Automated Testing of API Mapping Relations

Hao Zhong
Laboratory for Internet Software Technologies
Institute of Software
Chinese Academy of Sciences, China
zhonghao@nfs.iscas.ac.cn

Suresh Thummalapenta and Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
{sthumma,txie}@ncsu.edu

ABSTRACT

Software companies or open source organizations often release their applications in different languages to address business requirements such as platform independence. To produce the same applications in different languages, existing applications already in one language such as Java are translated to applications in a different language such as C#. To translate applications from one language (L_1) to another language (L_2), programmers often use automatic translation tools. These translation tools use Application Programming Interface (API) mapping relations from L_1 to L_2 as a basis for translation. It is essential that the API methods of L_1 and their mapped API methods of L_2 (as described by API mapping relations) exhibit the same behavior, since any inconsistencies among these mapping relations could result in different behaviors between the original and translated applications. Therefore, to detect behavioral differences between API methods described in mapping relations, and thereby to effectively translate applications, we propose the first novel approach, called *TeMAPI* (**Te**sting **M**apping relations of **API**s). In particular, given a translation tool from Java to C# (or vice versa), TeMAPI automatically generates test cases that expose differences among mapping relations described in the tool. To show the effectiveness of our approach, we applied our approach on five popular translation tools. The results show that TeMAPI effectively detects various behavioral differences among mapping relations. We summarize detected differences as eight findings and their implications. We expect that our findings and implications help improve automatic translation tools in effectively translating applications from one language to another, and also assist programmers in understanding the differences among API methods between Java and C#.

1. INTRODUCTION

Since the inception of computer science, many programming languages (*e.g.*, Cobol, Fortran, or Java) have been introduced to serve specific requirements[1]. For example, Cobol is introduced specifically for developing business applications. In general, software companies or open source organizations often release their applications in different languages to survive in competing markets and to address various business requirements such as platform independence. A recent study [9] shows that nearly one third applications exist in different languages. A natural way to implement a application in a different language is to translate from existing applications. For example, Lucene.Net is declared to be translated from Java Lucene in its website[2]. For another example, the NeoDatis object database is also being translated form Java to C# as declared by its website[3].

[1]http://hopl.murdoch.edu.au

```
private long readLong(ByteArrayInputStream is){
 ...
 l += ((long) (is.read())) << i;
...}
```

Figure 1: A method in the Java version of db4o.

```
private long ReadLong(ByteArrayInputStream @is){
 ...
 l += ((long)(@is.Read())) << i;
 ...}
```

Figure 2: A method in the C# version of db4o.

As existing applications typically use API libraries, it is essential to understand API mapping relations of one programming language, referred to as L_1, to another language, referred to as L_2 when translating applications from L_1 to L_2. As pointed out by our previous work [24], API mapping relations can be complicated. In some cases, programmers may fail to find a existing class that has the same behavior in the other language. For example, Figures 1 and 2 show two methods implemented in db4o[4] of its Java version and its C# version, respectively. When translating the code shown in Figure 1 into C#, programmers of db4o may fail to find an existing C# class that have the same behaviors with the `ByteArrayInputStream` class in Java, so they implement a C# class with the same name to hide the behavior difference. Behavior differences of mapped API invocations may occur in many places since API invocations are often quite large in size. To relief efforts, programmers of db4o developed their own translation tool, called sharpen[5], for translating db4o from java to C#. In particular, sharpen systematically replaces all API invocations in Java with equivalent invocations in C# to ensure that translated C# applications have the same behaviors with original ones.

In practice, programmers may fail to hide some behavior differences of mapped API invocations. These behavior differences are harmful since they may accumulate and cause serious defects in translated applications silently. It is desirable to detect these different behaviors, but existing approaches [5, 11] cannot detect such differences effectively with four major reasons. (1) Existing ap-

[2]http://lucene.apache.org/lucene.net/
[3]http://wiki.neodatis.org/
[4]http://www.db4o.com
[5]http://tinyurl.com/22rsnsk

```
09: DatagramSocket socket = ...;
10: DatagramPacket package = ...;
11: socket.receive(package);
```

Figure 3: Sample code in Java.

```
12: UdpClient socket = ...;
13: IPEndPoint remoteIpEndPoint = ...;
14: try{
15:  byte[] data_in = socket.Receive(ref remoteIpEndPoint);
16:  PacketSupport tempPacket =
         new PacketSupport(data_in, data_in.Length);
17:   tempPacket.IPEndPoint = remoteIpEndPoint;
18: } catch (System.Exception e){...}
19: PacketSupport package = tempPacket;
```

Figure 4: Translated C# code by JLCA.

proaches require both the versions under consideration belong to the same language, but in our context, both versions belong to different languages, making these existing approaches inapplicable. (2) Existing approach may not effectively generate test cases to detect behavior differences of mapped API invocations since there is no report that thoroughly describes where such differences occur, to the best of our knowledge. (3) API libraries typically provide many classes. Detecting behavior differences from existing applications address the problem partially since applications typically use a small part of API classes. (4) It requires aligning inputs of API methods if we are to compare their outputs. In some cases, one API method is translated to multiple methods, and it becomes difficult to align their inputs. For example, JLCA[6] translates the code as shown Figure 3 into the code as shown in Figure 4. We find that it translates the API invocation in Java of Line 11 into many API invocations in C#.

