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

Since the inception of programming languages, researchers and practitioners developed various languages such as Java and C#. To address business requirements and to survive in competing markets, companies often have to develop different versions of their projects in different languages. Migrating projects from one language to another language (such as from Java to C#) manually is a tedious and error-prone task. In contrast, automatic translation of projects from one language to another requires the knowledge of how Application Programming Interfaces (API) of one language are mapped with the APIs of the other language. In this paper, we propose a novel approach that mines API mapping relations from API client code. Our approach accepts a set of projects with versions in two languages and mines API mapping relations based on how APIs are used by the two versions. These mined API mapping relations assist in translation of projects from one language to another. Based on our approach, we implemented a tool, called MAM (**M**ining **A**PI **M**apping), and conducted two evaluations to show the effectiveness of our approach. The results show that our approach mines various mapping relations of APIs between Java and C# with 83.2% accuracy. The results also show that mined API mapping relations reduce 54.4% compilation errors in translated projects of the Java2CSharp tool as our mappings include relations that do not exist in language migration tools such as Java2CSharp.

1. INTRODUCTION

A programming language serves as a means for instructing computers to achieve a programming task at hand. Since their inception, various programming languages came into existence due to reasons such as existence of many platforms and requirements for different programming styles. To address business requirements and to survive in competing markets, companies often have to develop different versions of their projects in different languages. For example, many well-known projects such as Lucene[1] and WordNet[2] provide multiple versions in different languages. As described by Jones [6], about one-third of the existing projects have multiple versions in different languages.

Migrating projects from one language to another language (*e.g.*, from Java to C#) manually is a tedious and error-prone task. Some companies have incurred huge losses during the process of language migration. For example, Terekhov and Verhoef [10] stated that at least three companies went bankrupt and another company lost 50 million dollars due to failed language migration projects. A natural way to address this issue is to develop a translation tool that can automatically translate projects from one language to another. However, it is challenging to develop such a translation tool as the translation tool should have knowledge of how one programming language is mapped to the other language. In literature, exist approaches [3, 7, 16] address the problem of language migration partially, because these approaches expect programmers to describe how one language is mapped to another language. Based on the mappings provided as input, these existing approaches translate projects from one language to another. As programming languages provide a large number of APIs, writing rules manually for all APIs is tedious and error-prone. As a result, 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 usage in practice.

In this paper, we propose a novel approach that mines how APIs of one language are mapped to the APIs of another language. We refer this mapping as *mapping relations of APIs*. In our approach, we use to mine mapping relations based on API usages in the client code rather based on API implementations for three major reasons: (1) API libraries often do not provide source files especially for those C# libraries. (2) Mining relations based on API implementations often can have relatively low confidence than mining mapping relations based on API usages. The reason is that API implementations have only one call site for the analysis, whereas API usages can have many call sites for mining. (3) Mapping relations of APIs are often complex and cannot be mined solely based on the information available in the API implementations. First, mapping parameters of an API method in one language with an API method in the other language can be complex. For example, consider the following two API methods in Java and C#:

m_1 in Java: BigDecimal java.math.BigDecimal.multiply (BigDecimal p_1^1)
m_2 in C#: Decimal System.Decimal.Multiply (Decimal p_1^2, Decimal p_2^2)

Here, m_1 has a receiver, say v_1^1, of type `BigDecimal` and has one parameter p_1^1, and m_2 has two parameters p_1^2 and p_2^2. Based on the definitions of these inputs, v_1^1 is mapped to p_1^2, and p_1^1 is mapped to p_2^2. Second, an API method of one language is mapped to more than one API method in the other language. For example, consider the following two API methods:

m_3 in Java: E java.util.LinkedList.removeLast()

[1] `http://lucene.apache.org/`

[2] `http://wordnet.princeton.edu/`

ICSE '2010 Cape Town, South Africa

m_4 in C#: void System.Collections.Generic.LinkedList.RemoveLast()

Although the method names of m_3 and m_4 are the same, m_3 in Java cannot be directly mapped with m_4 in C#. The reason is that m_3 in Java returns the last element removed from the list, whereas m_4 does not return any element. Therefore, m_3 is mapped to two API methods m_4 and m_5 (shown below) in C#. The API method m_5 returns the last element and should be called before calling m_4.

