Mining API Mapping for Language Migration

Hao Zhong[1,2]*, 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,txie}@ncsu.edu, zhanglu@sei.pku.edu.cn, wq@itechs.iscas.ac.cn

ABSTRACT

To address business requirements and to survive in competing markets, companies or open source organizations often have to release different versions of their projects in different languages. Manually migrating projects from one language to another (such as from Java to C#) is a tedious and error-prone task. To reduce manual effort or human errors, tools can be developed for automatic migration of projects from one language to another. However, these tools require the knowledge of how Application Programming Interfaces (APIs) of one language are mapped to APIs of the other language, referred to as API mapping relations. In this paper, we propose a novel approach, called MAM (**M**ining **A**PI **M**apping), that mines API mapping relations from one language to another using API client code. MAM accepts a set of projects each with two versions in two languages and mines API mapping relations between those two languages based on how APIs are used by the two versions. These mined API mapping relations assist in migration of projects from one language to another. We implemented a tool and conducted two evaluations to show the effectiveness of MAM. The results show that our tool mines 25,805 unique mapping relations of APIs between Java and C# with more than 80% accuracy. The results also show that mined API mapping relations help reduce 54.4% compilation errors and 43.0% defects during migration of projects with an existing migration tool, called Java2CSharp. The reduction in compilation errors and defects is due to our new mined mapping relations that are not available with the existing migration tool.

Categories and Subject Descriptors

D.2.13 Reusable Software [**Reusable Software**]: Reusable libraries

General Terms

API mapping relation, Language migration

1. INTRODUCTION

To address business requirements and to survive in competing markets, companies or open source organizations often have to release 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. For some open source projects, although their project teams do not officially provide multiple versions, external programmers often create their versions in different languages. For example, the WordNet team does not provide a C# version, but Simpson and Crowe developed a C# version of WordNet.Net[3]. As described by Jones [7], 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. In the literature, there exist approaches [8,18] and tools (*e.g.*, Java2CSharp[4]) that address the problem of language migration partially, since these approaches and tools expect programmers to describe how Application Programming Interfaces (APIs) of one language are mapped to APIs of another language. Since there are a large number of APIs provided in different languages, writing mappings manually for all APIs is tedious and error-prone. As a result, these approaches and tools support only a subset of APIs for migration. Such a limitation results in many compilation errors in migrated projects and limits these approaches' usage in practice (See Section 5.2 for details).

In this paper, we propose a novel approach, called MAM (**M**ining **A**PI **M**apping), that automatically mines how APIs of one language are mapped to APIs of another language. We refer to this mapping as *API mapping relations* (in short as relations in the rest of the paper). MAM mines relations based on API usages in client code rather than based on API implementations for three major reasons. (1) API libraries often do not come with source files, especially for those C# libraries. (2) Relations mined based on API implementations often have lower confidence than relations mined based on API usages. The reason is that API implementations have only one data point for analysis, whereas API usages can have many data points (*i.e.*, call sites) for mining. (3) Mapping relations of APIs are often complex and cannot be mined 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, whereas m_2 has two parameters p_1^2 and p_2^2. For these two API methods, 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 can be mapped to more

*Corresponding authors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...$10.00.

[1]`http://lucene.apache.org/`
[2]`http://wordnet.princeton.edu/`
[3]`http://opensource.ebswift.com/WordNet.Net/`
[4]`http://j2cstranslator.wiki.sourceforge.net`

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()

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 (represented as `E`), 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 invoked before invoking m_4.

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

To deal with the complexity of mining API mapping, we construct a graph, referred to 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 complex mapping relations of API methods.

This paper makes the following major contributions:

- The first approach, called MAM, that mines API mapping relations between different languages using API client code. MAM addresses an important and yet challenging problem that is not addressed by previous work on language migration.
- API transformation graphs (ATGs) proposed to capture inputs and outputs of API methods, and a technique for building ATGs and comparing built ATGs for mining API mapping relations. Since ATGs describe data dependencies among inputs and outputs of API methods, MAM is able to mine complex mapping relations between API methods of the two languages.
- A tool implemented for MAM and two evaluations on 15 projects that include 18,568 classes and 109,850 methods with both Java and C# versions. The results show that our tool mines 25,805 unique mapping relations (with more than 80% accuracy), including 6,695 mapping relations of API classes with accuracy of 86.7% and 19,110 mapping relations of API methods with accuracy of 90.0%. The results also show that the mined relations reduce 54.4% of compilation errors and 43.0% defects during migration of projects from Java to C# using Java2CSharp, an existing migration tool.

The rest of this paper is organized as follows. Section 2 presents definitions. Section 3 illustrates our approach using an example. Section 4 presents our approach. Section 5 presents our evaluation results. 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) [9] is a set of classes and methods provided by frameworks or libraries.

API library. An API library is a framework or library that provides reusable API classes and methods.

Client code. Client code is 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 classes and methods provided by J2SE[5]. Therefore, we consider Lucene as client code and J2SE as an API library. At the same time, Nutch[6] uses classes and methods provided by Lucene. Therefore, we consider Nutch as client code and Lucene as an API library. In general, for programmers of client code, source files of API libraries may not be available.

Mapping relation. For entities E_1 (such as API classes and methods) in a language L_1 and entities E_2 in another language L_2, a mapping relation is a triple $\langle E_1, E_2, b\rangle$ where migrating between E_1 and E_2 maintains the b behavior. The b behavior is specific to the type of the entities.

Mapping relation of API classes. For data entities whose type set is C_1 in L_1 and data entities whose type set is C_2 in L_2, a mapping relation of API classes is a triple $\langle C_1, C_2, s\rangle$, where migrating between C_1 and C_2 maintains the s behavior.

Since we use mapping relations of API classes for migrating data entities such as variables, parameters, and constants, we require that two mapped API classes have the same program behavior to store data, referred to as the s behavior. For example, the current time in `java.lang.System` of Java is stored in `System.DateTime` of C#, whereas the environment settings in `java.lang.System` of Java is stored in `System.Environment` of C#. Therefore, the Java class has a one-to-many mapping relation with two C# classes.

