loses some mapping relations of APIs but does not introduce many false mapping

[13]`http://www.sf.net`
[14]`http://www.apache.org/`
[15]`http://www.hibernate.org/`
[16]`http://wiki.neodatis.org/`

Project	Class		Method		
	Num.	*Acc.*	*Num.*	*Acc.*	*1-1*
neodatis	526	96.7%	3517	100.0%	3506
db4o	3155	83.3%	10787	90.0%	10746
numerics4j	97	83.3%	429	83.3%	427
fpml	199	83.3%	508	83.3%	503
PDFClown	539	96.7%	514	100.0%	504
OpenFSM	64	86.7%	139	73.3%	139
binaryNotes	287	90.0%	671	90.0%	665
lucene	718	90.0%	2725	90.0%	2707
logging	305	73.3%	56	90.0%	53
hibernate	1126	66.7%	7	13.3%	7
Total	7016	85.0%	19353	81.3%	19257

Table 3: Mined API mapping

relations. In other word, our approach is robust to mine accurate API mapping. Sub-column "*1-1*" list the number of one-to-one relations. We find that most mined mapping relations are of this kind. Our approach still needs improvements to mine those n-to-n mapping relations. We further discuss this issue in Section 6.

In summary, our approach mines many mapping relations of APIs totally. These mined mapping relations are accurate and cover various libraries.

Comparing with manually built API mapping. Some translation tools such as CSharp2Java[17] have manually built files that describe mapping relations of APIs. For example, one item from the mapping files of CSharp2Java is as follows:

```
package java.math :: System {
  class java.math.BigDecimal :: System:Decimal {
    method multiply(BigDecimal)
        { pattern =  Decimal.Multiply(@0, @1); }
  }
}
```

This item describes mapping relations between `java.math.BigDecimal.multiply()` and `System.Decimal.Multiply()`. The pattern string describes mapping relations of inputs. In particular, "`@0`" denotes the receiver, and "`@1`" denotes the first parameter. Based on this item, CSharp2Java translates the following code snippet from Java to C# as follows:

```
  BigDecimal m = new BigDecimal(1);
  BigDecimal n = new BigDecimal(2);
  BigDecimal result = m.multiply(n);
  ->
  Decimal m = new Decimal(1);
  Decimal n = new Decimal(2);
  Decimal result = Decimal.Multiply(m,n);
```

To compare with manually built mapping files of CSharp2Java, we translate the mined API mapping with the following strategy.

(1) For each Java class, we translate its mapping relations of classes with the highest supports into mapping files as relations of packages and classes.

(2) For each Java method, we translate its mapping relations of methods with the highest supports into mapping files as relations of methods with pattern strings. For 1-to-1 mapping relations of methods, this step is automatic as mined mapping relations describe mapping relations of corresponding methods and inputs. For n-to-n mapping relations of methods, this step is manual as mined mapping relations do not include adequate details such as how to deal with multiple outputs. For example, the mapping relation of method as shown in Figure 2 has two outputs. We need manually combine the two outputs with an *or* operator.

After the preceding translation,we find that some packages overlap in the mined mapping files and the mapping files of CSharp2Java.

[17] `http://j2cstranslator.wiki.sourceforge.net`

Package	Class			Method		
	P	*R*	*F*	*P*	*R*	*F*
java.io	78.6%	26.8%	52.7%	93.1%	53.2%	73.1%
java.lang	82.6%	27.9%	55.3%	93.8%	25.4%	59.6%
java.math	50.0%	50.0%	50.0%	66.7%	15.4%	41.0%
java.net	100.0%	12.5%	56.3%	100.0%	25.0%	62.5%
java.sql	100.0%	33.3%	66.7%	100.0%	15.4%	57.7%
java.text	50.0%	10.0%	30.0%	50.0%	16.7%	33.3%
java.util	56.0%	25.5%	40.7%	65.8%	12.6%	39.2%
junit	100.0%	50.0%	75.0%	92.3%	88.9%	90.6%
orw.w3c	42.9%	33.3%	38.1%	41.2%	25.0%	33.1%
Total	68.8%	26.4%	47.6%	84.6%	28.7%	56.7%