m_5 in C#: void System.Collections.Generic.LinkedList.Last()

To deal with the complexity of mining API mapping, we construct a graph, referred as *API transformation graph* (ATG), for aligned methods of the client code in both languages. These ATGs precisely capture inputs and outputs of API methods, and help mine relationships API methods. This paper makes the following major contributions:

- A first approach that mines API mapping between different languages using API client code. Our approach addresses an important and yet challenging problem that is not addressed by previous work on language migration.
- A technique to build and ATGs. As ATGs describe data dependencies among inputs and outputs of API methods, our approach is able to mine complex mapping relations between API methods.
- A tool named MAM based on our approach and two evaluations on 15 projects with both Java and C# versions. These projects include 18,568 classes and 109,850 methods. The results show that our approach mines 26,369 mapping relations of APIs with 83.2% accuracies and the mined API mapping relations reduce 55.4% of compilation errors and 43.0% bugs during translation projects from Java to C#.

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

2. DEFINITIONS

We next present definitions of 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 uses J2SE[3] as an API library, whereas Nutch[4] 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 code.

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. For example, the `java.util.ArrayList` class of Java is mapped to either `System.Collections.ArrayList` **or** `System.Collections.Generic.List` of C#, whereas 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. In particular, when client code uses APIs to get the current time, `java.lang.System` is mapped with `System.DataTime`. At the same time, when client code uses APIs to get environment settings, `java.lang.System` is mapped with `System.Environment`.

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

Figure 1: A translated code snippet from Java to C#.

Furthermore, mapped API classes may have different behaviors. 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 3. Similarly, mapping relation of API methods is required to translate API methods such as `exist()` in Figure 3. 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. 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 $C_1.m_1(inp^1_1,\ inp^1_2,\ \ldots,\ inp^1_m)$
m_2 `signature:` o_2 $C_2.m_2(inp^2_1,\ inp^2_2,\ \ldots,\ inp^2_n)$

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^2_i) in Client code. The signature of the new merged API method m_{new} is shown below:

m_{new} `signature:` o_2 $m_{new}(inp^1_1,\ inp^1_2,\ \ldots,\ inp^1_m, inp^2_1,\ inp^2_2,\ \ldots,\ inp^2_n)$

We next present an example for a merged API method using the illustrative code example shown in Section 3. For the code snippet shown in Figure 3, 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 5 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.