Mapping relation of API methods. For invoked API methods M_1 in L_1 and invoked API methods M_2 in L_2, a mapping relation of API methods is a triple $\langle M_1, M_2, t\rangle$, where migrating between M_1 and M_2 maintains the t behavior.

Since we use mapping relations of API methods for migrating API methods that accept inputs to produce desirable outputs, we require two mapped API methods have the same program behavior of inputs, outputs, and functionalities. We refer to this behavior as the t behavior. For example, Section 1 shows a one-to-many mapping relation between $\{m_3\}$ of Java and $\{m_4, m_5\}$ of C#.

3. EXAMPLE

We next use an example to illustrate challenges in mining API mapping relations. Figure 1 shows a Java code example and its migrated C# code. This Java code example accepts a `string` input that represents the 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 invokes the `exists` method. The method takes the `string` input and `file` as its inputs and produces the desirable `boolean` value. Here, we consider `file` (a receiver) as a special input for the `exists` method.

To migrate this code example into C#, a language migration tool needs to know mapping relations of API classes, so that it can migrate inputs, outputs, and variables into C#. For example, the migration tool needs to know the mapped API class in C# for `java.io.File` to migrate the variable `file` to C#. In addition, the migration tool needs to know the mapped API methods, so that it can add code for invoking proper API methods that take migrated inputs and variables to produce desirable outputs. For this example, the migration tool adds code for invoking the `Exists` method and the `FullName` method to achieve the functionality. Here, we consider field accesses as special types of method invocations.

To mine these mapping relations, MAM uses projects such as Lucene that have both Java and C# versions. MAM includes three major steps to mine the preceding two types of mapping relations of APIs from these projects.

Aligning client code. First, MAM aligns classes and methods between the two versions of each project. Since two code examples with the same functionality of two languages may exhibit mapping relations of APIs, this step aligns classes and methods by their functionalities. To achieve this goal, MAM uses a mapping algorithm based on similarities in the names of classes and methods defined

[5]`http://java.sun.com/j2se/1.5.0/`

[6]`http://lucene.apache.org/nutch/`

```
Java code:
1  File file = new File("test");
2  Boolean b = file.exists();
Migrated C# code:
3  FileInfo file = new FileInfo("test");
4  Boolean b = System.IO.File.Exists(file.FullName)||
            System.IO.Directory.Exists(file.FullName);
```

Figure 1: Java code and its migrated C# code

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

Figure 2: Two versions (Java and C#) of client code

in the two versions of each project.

Aligning client code based on the names of classes and methods is based on an observation on many existing projects such as rasp[7] migrated from one language to another. We observed that while migrating the rasp project from C# to Java, programmers first renamed source files from C# to Java and systematically addressed the compilation errors by replacing C# APIs with Java APIs. During this procedure, the names of classes, methods, fields of classes, or local variables in methods often remain the same or similar between the two versions. Therefore, we use name similarities for aligning client code of the two versions. For example, MAM aligns `IndexFiles.java` with the `IndexFiles.cs` (shown in Figure 2) since the names of their classes and methods are similar.

Mining API mapping of classes. Next, MAM mines mapping relations of API classes by comparing the names of data entities such as the names of fields in aligned classes, variables, or parameters in aligned methods. MAM uses name similarities for comparing the names of these entities. For example, MAM identifies the `args` parameters in Lines 7 (Java) and 12 (C#) (Figure 2) and maps the API classes that are the types of the two parameters. Based on this parameter, MAM maps the API class `java.lang.String` of Java to `System.String` of C#.

Mining API mapping of methods. After mapping API classes between the two languages, MAM maps API methods. Mapping API methods is challenging since 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 returns are mapped among these API methods. To address these challenges, MAM constructs a graph, referred to as *API Transformation Graph* (ATG), for each aligned method of the client code in both languages. These ATGs precisely capture inputs and outputs of API methods, and help mine mapping relations of API methods. For example, MAM mines a mapping relation from `java.io.File.Exists` in Java to `System.IO.File.Exists` and `System.IO.Directory.Exists` in C#. By a close look at the API document for these API methods, we can find that the Java method can check whether a file or directory exists, and this functionality is fulfilled by two C# API methods: `System.IO.File.Exists` for checking whether a file exists and `System.IO.Directory.Exists` for checking whether a directory exists. Section 4.2 presents more details on how we mine these mapping relations of API methods.

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

4. APPROACH

MAM accepts a set of projects as data sources for mining API mapping relations between two languages L_1 and L_2. For each project used as a data source, MAM requires two versions of the project (one version in L_1 and the other version in L_2). Figure 3 shows the overview of MAM.

4.1 Aligning Client Code

Initially, MAM accepts two versions of a project (one version in L_1 and the other version in L_2) and aligns classes and methods defined in the two versions. Aligned classes or methods between the two versions implement a similar functionality. Since these two versions implement a similar functionality, APIs used by these classes or methods could be replaceable.

To align classes and methods defined in the two versions, MAM uses name similarities between entities (such as class names or method names) defined by the two versions of the project. In MAM, 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 when naming entities such as classes, methods, and variables. Therefore, name similarity of the first kind is often helpful to distinguish functionalities compared to the second kind. MAM uses the Levenstein measure provided by SimMetrics[8] to calculate name similarities.

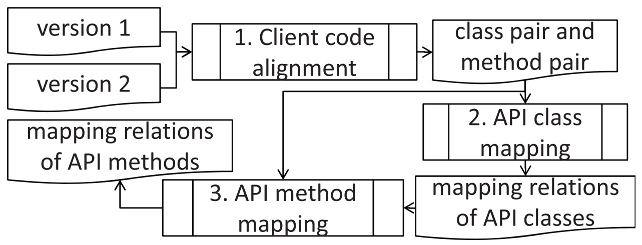
We next describe how MAM aligns client-code classes (i.e., classes defined in client code). The first step is to find candidate class pairs based on similarities of class names. For two sets of classes c and c' from two versions, MAM returns candidate class pairs M with name similarity greater than a given threshold, referred to as *SIM_THRESHOLD*. Since some projects may have more than one class with the same or similar name, M may contain more than one pair for a class in a version. To align these classes, MAM uses package names of these classes to refine M and returns only one pair with the maximum similarity for each class. For C#, we refer to namespace names for package names.