Table 4: Compared results

We compare mapping relations of APIs within these mapping packages, and Table 4 shows the results. For its sub-columns, sub-column "*P*" denotes precision. Sub-column "*R*" denotes recall. Sub-column "*F*" denotes F-score. *Precision*, *Recall*, and *F-score* are defined as follows:

$$Precison = \frac{true\,positives}{true\,positives + false\,positives} \quad (1)$$

$$Recall = \frac{true\,positives}{true\,positives + false\,negatives} \quad (2)$$

$$F-score = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (3)$$

In the preceding formulae, true positives represent those mapping relations that exist in both the mined API mapping and the golden standard; false positives represent those transitions that exist in the mined API mapping but not in the golden standard; false negatives represent those transitions that exist in the golden standard but not in the mined API mapping. From the results of Table 4, our approach achieves a relatively high precision and a relatively low recall. We further investigate the differences, and we find the impact factors as follows.

(1) The mined mapping files contain correct items that do not exist in the mapping files of CSharp2Java. For example, the mined mapping files contain a mapping relation between `org.w3c.dom.Attr` and `System.Xml.XmlAttribute`, and the mapping relation does not exit in the mapping files of CSharp2Java. As these items are counted as false positives, this impact factor reduces the precisions.

(2) Although we use ten large projects as subjects to mine API mapping, these projects do not cover mapping relations of all API classes and all API methods. Consequently, our approach does not mine mapping relations of the entire API classes and the entire API methods. Although as shown in Table 3 our approach mines many mapping relations, these mapping relations cover many libraries. When we limit mapping relations to the packages as shown in Table 4, the mined mapping relations are actually not so many as expected. On the contrary, the mapping files of CSharp2Java are more detailed as they are manually built. This impact factor reduces the recalls.

(3) Some API classes and API methods between Java and C# have different behaviors. To hide these behaviors from client code, CSharp2Java maps these classes and methods to its implemented classes and methods. For example, CSharp2Java maps `java.util.Set` to `ILOG.J2CsMapping.Collections.ISet`. Our approach did not mine these mapping relations as the subjects in Table 1 do not show such mapping relations. This impact factor reduces both the precisions and the recalls.

In summary, compared with the mapping files of CSharp2Java, our mined mapping files show a relatively high precisions and relatively low recalls. The relatively high precisions show that our mined mapping relations are accurate and contain some mapping relations that are not covered by CSharp2Java. The relatively low recalls show that we need improvements such as introducing more subject projects to cover detailed API mapping.

5.2 Aiding Language Migration

To evaluate the second research question, we feed the mined API mapping to the CSharp2Java tool and investigate whether these mined API mapping can improve the tool. We choose this tool because this tool is a relatively mature project at ILOG[18] (now IBM) and supports the extension of user-defined mapping relations of APIs.

We use CSharp2Java to translate five projects listed in Table 1 from Java to C#, and Table 5 shows the results. For each translated C# project, Column "No MF" lists the number of compilation errors without mapping files. Column "MF" lists the number of compilation errors with the mapping files of CSharp2Java. Column "Ext. MF" lists the number of compilation errors with extended mapping files. We produce these mapping files by combing mined API mapping with the existing mapping files of CSharp2Java. Totally, the mapping files of CSharp2Java helps reduce 36.6% compilation errors, and the extended mapping files helps further reduces 54.4% compilation errors. For the three columns, sub-column "*U*" lists numbers of compilation errors that are caused by wrong using statements. Sub-column "*T*" lists numbers of compilation errors that are caused by unsolved classes. Sub-column "*O*" lists the numbers of compilation errors that are caused by other factors. The three types of compilation errors are not independent. For example, if a mapping relation maps a class in Java to a wrong class in C#, both "*U*" and "*T*" decrease as CSharp2Java can translate a variable in Java to a variable in C# whose type is resolvable. However, as the translated variable may not provide desirable methods, '*O*" may increase when code use some methods of the variable. From the results of Table 5, we find that API mapping helps reduces "*U*" and "*T*" without significant increases of "*O*". In other word, both manually built mapping files and our mined mapping files are useful and accurate. As the five projects use different libraries, the numbers of translated projects are different. In particular, *simmetrics* and *fit* use API classes of J2SE that are covered by mapping files. Consequently, the translate projects of *simmetrics* and *fit* have no "*U*" and "*T*" errors. The *aligner* project also mainly uses J2SE, but it uses many API classes and methods from `java.awt` for its GUI. The mapping files of CSharp2Java do not cover `java.awt`, so the translated project has many "*U*" and "*T*" errors. The mined files map `java.awt` to `System.Windows.Forms` and thus reduce "*U*" errors. However, the mined files also introduce many "*T*" errors as many classes of the two packages are still not mapped. For *rasp* and *llrp*, they both use various libraries besides J2SE. Consequently, the translated projects both have many "*U*" and "*T*" errors. In particular, *llrp* use log4j[19] and jdom[20], and the mined mapping files contain mapping relations of the two libraries. As a result, the mined API mapping helps reduce compilation errors significantly. For *rasp*, it uses some libraries such as Neethi[21] and WSS4J[22]. Even mined API mapping does not cover the two libraries. As a result, the translated project of *rasp* contains many "*U*" and "*T*" errors.