3. EXAMPLE

Our approach accepts existing projects such as Lucene that have

[3] `http://java.sun.com/j2se/1.5.0/`
[4] `http://lucene.apache.org/nutch/`

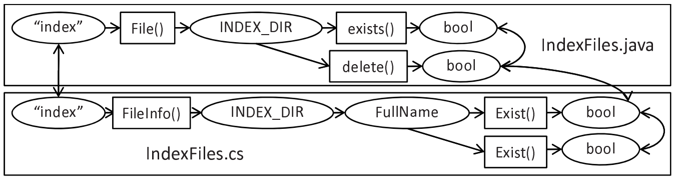


Figure 2: API methods connected by inputs and outputs

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

Figure 3: A code example for language migration.

```
                  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 4: Two versions (Java and C#) of client code

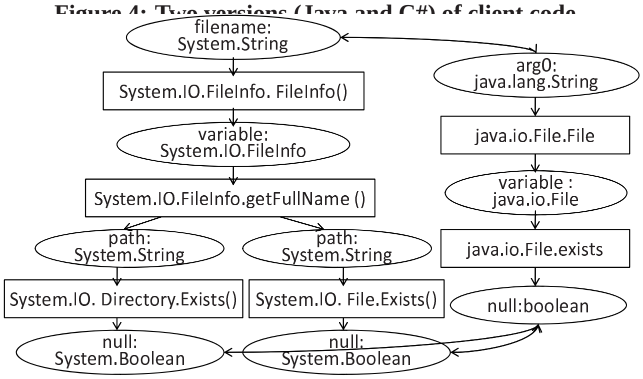


Figure 5: API mapping

both Java and C# versions, and mines mapping relations of APIs. We refer these existing projects as client code using the APIs of two languages. First, our approach aligns classes and methods of the two versions by using a matching algorithm based on similarities in the names of classes and methods. Aligning client code based on names of classes and methods is based on our observation of how existing projects such as rasp[5] are migrated from one language to another. We observed that while migrating rasp project from Java to C#, programmers first rename source files from Java to C# and systematically address the compilation errors by replacing Java APIs with C# APIs. During this procedure, names of classes, methods, fields of classes, or local variables in methods often remain the same between the two versions. Therefore, we use name similarities for aligning client code of the two versions. Second, our approach maps API classes of one language with the other language by matching the names of fields of classes and local variables of methods in the client code. Finally, our approach maps API methods of one language with the other language. Mapping API methods is challenging as one API method of one language can be mapped to multiple API methods of the other language (as shown in our preceding example).

We next use an example to illustrate challenges involved in mining API mapping. Consider that a programmer needs to migrate a Java code example (shown in Figure 3) to C# using a translation tool. This code example accepts a `string` input that represents name of a file or directory and returns a `boolean` value that describes whether the file or directory exists. To achieve this functionality, the code example declares a local variable, called `file`, of type `java.io.File` and calls the `exists` method. We consider the local variable `file` as a receiver object for the `exists` method.

To translate this code example into C#, the translation tool needs to know mapping relations of API classes. For example, the translation tool needs to know the mapped API class for `java.io.File` in C# to translate the variable `file` to C#. In addition, the translation tool needs to know the mapped API methods of `exists`. Furthermore, the translated code should be able to accept the same input "`test`" and produce the same output as the Java code example.

To mine mapping relations of APIs, our approach uses existing projects such as Lucene that have both Java and C# versions. First, our approach aligns classes and methods of the two versions by using a matching algorithm based on similarities in the names of classes and methods between the two versions. For example, our approach aligns `IndexFiles.java` with the `IndexFiles.cs` (shown in Figure 4) as the names of their classes and methods are similar.

Next, our approach mines mapping relations of API classes by comparing entities such as names of fields in aligned classes, or variable names or constants in aligned methods. Figure 2 shows how our approach maps variables and constants in aligned methods of the client code. Our approach uses a text-based similarity measure for comparing these entities and considers the entities as similar if the measure is greater than a given threshold. These relations help translate variables from one language to another. For example, our approach identifies the constant value "`index`", in Lines 4 (Java) and 9 (C#) (Figure 4) and maps the API classes associated with these constants. For example, using the constant value "`index`", our approach maps the API class `java.io.File` of Java to `System.IO.FileInfo` of C#.