In each aligned class pair, MAM further aligns methods within the class pair. The alignment algorithm for methods is similar to the algorithm for classes and also may return more than one candidate method pair due to overloading. Here, the algorithm for methods relies on criteria such as the number of parameters and names of parameters to refine candidate method pairs. For the example shown in Section 3, MAM correctly aligns the class `IndexFiles` and the method `main` in Java to the class `IndexFiles` and the method `Main` in C#, respectively, since their names are quite similar. Since MAM returns pairs, MAM may sometimes fail to map some classes or methods (See Section 6 for details).

4.2 Mapping API classes

In the second step, MAM mines mapping relations of API classes. As mapping relations of API classes are used to migrate variables in language migration, MAM mines mapping relations of API classes based on how aligned client code declares variables such as fields of aligned classes, and parameters and local variables of aligned methods. For each aligned class pair $\langle c_1, c_2\rangle$, MAM analyzes each

[8]`http://sourceforge.net/projects/simmetrics/`


version 1
version 2
1. Client code alignment
class pair and method pair
2. API class mapping
mapping relations of API methods
3. API method mapping
mapping relations of API classes
Figure 3: Overview of MAM

field pair $\langle f_1, f_2\rangle$ and considers $\langle f_1.type, f_2.type\rangle$ as a relation, if the similarity between $f_1.name$ and $f_2.name$ is greater than *SIM_THRESHOLD*. Similarly, for each aligned method pair $\langle m_1, m_2\rangle$, MAM analyzes each local variable pair $\langle v_1, v_2\rangle$ and considers $\langle v_1.type, v_2.type\rangle$ as a relation, if the similarity between $v_1.name$ and $v_2.name$ is greater than *SIM_THRESHOLD*. MAM also analyzes each parameter pair $\langle p_1, p_2\rangle$ of m_1 and m_2, and considers $\langle p_1.type, p_2.type\rangle$ as a relation when the similarity between $p_1.name$ and $p_2.name$ is greater than *SIM_THRESHOLD*.

For the example shown in Figure 2, MAM mines the mapping relation between `java.lang.String` and `System.String` based on the mapped parameters of Lines 7 and 12. The mapping relation of API classes helps migrate the variable declared in Line 1 (Figure 1) to the variable declared in Line 3 (Figure 1).

4.3 Mapping API methods

In the third step, MAM first builds a graph for each client code method, and then compares the two graphs of each pair of client-code methods for mining mapping relations of API methods.

4.3.1 *API Transformation Graph*

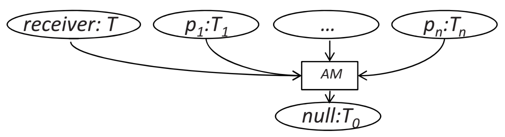
We propose API Transformation Graphs (ATGs) to capture API usages of client-code methods. Using ATGs has two main benefits for mining. First, an ATG describes inputs, outputs, and names of API methods and helps compare API methods from various dimensions. Second, an ATG describes data dependencies among API methods and helps mine mapping relations of multiple API methods.

An 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 F of m's declaring class, local variables V of m, parameters P_1 of m, parameters P_2 of API methods invoked by m, and returns R of all invoked methods. N_m is a set of all methods (both API methods and client-code methods) invoked 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 that d_2 is data-dependent on d_1, referred to as data dependency from d_1 to d_2. In our representation, a data-dependency edge is shown as a dotted line. In addition, an edge $d_1 \rightarrow m_1$ from a datum $d_1 \in N_{data}$ to a method $m_1 \in N_m$ denotes that d_1 is a parameter or receiver of m_1. Similarly, an edge $m_1 \rightarrow d_1$ from a method $m_1 \in N_m$ to a datum $d_1 \in N_{data}$ denotes that d_1 is the return of m_1.

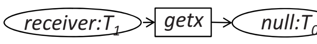
4.3.2 *Building API Transformation Graphs*

MAM builds an ATG for each method m defined in the client code, called a client-code method. ATG includes information such as inputs and outputs for each client-code method. In particular, for each client-code method m, MAM first builds subgraphs for its local variables, invoked API methods, and field accesses. MAM adds additional edges to the built ATG (and sub-graphs inside the ATG) and these additional edges represent data dependencies among built sub-graphs. We use two notations for representing nodes in the ATG. A rectangle represents a method labeled with the method name, whereas an ellipse represents a datum such as fields, local variables, and parameters. An ellipse is labeled as "*n:t*", where n is the name of the variable, t is its type, and "*null:t*" denotes return values, which have no names. We use the following rules for adding nodes and edges to the ATG.

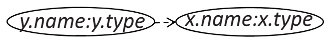
1. For each $f \in F \cup V \cup P_1$, MAM adds a node to the built ATG. The reason for considering these variables such as fields in the declaring class or local variables in method m used in client code is that these variables are useful to analyze data dependencies among API methods. Section 4.3.1 defines F, V, and P_1.
2. For each API method in the form "$T_0\ T.AM(T_1\ p_1, \ldots, T_n\ p_n)$" invoked by method m, MAM adds a receiver node (of type T), a return node, and parameter nodes to the built ATG as shown below. MAM does not add a receiver node for static API methods. Note that $T_0 \in R$ and $p_i \in P_2$.



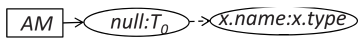
3. For each $f \in F \cup V$, if f is a non-primitive variable of type T_1 and a field x of T_1 is accessed as $f.x$, MAM adds nodes to the built ATG as shown below. Since Java often uses getters and setters whereas C# often uses field accesses, MAM treats field accesses as special types of method invocations.



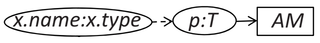
4. For each statement in the form $x = y$, where $x \in F \cup V \wedge y \in F \cup V$, MAM adds an edge from y to x. This edge represents that x is data-dependent on y.



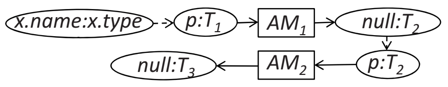
5. For each statement in the form $x = AM()$, where $x \in F \cup V$, MAM adds an edge from AM to x to indicate that the return of AM is assigned to x. This edge represents that x is data-dependent on the return of AM.