In summary, the mined API mapping improves existing language translation tools such as CSharp2Java. In particular, the mined API mapping helps reduce "*U*" and "*T*" errors in the translated projects.

[18]`http://www.ilog.com/`

[19]`http://logging.apache.org/log4j/`

[20]`http://www.jdom.org/`

[21]`http://ws.apache.org/commons/neethi/`

[22]`http://ws.apache.org/wss4j/`

Projects	No MF			MF			Ext. MF		
	U	***T***	***O***	***U***	***T***	***O***	***U***	***T***	***O***
rasp	405	518	50	173	480	55	103	455	69
llrp	1033	1292	3	463	1076	1	6	262	1
simmetrics	164	32	21	0	12	0	0	6	0
aligner	283	77	8	223	57	9	140	113	9
fit	63	94	17	0	20	7	0	3	7
Total	1948	2016	99	859	1645	72	249	839	86
	4063			2576			1174		

Table 5: Compilation errors

5.3 Threats to Validity

The threat to external validity includes the representativeness of the subjects in true practice. bla bla...

The threat to external validity includes the representativeness of the subjects in true practice. bla bla...

The threat to external validity includes the representativeness of the subjects in true practice. bla bla...

The threat to external validity includes the representativeness of the subjects in true practice. bla bla...

6. DISCUSSION AND FUTURE WORK

In this section, we discuss the related issues of our approach.

Aligning client code of similar functionalities. As shown in Table 2, our approach sometimes fail to align client code. For some considerations, programmers may implement one functionality as one class or one method in one version but implement the same functionality as multiple classes or methods. One feasible way to align these functionalities is to analyze them dynamically. For example, Jiang and Su [5] propose an approach to mine code snippets of similar functionalities. We plan to introduce their approach for those unmatched classes or methods in our future work.

Mining richer API mapping. As shown in Table 4, although we use ten large projects as subjects, our mined API mapping does not achieve high recalls. For a given library, these projects still do not provide adequate source files for mining. Our previous work [11, 12] shows that it is feasible to use the entire open source code on the web as subjects for mining with the help of code search engines such as Google code search[23]. We plan to leverage those search engines to mine richer API mapping in our future work.

Ranking mined mapping relations. One API class or method can be mapped to more than one class or method. For example, both `java.lang.Boolean`↔ `System.Boolean` and `java.lang.Boolean`↔ `System.Byte` are mined as mapping relations of API classes in our experiment. When comparing with CSharp2Java, we choose the formal to generate mapping files as the support of the former is 46 whereas the support of the latter is 4. However, in some cases, the API mapping with the highest support is not necessarily the best choice. For example, `java.util.ArrayList` is mapped to `System.Collections.ArrayList` by support values. The Java class support generic programming, whereas the C# class does not. Consequently, the Java class seems to be better mapped to `System.Collections.Generic.List` as this C# class supports generic programming. We plan to develop some ranking techniques for this issue in future work.

Mining more n-to-n mapping relations of API methods. As shown in Table 3, most mined mapping relations of API methods describe one-to-one relations. Algorithm 2 merges next APIs in

[23]`http://www.google.com/codesearch`

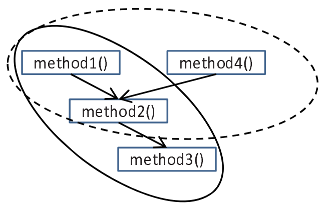


Figure 5: Merging technique

a forward strategy. For the example shown in Figure 5, if the algorithm merges `method1()` and `method2()` but fails to find a match, the algorithm tries to merge `method3()`. In some cases, a match can be found if the algorithm merges `method4()` instead of `method3()`. We plan to refine the algorithm so that it can mine more n-to-n relations in future work.