After mapping API classes between the two languages, our approach maps API methods. Mapping API methods is challenging as often an API method of one language can be mapped to multiple API methods of the other language. Furthermore, mapping relations of API methods should also describe how parameters and return values are of an API method in one language are mapped to an API in the other language. To address these challenges, our approach constructs a graph, referred as *API transformation graph* (ATG), for aligned methods of the client code in both languages. These ATGs precisely capture inputs and outputs of API methods, and help mine relationships of API methods. Figure 5 shows a mapping relation between API method `Exist` from one language to another. Section 4.3 presents more details on how we mine the mapping relations of API methods using ATGs.

4. APPROACH

Our approach accepts a set of projects as data sources and mines API mapping between two different languages L_1 and L_2. As mined API mapping describes mapping relations of APIs between the two languages, this mapping is useful for language migration

[5] `http://sourceforge.net/projects/r-asp/`

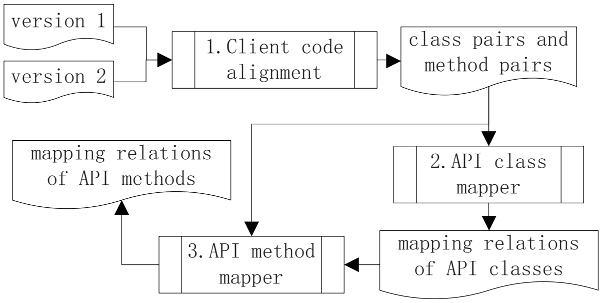

version 1
version 2
1.Client code alignment
class pairs and method pairs
2.API class mapper
mapping relations of API methods
3.API method mapper
mapping relations of API classes

Figure 6: Overview of our approach

between the two languages. For each project used as a data source, our approach requires atleast two versions of the project (one version in L_1 and the other version in L_2). Figure 6 shows the overview of our approach.

First, our approach aligns client code in languages L_1 and L_2 so that the aligned source files implement similar functionalities (Section 4.1). Second, our approach mines mapping relations of API classes (Section 4.3). Finally, our approach mines mapping relations of API methods (Section 4.3) defined by the mapped API classes.

4.1 Aligning Client Code

Initially, our approach accepts two versions of a project (one version in L_1 and the other version in L_2) and aligns classes and methods of the two versions. Aligned classes or methods between the two versions implement a similar functionality. As they implement a similar functionality, APIs used by these classes or methods can be replaceable.

To align classes and methods of the two versions, our approach uses name similarities between entities (such as class names or method names) defined by the two versions of the project. In our approach, we have two different kinds of entity names: entity names defined by the two versions of the project and entity names of third-party libraries used by the two versions of the project. The first kind 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. Therefore, name similarity is often reliable to distinguish functionalities of the first kind compared to the second kind. Our approach uses Simmetrics[6] to calculate name similarities.

Algorithm 1 shows how our approach aligns client code classes. The first step is to find candidate class pairs by names. For two sets of classes (C and C'), the algorithm returns candidate class pairs (M) with a similarity greater than a given threshold, referred to as *SIM_THRESHOLD*. As some projects may have many classes with the same name, M may contain more than one matching pair for a class in a version. To align those classes, our algorithm uses package names of these classes to refine M and returns only one matching pair with the maximum similarity[7].

In each aligned class pair, our approach further aligns methods within the class pair. The algorithm for methods is similar to the algorithm for classes but relies on other criteria such as the number of parameters and names of parameters to refine candidate method pairs. These candidates may contain more than one method pair due to overloading. For the example shown in Section 3, our approach correctly aligns the class `IndexFiles` and the method `main` in Java to the class `IndexFiles` and the method `Main` in C# as their names are quite similar.

Algorithm 1: Align Classes Algorithm