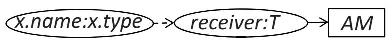
6. For each API method $AM(x)$ invoked by method m, MAM adds an edge from x to the parameter node of AM. This edge represents that the parameter of AM is data-dependent on x.



7. For each statement in the form $m_2(m_1(x))$, MAM adds an edge from the return node of m_1 to the parameter node of m_2. This edge represents that the parameter of m_2 is data-dependent on the return of m_1.



8. For each statement in the form $x.m()$, MAM adds an edge from x to m since x is the receiver of m. This edge represents that the receiver of m is data-dependent on x.



9. For each statement in the form $x = y\ op\ z\ op\ \ldots, op \in \{+, -, *, /\}$, MAM adds edges from y, z, and others to x, since these variables are connected by binary operations and the return is assigned to x. The edge denotes the data dependency from y, z, and other variables to x. For simplicity, MAM ignores *op* info. We discuss this issue in Section 6.

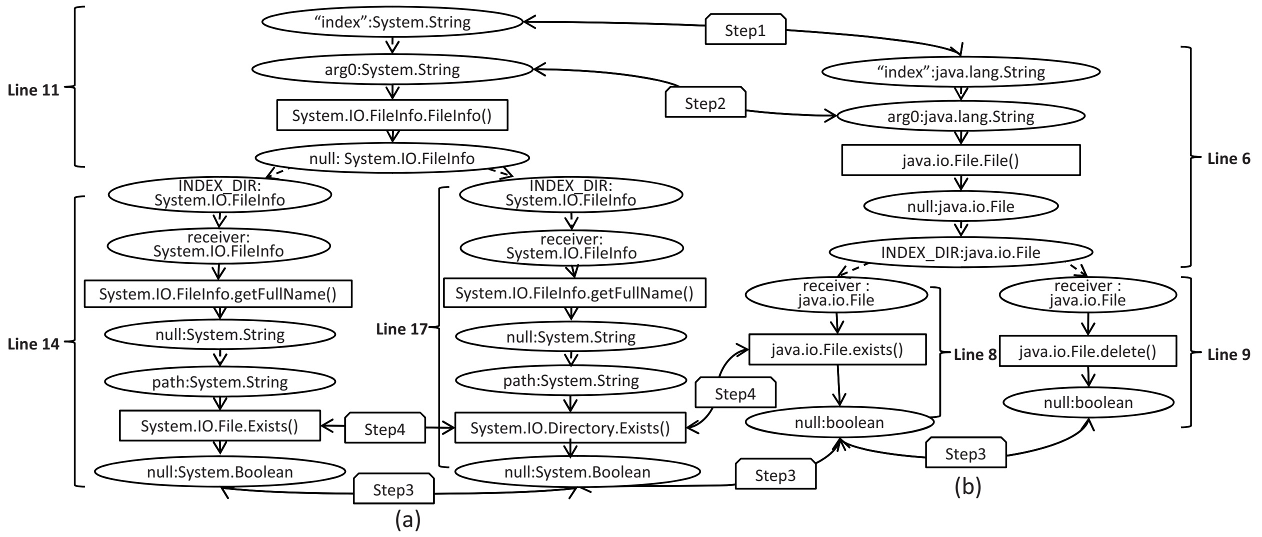
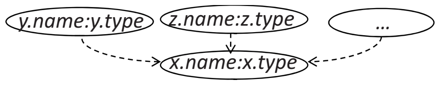


Figure 4: Built ATGs and the main steps of comparing ATGs



For each client-code method m, MAM applies the preceding rules for each statement from the beginning to the end of m's method body. Within each statement, MAM applies these rules based on their nesting depth in the abstract syntax tree. For example, for the statements of the form $m_2(m_1(x))$, MAM first applies these rules on m_1 and then on m_2. Our construction is similar to static slicing [14] with emphasis on API call sites.

Figures 4a and 4b show partial ATGs for the two `main` methods of `IndexFiles.cs` and `IndexFiles.java` shown in Figure 2, respectively. Figure 4 also shows corresponding line numbers of each sub-graph. MAM applies Rules 2 and 8 for Lines 6 and 9 (Figure 2) to build corresponding sub-graphs in the ATG. MAM applies Rules 2, 3, and 6 to build corresponding sub-graphs for Lines 11 and 14 (Figure 2). For simplicity, we do not show the nodes for the local variable `tmpBool` in the built ATGs in Figure 4.

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 should satisfy three criteria. (1) The mapped API methods implement the same functionality. (2) The mapping relation describes the relation between parameters and receivers of mapped API methods. (3) The mapping relation describes the relation between returns 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 1 presents major steps of comparing ATGs for mining mapping relations of API methods. For each pair of aligned methods m and m', the `findVarPairs` function finds mapped variables and constants as follows. For two variables v and $v' \in F$, V, and P_1 in m and m', respectively, `findVarPairs` maps v and v', if the similarity metric value on their names is greater than *SIM_THRESHOLD*. For constants in m and m', `findVarPairs` maps those two constants, if they have exactly the same value. From mapped variables and constants, our algorithm uses the following criteria to find mapping relations between the two API methods jm and sm.

Mapped inputs: The first criterion is based on the inputs of jm and sm. We map jm with sm, if there is a 1-to-1 mapping between

Algorithm 1: ATG Comparison Algorithm

```
Input: $G$ is the ATG of a client-code method $m$; $G'$ is the ATG of
    $m$'s aligned method $m'$.
Output: $S$ is a set of mapping relations for API methods
begin
    $P \leftarrow findVarPairs(m, m')$
    for Pair p in P do
        $JM \leftarrow G.nextMethods(p.j)$
        $SM \leftarrow G'.nextMethods(p.s)$
        $\Delta S = mapping(SM, JM); \Delta SM = \emptyset; \Delta JM = \emptyset$
        $S = \emptyset; S.addAll(\Delta S)$
        while $\Delta S \neq \emptyset || \Delta SM \neq \emptyset || \Delta JM \neq \emptyset$ do
            for Method jm in JM do
                if $jm.isMapped$ then
                    $\Delta JM = jm.nextMethod()$
                    $JM.delete(jm); JM.add(\Delta JM)$
                else
                    $\Delta JM = merge(jm, jm.nextMethod())$
                    $JM.delete(jm); JM.add(\Delta JM)$
            $\Delta S = mapping(SM, JM); S.addAll(\Delta S)$
            for Method sm in SM do
                if $sm.isMapped$ then
                    $\Delta SM = sm.nextMethod()$
                    $SM.delete(sm); SM.add(\Delta SM)$
                else
                    $\Delta SM = merge(sm, sm.nextMethod())$
                    $SM.delete(sm); SM.add(\Delta SM)$
            $\Delta S = mapping(SM, JM); S.addAll(\Delta S)$
end
```

inputs of jm and sm. Here, MAM considers both the receiver and the parameters as the inputs of an API method.