Migrating n-to-n mapping relations of API methods. A mined n-to-n mapping of API methods may have multiple outputs and complicated internal data processes. Our defined API transformation graphs help find out all essential API methods. However the graph do not describe adequate details to support automatic translations. For example, we need to manually add an *or* operator for the two outputs of the API mapping shown in Figure 2. We pan to add more details so that we can automate migrating of n-to-n mapping relations in our future work.

Migrating unmapped APIs. Our approach mines API mapping of methods that has mapped inputs, mapped outputs, and similar functionalities. Consequently, mined API mapping can be migrated automatically. However, some APIs between two languages cannot satisfy all the three criteria. For these APIs, if outputs are unmapped, our approach can simply ignores outputs when outputs are not used in client code. If inputs or functionalities are unmapped, we plan to develop techniques that analyze how two versions of a project deal with the problem for some reusable code snippets in future work.

7. RELATED WORK

In this section, we introduce related work and discuss our contributions.

Language migration. It is a research topic with a long history to migrate projects of one language into other languages [9]. To reduce the human effort of language migration, researchers propose various approaches to automate the process [3, 7, 13, 14, 16]. Most of these approaches focus the syntax differences among languages. For example, Deursen *et al.* [13] propose an approach to identify objects in legacy code, and the results are useful to deal with the difference between object-oriented languages and procedural languages. As shown by El-Ramly *et al.* [2]'s experience report, existing approaches and tools support only a subset of APIs, and consequently it becomes an important to automate API transformation. Our approach mines API mapping among languages to aid language migration, complementing the preceding approaches.

Library migration. With the evaluation of libraries, some APIs may become incompatible. To deal with the problem, some approaches have been proposed. In particular, Henkel and Diwan [4] propose an approach that captures and replay API refracturing actions to keep client code updated. Xing and Stroulia [15] propose an approach that recognizes the changes of APIs by comparing the differences of two versions of libraries. Balaban *et al.* [1] propose an approach to help translate client code when mapping relations of libraries are available. Different from these approaches, our approach focuses on mapping relations of APIs among different languages. In addition, as our approach uses ATGs to mine mapping relations of APIs, our approach helps mine mapping relations for those API methods whose input orders is changed or whose functionalities are split into several methods if our approach is applied in library migration.

8. CONCLUSION

Although researchers have proposed many approaches to help language migration. These approach achieves only limited success as projects often use many APIs and mapping relations of APIs are difficult to obtain. In this paper, we propose an approach to mine mapping relations of APIs from existing different versions of a project automatically. We conducted experiments on our approach. The results show that our approach mines various API mapping between Java and C# and API mapping improves existing language translators such as CSharpToJava.

9. REFERENCES

[1] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proc. 20th OOPSLA*, pages 265–279, 2005.

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

[3] J. Hainaut, A. Cleve, J. Henrard, and J. Hick. Migration of legacy information systems. *Software Evolution*, pages 105–138, 2008.

[4] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proc. 27th ICSE*, pages 274–283, 2005.

[5] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proc. the 8th ISSTA*, pages 81–92, 2009.

[6] T. Jones. *Estimating software costs*. McGraw-Hill, Inc. Hightstown, NJ, USA, 1998.

[7] M. Mossienko. Automated COBOL to Java recycling. In *Proc. 7th CSMR*, pages 40–50, 2003.

[8] D. Orenstein. QuickStudy: Application Programming Interface (API). *Computerworld*, 10, 2000.

[9] H. Samet. Experience with software conversion. *Software: Practice and Experience*, 11(10), 1981.

[10] A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, pages 111–124, 2000.

[11] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. 22nd ASE*, pages 204–213, November 2007.

[12] S. Thummalapenta and T. Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proc. 23rd ASE*, 2008.

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

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

[15] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.

[16] K. Yasumatsu and N. Doi. SPiCE: a system for translating Smalltalk programs into a C environment. *IEEE Transactions on Software Engineering*, 21(11):902–912, 1995.