```
Input: $C$ is the classes of a language; $C'$ is the classes of another
       language
Output: $P$ 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
```

4.2 Mapping API classes

In this step, our approach mines mapping relations of API classes. As defined in Section 2, 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 such as fields of aligned classes, parameters of aligned methods and local variables 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 *SIM_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 *SIM_THRESHOLD*.

For the example shown in Section 3, our approach mines the mapping relation between `java.io.File` and `System.IO.FileInfo` based on the matched fields of Lines 4 and 9 (Figure 4). The mapping relation of API classes helps translate the variable declared in Line 1 (Figure 3) to the variable declared in Line 16 (Figure 1).

4.3 Mapping API methods

In this step, our approach mines mapping relations of API methods. This step has two major sub-steps. First, our approach builds a graph, referred as API transformation graph, for each client code method. Second, our approach compares the two graphs of each paired client code methods for mining mapping relations of API methods.

4.3.1 API transformation graph

We propose API transformation graphs (ATG) to help deal with the two challenges of mining API mapping listed in Section 1. An API transformation graph 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 F of m's declaring class, local variables V of m, parameters P_1 of m, parameters P_2 of methods called by m, and return values R of all methods. 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 related variable 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.

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

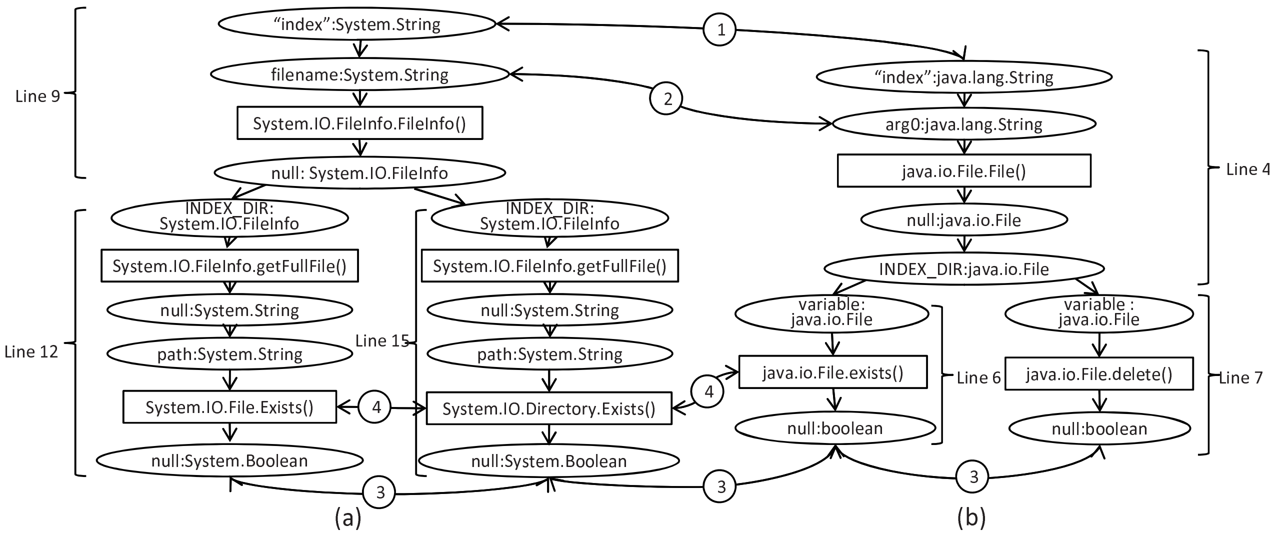
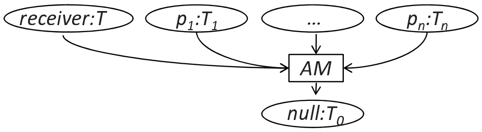


Figure 7: Built ATGs and the main steps of comparing ATGs

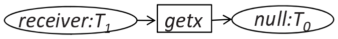
4.3.2 Building API transformation graphs

Our approach builds an ATG for each method m. ATG includes information such as inputs and outputs for each client code method. In particular, for each method m, our approach first builds subgraph for its variables, API methods, and field accesses according to the following rules:

1. $\forall\, f \in F \cup V \cup P_1$, our approach adds a node to the built ATG. The reason for considering these variables such as fields in declaring class or local variables in method m used in client code is that these variables are useful to analyze data dependencies among API methods.
2. $\forall$ API methods of the form "$T_0\ T.AM(T_1 p_1, \ldots, T_n p_n)$" called by method m, our approach adds receiver (of type T) and parameter nodes to the built ATG as shown below. Our approach does not add receiver node for static API methods.

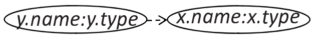