Mapped 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 metric value between the name of jm and the name of sm is greater than *SIM_THRESHOLD*.

Mapped outputs: The third criterion is based on the returns of jm and sm. Consider the returns 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 relations of API classes.

Our algorithm first attempts to map the first API method jm in m with the first API method sm invoked in m'. Our algorithm uses the `nextMethods` function to get these jm and sm API methods. In each iteration, this function merges an API method with its directly next API method to produce a merged API method. If our algorithm successfully maps jm with sm, our algorithm moves to the next available API methods invoked in client-code methods m and m'. If our algorithm is 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. Note that merged API methods are just synthesized intermediate representations for correlated API methods invoked in client code, and are used internally by our algorithm to construct many-to-many mapping relations of API methods. A merged API method cm is a sequence of API methods M combined by inputs and outputs, so the t behavior of cm is the combination of M. In the simplest case, a merged API method could include only one API method. For L_1, consider an API method m_1 defined in the class c_1 and an API method m_2 defined in c_2 with the following signatures:

m_1 `signature:` $o_1\ c_1.m_1(inp_1^1,\ inp_2^1,\ \ldots,\ inp_k^1)$
m_2 `signature:` $o_2\ c_2.m_2(inp_1^2,\ inp_2^2,\ \ldots,\ inp_l^2)$

Our algorithm merges 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 or as 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:` $o_2\ m_{new}(inp_1^1,\ inp_2^1,\ \ldots,\ inp_k^1,\ inp_1^2,\ inp_2^2,\ \ldots,\ inp_l^2)$

For example, our algorithm merges the API methods shown in Figure 4a to two merged API methods as follows.

```
cm1:Boolean {FileInfo,getFullName,File.Exist}(String)
cm2:Boolean {FileInfo,getFullName,Directory.Exist}(String)
```

Similarly, our algorithm merges the API methods shown in Figure 4b to two merged API methods as follows.

```
cm3:boolean {File,exist}(String)
cm4:boolean {File,delete}(String)
```

Our algorithm merges these API methods so that it is able to mine many-to-many mapping relations of API methods as shown in Figure 4. After merging API methods, our algorithm uses the `replace` function for merging an API method with its next available API method. For two merged API methods, our algorithm uses the maximum similarity of method names between jm and sm as a similarity metric value for mapping their functionalities. With each iteration, sm or jm or the mapping relation (represented as S) in the algorithm may change. 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 4. The numbers shown in rounded rectangles (such as *Step 1*) represent the major steps in our algorithm for mining mapping relations of API methods. We next explain each step in detail.

Step 1: 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 4, Step 1 maps two constants since both the constants have the same value "`index`".

Step 2: mapping inputs of API methods. Step 2 mines mapping relations of API methods using variable and constant mapping relations. Initially, this step identifies the first API methods in the two ATGs and tries to map their receiver and parameters of the two API methods. In our current example, this step maps the constructor parameter in Line 11 to the constructor parameter in Line 6 since these parameters are of the mapped types and their associated constants are mapped.

Step 3: mapping outputs of API methods. Step 3 maps returns of API methods. If MAM is not able to map returns, MAM merges the next API method and then attempts to map returns of merged API methods. In the example shown in Figure 4, MAM merges subsequent API methods in the ATG till the `Exists` API method, since the returns (shown as `Boolean`) can be mapped only after the `Exists` API method. Figure 4 shows Step 3 along with the mapped returns.

Step 4: mapping functionalities. After MAM maps parameters and returns, this step further maps functionalities of those merged API methods. Given two merged API methods with mapped parameters and returns, this step uses the similarity metric value based on their method names as a criterion for mapping their functionalities. In the preceding example, this step maps the two merged API methods shown in Figure 4a to the API method of `java.io.File.exists` since all three merged API methods include the method named `exists`.

After finding out the mapped pair of API methods as shown in Figure 4, MAM merges all variables and returns to corresponding parameters and receivers, and produces the mapping relation of API methods from `java.io.File.Exists` in Java to `System.IO.File.Exists` and `System.IO.Directory.Exists` in C# (See Section 5.2 for how mined mapping relations help language migration).

Project	**Source**	**Java version**		**C# version**	
		#C	***#M***	***#C***	***#M***
neodatis	SourceForge	913	6808	408	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		10283	73453	6744	34165

Table 1: Subjects

5. EVALUATIONS

We implemented a tool for MAM and conducted two evaluations using our tool to show the effectiveness of our approach. In our evaluations, we address the following two research questions:

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

Table 1 shows 15 open source projects with both Java and C# versions used as subjects in our evaluations. Column "Project" lists names of subjects. Column "Source" lists sources of these subjects. These subjects are collected from popular open source hosting sites such as SourgeForge[9], Apache[10], and hibernate[11]. Columns "Java version" and "C# version" list information for versions in Java and C#, respectively. In these two columns, sub-columns "*#C*" and "*#M*" list the number of classes and methods, respectively. As shown in the table, Java versions are much larger than C# versions

[9]`http://www.sf.net`
[10]`http://www.apache.org/`
[11]`http://www.hibernate.org/`

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%	93.1%	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%	47.6%	95.4%	89.9%	83	463
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	55.5%	32.3%	89.9%	70.2%	525	2150

Table 2: Results of aligning client code

Project	Class			Method		
	Num.	*Acc.*	*J2SE*	*Num.*	*Acc.*	*J2SE*
db4o	3155	83.3%	117	10787	90.0%	297
fpml	199	83.3%	41	508	83.3%	216
PDFClown	539	96.7%	36	514	100.0%	111
OpenFSM	64	86.7%	16	139	73.3%	12
binaryNotes	287	90.0%	31	671	90.0%	55
neodatis	526	96.7%	41	3517	100.0%	539
numerics4j	97	83.3%	2	429	83.3%	29
lucene	718	90.0%	83	2725	90.0%	522
logging	305	73.3%	45	56	90.0%	19
hibernate	1126	66.7%	87	7	13.3%	5
Total	6695	86.7%	344	19110	90.0%	1768

Table 3: Results of mining API mapping relations

for all subjects. We found two major factors for such a difference. First, Java versions of some of the projects are more up-to-date. For example, the latest Java version of *numericas4j* is 1.3, whereas the latest C# version is 1.2. Second, for some projects, migration from Java to C# is still in progress. For example, the website[12] 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 since our approach aids migrating projects from one language to other languages. In total, all these projects include 18,568 classes and 109,850 methods.