3. $\forall\, f \in F \cup V$, if f is a non-primitive variable of type T_1 and a field x of T_1 is accessed as $f.x$, our approach adds nodes to the built ATG as shown below. As Java often uses getters and setters whereas C# often use field accesses, our approach treats field accesses as a special type of method calls.

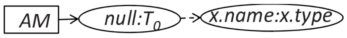


Our approach adds additional edges to the built ATG (and subgraphs inside ATG) representing data dependencies among built sub-graphs. We use the following rules for adding additional edges to the built ATG.

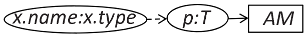
1. $\forall$ statements of the form $x = y$, where $x \in F \cup V \wedge y \in F \cup V$, our approach adds an edge from y to x. This edge represents that x is data dependent on y.



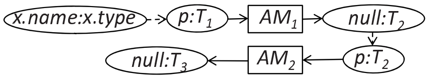
2. $\forall$ statements of the form $x = AM()$, where $x \in F \cup V$, our approach adds an edge from AM to x if the return value of AM is assigned to x. This edge represents that x is data dependent on the return value of AM.



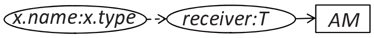
3. $\forall$ API methods $AM(x)$ called by method m, our approach adds an edge from x to the parameter node of AM. This edge represents that the parameter of AM is data dependent on x.



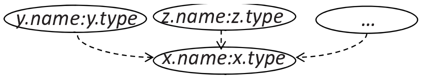
4. $\forall$ statements of the form $m_2(m_1(x))$, our approach adds an edge from the return value node of m_1 to the parameter node of m_2 parameter node. This edge represents that the parameter of m_2 is data dependent on the return value of m_1.



5. $\forall$ statements of the form $x.m()$, our approach adds an edge from x to m as x is the receiver object of m. This edge represents that the receiver object of m is data dependent on x.



6. $\forall$ statements of the form $x = y\ op\ z\ op\ \ldots, op \in \{+, -, *, /\}$, our approach adds edges from y, z, and others to x, as these variables are connected by binary operations and the return value is assigned to x. The edge denotes the data dependency from y, z, and other variables to x. For simplicity, our approach ignores *op* info. We discuss the issue in Section 6.



For each method m in the client code, our approach applies preceding rules for each statement from the beginning to the end of m. Within each statement, our approach applies these rules based on their nesting depth in the abstract syntax tree. For example, for the statements of the form $m_2(m_1(x))$, our approach first applies these rules on m_1 and then on m_2.

Figures 7a and 7b show partial ATGs for C# (`IndexFiles.cs`) and Java (`IndexFiles.java`) code examples shown in Figure 4, respectively. Figure 7 also shows corresponding line numbers of each sub-graph. Our approach applies Rules 2 and 6 for Lines 4 and 9 (Figure 4) to build corresponding sub-graphs in the ATG. For Lines 6 and 7 (Figure 4), our approach applies Rules 2 and 8 to build corresponding sub-graphs in the ATG. For Lines 12 and 15 (Figure 4), our approach applies Rule 2, 3, and 6 to build corresponding sub-graphs.

Algorithm 2: ATG Comparison Algorithm