We conducted all evaluations on a PC with Intel Qual CPU @ 2.83GHz and 1.98G memory running Windows XP. More details of our evaluation results (*e.g.*, version numbers of subjects) are available at `http://tinyurl.com/yh98oeo`.

5.1 Mining API mapping relations

To investigate the first research question, we use the first 10 projects listed in Table 1 for mining API mapping relations.

Aligning client code. We first present the results of aligning client code. We use the *SIM_THRESHOLD* value as 0.6, which is set based on our initial empirical experience. We choose a relatively low threshold since it helps our approach to take into account as much client code as possible.

Table 2 shows our evaluation results. In column "Aligned", sub-columns "*# C*" and "*# M*" list the number of aligned classes and aligned methods, respectively. For each project of Columns "C# version" and "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 "Average" 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. The first category includes *db4o*, *fpml*, *PDFClown*, *OpenFSM*, and *binaryNotes*. In this category, our approach achieves relatively high percentages for both Java and C# versions. For all these five projects, "*%M*" is relatively smaller than "*%C*" because methods of those unaligned classes cannot be aligned and hence are counted as unaligned[13]. The second category includes *neodatis*, *numerics4j*, and *lucene*. In this category, our approach aligns C# versions well but does not align Java versions so well. We find that the migration of *neodatis* and *lucene* from Java to C# is in progress, and the Java version of *numerics4j* is more up 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 hence are not aligned. The third category includes *logging* and *hibernate*. In this category, 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 these two projects, the percentages of aligned classes are relatively high, and the percentages of aligned methods are relatively low. We find that even if our approach aligns a wrong class pair, our approach does not align methods within the wrong pair since the method names of a wrong pair are quite different. These results suggest that we could take method names into account when aligning classes in future work. For any of these these projects, our approach does not align all classes or all methods. We discuss these issues in Section 6.

In summary, as shown by Row "Average", our approach aligns classes and methods well 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 relations. Table 3 shows the results of mined mapping relations of API classes and methods. Columns "Class" and "Method" list results of mining mapping relations of API classes and API methods, respectively. Sub-column "*Num.*" lists the numbers of mined mapping relations. The numbers of mined API mapping relations are highly proportional to the sizes of projects shown in Table 1, except for *logging* and *hibernate*. Since classes and methods of these two projects are not well aligned, our approach does not mine many API mapping relations from these 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 first 30 mined API mapping relations (*i.e.*, percentages of correct mapping relations). For API mapping relations mined from each project, we manually inspect the first 30 mined API mapping relations by alphabetical order (sorted based on the names of their corresponding API classes and methods). We next classify these relations as correct or incorrect based on inspecting API client code and API documents. We find that our approach achieves high accuracies, except for *hibernate*. Although our approach does not align *logging* quite well either, the accuracies of API mapping relations mined from *logging* are still relatively high. To mine API mapping relations of classes, our approach requires that the names of fields in aligned classes, variables or parameters in aligned methods are similar. To mine API mapping relations of methods, our approach requires that two built ATGs are mapped. These two requirements are relatively strict. As a result, if the first step does not align client code well, our approach may miss some API mapping relations but does not introduce many false mapping relations. In other words, our approach is robust to mine accurate API mapping relations. Sub-column "*J2SE*" lists the number of mined API mapping relations between J2SE APIs and .NET frame-

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

[13]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.

Package	Class			Method		
	P	***R***	***F***	***P***	***R***	***F***
java.io	78.6%	73.3%	76.0%	93.1%	66.3%	79.7%
java.lang	82.6%	86.4%	84.5%	93.8%	81.5%	87.6%
java.math	50.0%	50.0%	50.0%	66.7%	66.7%	66.7%
java.net	100.0%	50.0%	75.0%	100.0%	50.0%	75.0%
java.sql	100.0%	66.7%	83.3%	100.0%	66.7%	83.3%
java.text	50.0%	50.0%	50.0%	50.0%	50.0%	50.0%
java.util	56.0%	87.5%	71.8%	65.8%	67.6%	66.7%
junit	100.0%	50.0%	75.0%	92.3%	88.9%	90.6%
orw.w3c	42.9%	75.0%	58.9%	41.2%	77.8%	59.5%
Total	68.8%	77.9%	73.4%	84.6%	73.9%	79.3%

Table 4: Results of comparison with manually written relations

work APIs. We next compare these API mapping relations with manually written mapping relations.

Row "Total" (in Table 3) lists the total result after we merge all duplicated mapping relations. In summary, our approach mines a large number of API mapping relations. These mined API mapping relations are accurate and associated with various libraries.

Comparing with manually written API mapping relations. Some migration tools such as Java2CSharp include manually written API 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 the mapping relation between `java.math.BigDecimal` of Java and `System.Decimal` of C#, and also describes the mapping relation between `multiply` of Java and `Multiply` of C#. For the two API methods, the pattern string describes the mapping relation of inputs: "`@0`" denotes the receiver of the `multiply` method, and "`@1`" denotes the first parameter of the `multiply` method. Based on this item, Java2CSharp migrates 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 written mapping files of Java2CSharp, we migrate our mined API mapping relations using the following strategy. First, for each Java class, we migrate its mapping relations of classes with the highest support values. Here, the support value of a mapping relation is the frequency that the mapping relation is mined from the subjects listed in Table 3. Second, for each Java method, we migrate its mapping relations of methods with the highest support values into mapping files as relations of methods with pattern strings. For 1-to-1 mapping relations of API methods, this step is automatic since mined mapping relations describe mapping relations of corresponding methods and inputs. For a few many-to-many mapping relations of API methods, this step is manual since mined mapping relations do not include adequate details.