```
Input: $G$ is the ATG of a method $(m)$; $G'$ is the ATG of $m$'s mapped
       method.
Output: $S$ 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
```

4.3.3 Comparing API transformation graphs

The second sub-step compares each pair of built ATGs for mining mapping relations of API methods. Our mapped API methods satisfy three criteria: (1) Mapped API methods implement the same functionality. (2) Mapping relation describes the relation between parameters of mapped API methods. (3) Mapping relation describes the relation between return values of mapped API methods. The two mapped API methods in two different languages satisfying the preceding three criteria are replaceable in the client code. Therefore, these mapped API methods assist for migrating client code from one language to another.

Algorithm 2 presents major steps of comparing ATGs for mining mapping relations of API methods. Consider two methods m and m' of two different languages L and L', respectively, in the client code. Consider that the associated ATGs of m and m' are compared to mine mapping relations of API methods. First, our algorithm finds matching variables $\in F$, V, and P_1 in m and m'. Our algorithm maps two variables v and v' of methods m and m', respectively, if the similarity measure on their names is greater than *SIM_THRESHOLD*. For constants in m and m', our algorithm maps those two constants, if they have exactly the same value. Our algorithm uses these variable and constant mappings to compute mappings between API methods that use these variables and constants. Our algorithm uses the following criteria for mapping two API methods jm and sm.

Matching entities: The first criterion is based on entities such as receiver variable or parameters of jm and sm to map jm and sm. We map jm with sm, if the receiver variable of API method jm is mapped to the receiver variable of sm, and there is a one-to-one mapping between parameters of jm and sm.

Matching functionalities: The second criterion is based on functionalities of jm and sm. We consider that jm and sm implement the same functionality, if the similarity measure between the name of jm and the name of sm is greater than *SIM_THRESHOLD*.

Matching outputs: The third criterion is based on the return values of jm and sm. Consider the return values of jm and sm as r_1 and r_2, respectively. We map jm with sm, if the type of r_1 is mapped with the type of r_2 in mapping API classes relationship.

Our algorithm first attempts to map first API method jm in m with the first API method sm in m'. If our algorithm successfully maps jm with sm, our algorithm moves to the next available API methods in m and m' of the client code. If our algorithm does not able to map jm with sm, our algorithm merges sm and jm with their next available API methods in the corresponding ATGs, respectively, and attempts to map merged API methods. For two merged API methods, our algorithm uses the maximum similarity of method names between jm and sm as a similarity measure for matching their functionalities. With each iteration, sm or jm or the mapping relation (represented as S) in the algorithm changes. Therefore, we repeat our algorithm till S, sm, and jm do not change anymore.

We next explain our algorithm using the illustrative example shown in Figure 7. The numbers shown in circles represent the major steps in our algorithm for mining mapping relations of API methods. We next explain each step in detail.

S1: mapping parameters, fields, local variables, and constants. Given two ATGs of each method pair $\langle m, m' \rangle$, this step maps variables such as parameters, fields, and local variables by comparing their names and maps constants by comparing their values. As shown in Figure 7, Step 1 maps two constants as both the constants have the same value `index`.

S2: mapping inputs of API methods. Step 2 mines mapping relations of API methods using variable and constant mapping relations. Initially, this step identifies first API methods in the two ATGs and tries to map their parameters and receiver objects of the two API methods. In our current example, this step maps the parameter `filename` to the parameter `arg0` as these parameters are of the same type and their associated constants are mapped.

S3: mapping outputs of API methods. In contrast to Step 2 that maps parameters, Step 3 maps return values of API methods. In this step, if our approach is not able to map return values, our approach merges the next API method and then attempts to map return values of merged API methods. In our current example shown in Figure 7, return value of `System.IO.FileInfo.FileInfo()` cannot be mapped to the return value of `java.io.File.File()`. Therefore, our approach merges next API methods in the ATG till the `Exists` API method, as the return values (shown as `Boolean`) match only after the `Exists` API method. Figure 7 shows Step 3 along with the matching return values.

S4: mapping functionalities. After our approach maps parameters and return values, this step further maps functionalities of those merged API methods. Given two merged API methods with mapped parameters and return values, this step uses the similarity measure based of their method names as a criterion for matching their functionalities. In the preceding example, this step maps the two merged API methods shown in Figure 7a to the merged API methods of the `java.io.File.exist()` as all three merged API methods include the method named `exist`.

Our approach applies preceding steps on ATGs (as shown in Figures 7a and 7b) and mines mapping relations. An example mapping relation from the preceding ATGs is shown in Figure 5.

5. EVALUATIONS

We implemented a tool named MAM based on our approach and conducted two evaluations on the tool. Our evaluations 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)?

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

We choose 15 open source projects that have both Java versions and C# versions as the subjects of our evaluations, 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[8], Apache[9], and hibernate[10]. Columns "Java version" and "C# version" list the two versions from each subject. All these used versions are the latest versions at the time of writing. For these two columns, sub-column "*#C*" lists numbers of classes, and sub-column "*#M*" lists 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-date. 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 to C# in progress. For example, the website[11] of *neodatis* states that *neodatis* is a project in Java and is being ported to C#. This observation further confirms the usefulness of our approach as our approach aids migrating from one language to other languages. Totally, these projects have 18,568 classes and 109,850 methods.

We conducted all the evaluations on a PC with an Intel Qual CPU @ 2.83GHz and 1.98M memory running Windows XP.

5.1 Mining API mapping

To evaluate the first research question, we use 10 projects from Table 1 as the subjects for mining API mapping.

Aligning client code. We first use our approach to align client code. The threshold is set to 0.6 based on our initial experience. We choose a relatively low threshold so that our approach can take into account as much client code as possible.