The mapping files of Java2CSharp are associated with 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. Table 4 shows the comparison results of our mined API mapping relations within these mapping packages. Columns "Class" and "Method" list results of comparing API classes and methods, respectively. Sub-columns "*P*", "*R*", and "*F*" denote precision, recall, and F-score. *Precision*, *Recall*, and *F-score* are defined as follows[15]:

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

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

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

In these preceding formula, true positives represent those API mapping relations that exist in both the mined API mapping relations and the golden standard; false positives represent those relations that exist in the mined API mapping relations but not in the golden standard; false negatives represent those mapping relations that exist in the golden standard but not in the mined API mapping relations. Row "Total" shows the results when we compare mapping relations of all the packages listed in Table 4.

From sub-columns "*P*" of Table 4, we find that our approach achieves relatively high precisions, but the precisions are lower than the accuracies shown in Table 3. After inspecting those differences, we find 25 new correct mapping relations of API classes from our mined mapping files. For example, these mined mapping files contain a mapping relation between `org.w3c.dom.Attr` and `System.Xml.XmlAttribute`, and the mapping files of Java2CSharp do not contain mapping relations for the two API classes. If we consider these 25 new relations as true positives, the total precision would be 85.7%. These new mapping relations are useful and complements the existing mapping files of the Java2CSharp tool.

From sub-columns "*R*" of Table 4, we find that our approach achieves relatively high recalls, but the recalls still have space for improvements. For example, our approach does not mine the mapping relation between `java.util.ResourceBundle` and `System.Resources.ResourceManager` as described in the mapping files of Java2CSharp. Although this mapping exists in *hibernate*, our approach fails to mine the mapping relation since *hibernate* uses the two API classes in two classes with quite different names. Our approach also fails to mine the mapping relations between `java.util.getTime` and `System.DateTime.Ticks` correctly, since our approach cannot infer that 1 millisecond (returned by `getTime` of Java) is equal to 1000 ticks (returned by `Ticks` of C#).

In summary, compared with the mapping files of Java2CSharp, our mined mapping files show reasonably high precisions and recalls. The precisions are relatively high since our mined mapping relations are accurate and include new mapping relations that are not covered by Java2CSharp. The recalls are also relatively high since our approach mines many mapping relations although we still have space for further improvements.

Projects	No MF		MF		Ext. MF			
	E	***D***	***E***	***D***	***E***	***%E***	***D***	***%D***
rasp	973	159	708	123	627	11.4	93	24.4
llrp	2328	122	1540	114	269	82.5	42	63.2
SimM	217	13	12	0	6	50.0	0	0
aligner	368	34	289	0	262	9.3	0	0
fit	177	29	27	0	10	25.9	0	0
Total	4063	357	2576	237	1174	54.4	135	43.0

Table 5: Compilation errors and defects (SimM: SimMetrics)

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

[15]We ignore those API mapping relations that do not have call sites in the projects listed in Table 3, since our approach relies on API call sites to mine mapping relations. By adding more projects using these APIs to our subjects, our approach can then mine relations of these APIs.

5.2 Aiding Language Migration

To investigate the second research question, we feed the mined API mapping relations to the Java2CSharp tool and investigate whether these relations can improve the tool's effectiveness. We choose this tool because this tool is a relatively mature project at ILOG[16] (now part of IBM) and supports the extension of user-defined mapping relations of APIs.

We use Java2CSharp to migrate the last five projects listed in Table 1 from Java to C#, and Table 5 shows the results. For each migrated C# project, Column "No MF" lists results without mapping files. Column "MF" lists results with only the mapping files of Java2CSharp. Column "Ext. MF" lists results with mapping files that combine our mined API mapping relations with the existing mapping files of Java2CSharp. Sub-columns "*E*" and "*D*" list the number of compilation errors and API related defects found, respectively. Compilation errors provide an overview for qualities of migrated projects, but some compilation errors are not related to APIs. To find out the effectiveness of reducing API related defects, we manually inspect some migrated C# files. In particular, for each project, we first select those overlapping files between migrated files and existing C# files from the C# version of the five projects. After that, we manually compare the top five largest files among these overlapping files with existing C# files and analyze those differences for detecting API related defects. For example, a migrated C# statement of *SimMetrics* in "No MF" is as follows.

```
totalDistance = (float)Java.Lang.Math.Sqrt(totalDistance);
```

This statement contains an API related defect since Java2CSharp does not migrate the `sqrt` method of Java to its corresponding API method of C#. Sub-columns "*%E*" and "*%D*" list percentages of improvements over the results of "MF". On average, mined API mapping relations help further reduce 54.4% compilation errors and 43.0% API related defects. Since the five projects use different libraries, compilation errors and defects in migrated projects are different. In particular, *SimMetrics* and *fit* use API classes of J2SE that are covered by mapping files. Consequently, the migrated projects of *SimMetrics* and *fit* have only a few errors and defects. 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 Java2CSharp do not cover any classes of `java.awt`, so the migrated project has many compilation errors. Since the existing C# version of *aligner* does not have GUI, we do not compare those defective migrated GUI files, and we do not find any API related defects in the compared five files. The mined files map `java.awt` to `System.Windows.Forms` and thus reduce compilation errors. However, the result is not significant since many classes of the two packages are still not mapped. For *rasp* and *llrp*, they both use various libraries besides J2SE. Consequently, the migrated projects have both many errors and API related defects. In particular, *llrp* uses log4j[17] and jdom[18], and the mined mapping files contain mapping relations of the two libraries. As a result, the mined API mapping relations help reduce compilation errors and API related defects significantly. *rasp* uses some libraries such as Neethi[19]. Since the used subjects for mining and thus our mined API mapping relations do not cover these libraries, the migrated project of *rasp* contains many compilation errors and API related defects.