Table 2 shows the results of this step. For column "Aligned", sub-column "*# C*" lists numbers of aligned classes, and sub-column "*# M*" lists numbers of aligned methods. For each project of Column "C# version" and Column "Java version", sub-column "*%C*" lists the percentage of the aligned classes among total classes of corresponding versions. Sub-column "*%M*" lists the percentage of the aligned methods among total methods of corresponding versions. Row "Total" of the two sub-columns lists the percentage of aligned methods/classes among the total methods/classes as shown in Table 1. We find that the results of Table 2 fall into three categories. This first category includes *db4o*, *fpml*, *PDFClown*, *OpenFSM*, and *binaryNotes*. There, our approach achieves relatively high percentages for both Java versions and C# versions. For each of the five project, "*%M*" is relatively smaller than "*%C*" since methods of those unaligned classes cannot be aligned and thus are counted as unaligned[12]. The second category includes *neodatis*, *numerics4j*, and *lucene*. There, our approach aligns C# versions well but does not align 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*. There, our approach does not align classes and methods of the two projects well. Although both of the two projects seem to be migrated from existing Java versions, the programmers of the two projects often do not refer to names of existing Java versions for naming entities. For each of the two projects, the percentage of aligned classes is relatively high, and the percentage of aligned methods is relatively low. We find that even if our approach aligns a wrong class pair, our approach does not align methods within the wrong pair as the method names of a wrong pair are quite different. The result suggests that we can take method names into account when aligning classes in future work.

Project	Java version		C# version		Aligned	
	%C	*%M*	*%C*	*%M*	*#C*	*#M*
db4o	87.8%	65.5%	87.6%	74.1%	2674	11433
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
neodatis	44.7%	54.8%	100.0%	93.6%	408	3728
numerics4j	57.2%	48.6%	95.4%	89.9%	75	174
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
Average	53.2%	30.8%	88.8%	69.2%	524	2121

Table 2: Results of Aligning client code

For all these projects, our approach does not align all classes and all methods. We find another two factors besides the factor of different entities naming across languages. First, one functionality may be implemented as a single class in one language version and is implemented as multiple classes in the other language version. Second, a Java version and a C# version sometimes may have quite different functionalities. We discuss these issues in Section 6.

In summary, as shown by Row "Average", our approach aligns most classes and methods on average. The result confirms that many programmers refer to existing versions of another language 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 Columns "Class" and "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 these 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 the top 30 mined

[8] `http://www.sf.net`

[9] `http://www.apache.org/`

[10] `http://www.hibernate.org/`

[11] `http://wiki.neodatis.org/`

[12] Another factor lies in that Java versions usually have many getters and setters and these getters and setters often do not have corresponding methods in C# versions.

Project	Class			Method		
	Num.	*Acc.*	*J2SE*	*Num.*	*Acc.*	*J2SE*
db4o	3155	83.3%	246	10787	90.0%	513
fpml	199	83.3%	53	508	83.3%	229
PDFClown	539	96.7%	102	514	100.0%	126
OpenFSM	64	86.7%	27	139	73.3%	12
binaryNotes	287	90.0%	56	671	90.0%	59
neodatis	526	96.7%	86	3517	100.0%	600
numerics4j	97	83.3%	8	429	83.3%	34
lucene	718	90.0%	147	2725	90.0%	584
logging	305	73.3%	93	56	90.0%	21
hibernate	1126	66.7%	164	7	13.3%	5
Total	6695	86.7%	344	19110	90.0%	1768

Table 3: Results of mining API mapping

API mapping (*i.e.*, percentages of correct mapping relations). For mined API mapping from each project, we manually inspect top 30 mined mapping relations of APIs and classify them as correct or incorrect based on programming experiences. 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 that names of classes, methods, and variables are all similar. To mine API mapping of methods, our approach requires that two built API transformation graphs are similar. The two requirements are relatively strict. As a result, if the first step does not align client code quite well, our approach misses some mapping relations of APIs but does not introduce many false mapping relations. In other words, our approach is robust to mine accurate API mapping. Sub-column "*J2SE*" lists mined API mapping between J2SE and NET. We next compare these mapping relations with manually built mapping relations.

Row "Total" lists the total result after we merge all duplicated mapping relations. In summary, our approach mines a large number of 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 Java2CSharp[13] have manually built files that describe mapping relations of APIs. For example, one item from the mapping files of Java2CSharp 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, Java2CSharp 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 Java2CSharp, we translate our mined API mapping with the following strategy. First, for each Java class, we translate its mapping relations of classes with the highest supports into mapping files as relations of packages and classes. Second, 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 many-to-many 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. We further discuss this issue in Section 6.

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%
Average	68.8%	26.4%	47.6%	84.6%	28.7%	56.7%

Table 4: Results of comparing results

The mapping files of Java2CSharp cover 13 packages defined by J2SE and 2 packages defined by JUnit[14], and we treat these mapping files as a golden standard. We find 9 packages overlapping between the mined mapping files and the mapping files of Java2CSharp. We compare mapping relations of APIs within these mapping packages, and Table 4 shows the results. Column "Class" lists the results of comparing API classes. Column "Method" lists the results of comparing API methods. For their 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:

$$Precison = \frac{true\,positives}{true\,positives + false\,positives} \tag{1}$$

$$Recall = \frac{true\,positives}{true\,positives + false\,negatives} \tag{2}$$

$$F\!-\!score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3}$$

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 three main causing impact factors. First, the mined mapping files contain correct items that do not exist in the mapping files of Java2CSharp. 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 exist in the mapping files of Java2CSharp. As these items are counted as false positives, this impact factor reduces the precisions. Second, although we use 10 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

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

[14] `http://www.junit.org/`