In summary, the mined API mapping relations help improve existing language migration tools such as Java2CSharp. In particular, the mined API mapping relations help effectively reduce compilation errors and API related defects in the migrated projects.

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

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

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

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

5.3 Threats to Validity

The threats to external validity include the representativeness of the subjects and the used migration tool. Although we applied our approach on 10 projects for mining API mapping relations and on additional 5 projects for helping language migration, our approach is evaluated only on these limited projects. Although Java2CSharp is the best migration tool within our knowledge, other tools may perform better than Java2CSharp. This threat could be reduced by more evaluations on more subjects and more migration tools in future work. The threats to internal validity include human factors for determining correct mined API mapping relations and for determining API related defects in migrated projects. To reduce these threats, we inspected mined mapping relations and API related defects carefully, and we referred to existing mapping relations and existing C# files for determining correct API mapping relations and API related defects, respectively. The former threat could be further reduced by comparing with more existing mapping relations of APIs as we did for J2SE. The latter threat could be reduced by running test cases to detect API related defects.

6. DISCUSSION AND FUTURE WORK

We next discuss issues in our approach and describe how we address these issues in our future work.

Aligning client code. Table 2 shows that our approach could not align client code in a few cases. The primary reason is that the functionality associated with a class or a method in one language version is split among multiple classes or methods in the other language version. To address this issue, we plan to align classes and methods of client code based on their functionalities through developing or adapting dynamic approaches [6] in future work.

Mining richer API mapping. Table 4 shows that our approach still has space to improve recalls. Although we use 10 large projects as subjects, these projects still do not provide sufficient code examples for mining mapping relations of all APIs in J2SE. Our previous work [11] shows that it is feasible to use large-scale repositories available on the web as subjects with the help of code search engines. In future work, we plan to leverage these code search engines to mine richer API mapping.

Migrating many-to-many mapping relations of API methods. A mined many-to-many mapping relation of API methods can have multiple outputs and complex internal data processes. Although our ATGs help identify all API methods, our implementation is not complete for supporting automatic migration. For example, we need to manually add an *or* operator for the two outputs of the API mapping presented in Section 3 (*i.e.*, the returns of `System.IO.File.Exists` and `System.IO.Directory.Exists`). In future work, we plan to enhance our implementation to help automate migration with many-to-many mapping relations.

Migrating unmapped APIs. Our approach mines API mapping of methods along with the mappings of their inputs and outputs. These mappings are useful for migrating API methods of one language to another. Sometimes, our approach may not be able to map inputs and outputs of mapped API methods. If our approach is not able to map outputs, our approach currently simply ignores those outputs that are not used in the client code. However, since inputs cannot be ignored, the migrated code has compilation errors. In future work, we plan to address this issue by analyzing how two versions of a project deal with a similar unmapped API problem for some other code examples.

7. RELATED WORK

Our approach is related to previous work on two areas: language migration and library migration.

Language migration. To reduce manual efforts of language migration [10], researchers proposed various approaches [4, 8, 15, 16, 18] to automate the process. However, all these approaches focus on the syntax or structural differences between languages. Deursen *et al.* [15] proposed an approach to identify objects in legacy code. Their approach uses these objects to deal with the differences between object-oriented and procedural languages. As shown in El-Ramly *et al.* [3]'s experience report, existing approaches support only a subset of APIs for language migration, making the task of language migration a challenging problem. In contrast to previous approaches, our approach automatically mines API mapping between languages to aid language migration, addressing a significant problem not addressed by the previous approaches and complementing these approaches.

Library migration. With evolution of libraries, some APIs may become incompatible across library versions. To address this problem, Henkel and Diwan [5] proposed an approach that captures and replays API refactoring actions to update the client code. Xing and Stroulia [17] proposed an approach that recognizes the changes of APIs by comparing the differences between two versions of libraries. Balaban *et al.* [2] proposed an approach to migrate client code when mapping relations of libraries are available. In contrast to these approaches, our approach focuses on mapping relations of APIs across different languages. In addition, since our approach uses ATGs to mine API mapping relations, our approach can also mine mapping relations between API methods with different parameters or between API methods whose functionalities are split among several API methods in the other language.

Mining specifications. Some of our previous approaches [1, 12, 13, 19, 20] focus on mining specifications. MAM mines API mapping relations across different languages for language migration, whereas the previous approaches mine API properties of a single language to detect defects or to assist programming.

8. CONCLUSION

Mapping relations of APIs are quite useful for the migration of projects from one language to another language, and it is difficult to mine these mapping relations due to various challenges. In this paper, we propose a novel approach that mines mapping relations of APIs from existing projects with multiple versions in different languages. We conducted two evaluations to show the effectiveness of our approach. The results show that our approach mines many API mapping relations between Java and C#, and these relations improve existing language migration tools such as Java2CSharp.

Acknowledgments

Hao Zhong is grateful to his advisor, Prof. Hong Mei, for the insightful advice on research during his PhD program in Peking University. The authors from Chinese Academy of Sciences are sponsored by the National Basic Research Program of China (973) No. 2007CB310802, the Hi-Tech Research and Development Plan of China (863) No. 2007AA010303, and the National Natural Science Foundation of China No. 90718042. The authors from NCSU are supported in part by NSF grant CCF-0725190, NSF grant CCF-0845272, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI). Lu Zhang is sponsored by the National Basic Research Program of China (973) No. 2009CB320703, the High-Tech Research and Development Program of China (863) No. 2007AA010301, the Science Fund for Creative Research Groups of China No. 60821003, and the National Science Foundation of China No. 90718016.

9. REFERENCES

[1] M. Acharya and T. Xie. Mining API error-handling specifications from source code. In *Proc. FASE*, pages 370–384, 2009.

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

[3] 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.

[4] A. Hassan and R. Holt. A lightweight approach for migrating Web frameworks. *Information and Software Technology*, 47(8):521–532, 2005.

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

[6] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proc. 18th ISSTA*, pages 81–92, 2009.

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

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

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

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

[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, 2007.

[12] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proc. 31th ICSE*, pages 496–506, May 2009.

[13] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proc. 7th ESEC/FSE*, pages 193–202, 2009.

[14] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

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

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

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

[18] 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.

[19] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. the 23rd ECOOP*, pages 318–343, 2009.

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