To address these preceding issues, we propose a novel approach, called TeMAPI (**Te**sting **Ma**pping relations of **API**s), that generates test cases to detect behavioral differences among API mapping relations automatically. In particular, TeMAPI generates test cases on one version App_1 of the application (in a language) and translates those test cases into the other language L_2. TeMAPI next applies translated test cases on the other version App_2 to detect behavioral differences. In this paper, we primarily focus on behavioral differences that can be observed via return values of API methods or exceptions thrown by API methods. TeMAPI addresses two major technical challenges in effectively detecting behavioral differences. (1) Using a naive technique such as generating test cases with `null` values may not be significant in detecting behavioral differences among API mapping relations. Since we focus on object-oriented languages such as Java or C#, to detect behavioral differences, generated test cases need to exercise various object states, which can be achieved using method-call sequences. To address this issue, TeMAPI leverages two existing state-of-the-art test generation techniques: random [14] and dynamic-symbolic-execution-based [6, 10, 18]. (2) Generating test cases on App_1 and applying those test cases on App_2 may not exercise many behaviors of API methods in App_2, thereby related defects cannot be detected. To address this issue, TeMAPI uses a round-trip technique that also generates test cases on App_2 and applies them on App_1. We describe more details of our approach and how we address these challenges in subsequent sections.

This paper makes the following major contributions:

- A novel approach, called TeMAPI, that automatically generates test cases that detect behavioral differences among API mapping relations. Given a translation tool, TeMAPI detects different behaviors of its all API mapping relations automatically. It is important to detect these different behaviors, since they can introduce defects in translated code silently.
- Test adequacy criteria proposed for generating sufficient test cases to test API mapping. TeMAPI targets at generating adequate test cases that can reveal all behaviors of API methods to test their mapping relations.
- A tool implemented for TeMAPI and three evaluations on 5 migration tools that include 3 Java-to-C# tools and 2 C#-to-Java tools. The results show that various factors such as null values, string values, input ranges, different understanding, exception handling, static values, cast statements, and method sequences can lead to different behaviors of mapped API methods. Based on the results, we further analyze the implications of these factors for programmers and for migration tool developers.

The rest of this paper is organized as follows. Section 2 presents an illustrative example. Section 3 presents our approach. Section 4 presents our evaluation results. Section 5 discusses issues of our approach. Section 6 presents related work. Section 7 concludes.

2. EXAMPLE

We next present major steps in our approach for detecting the behavioral differences of mapped API invocations. In particular, we use the `java.io.BufferedInputStream` class and the JLCA translation tool as illustrative examples.

Translating generated wrapper methods. As described by its documentation[7], the `BufferedInputStream` class in Java has 5 fields, 2 constructors, and 8 methods. To fully explore the behaviors of the class, TeMAPI generates a wrapper method for each field and method given each constructor. For example, given the `Buffered-InputStream(InputStream)` constructor, TeMAPI generates the wrapper method as follows for the `skip(long)` method:

```
public long testskip24nm(long m0,InputStream c0){
  BufferedInputStream obj = new BufferedInputStream(c0);
  return obj.skip(m0);
}
```

Each wrapper method check the return value of only one API method or field. If outputs are different given the same inputs, it is easy to locate the method or field with behavior differences and to find the inputs that cause behavior differences. TeMAPI next uses JLCA and translates generated wrapper from Java to C#.

Handling compilation errors. In general, a translation tool may not include mapping relations for all API methods of a language. Therefore, translated wrapper methods can have compilation errors. TeMAPI parses translated wrapper methods and removes all wrapper methods with compilation errors. For example, following is the translated C# `testskip24nm` method:

```
public virtual long testskip24nm(long m0, Stream c0){
  BufferedStream obj = new BufferedStream(c0);
  BufferedStream temp_BufferedStream = obj;;
  Int64 temp_Int64 = temp_BufferedStream.Position;
  temp_Int64 = temp_BufferedStream.Seek(m0,
        System.IO.SeekOrigin.Current) - temp_Int64;
  return temp_Int64;
}
```

TeMAPI does not removes this method since it has no compilation errors. Although JLCA translates the `skip(long)` method into multiple methods, we can still test the mapping relation since all translated code is within the wrapper method and a translation tool does not change the signatures of a wrapper method.

Generating test cases. TeMAPI leverages various techniques to generate test cases for the remaining wrapper methods. For example, TeMAPI extends Pex [18], a state-of-the-art test generation technique, to record all inputs and their corresponding outputs generated for each translated wrapper method. Each input generated

[6]http://tinyurl.com/35o5mo7

[7]http://tinyurl.com/2bca7vh

by Pex exercises a unique feasible path in the wrapper method. For example, TeMAPI generates the following JUnit test case based on inputs generated by Pex.

```
public void testskip24nm36(){
  try{
     sketch.Test_java_io_BufferedInputStream obj =
        new sketch.Test_java_io_BufferedInputStream();
     long m0 = java.lang.Long.valueOf(
                 "-9223372036582079488").longValue();
     InputStream c0 = new InputStream(null);
     obj.testskip24nm(m0,c0);
  }catch(java.io.IOException e){
     Assert.assertTrue(true);return;
  }
  Assert.assertTrue(false);
}
```

This test case fails, since given the preceding inputs, the `skip (long)` method does not throw any exceptions as the translated C# code does. Thus TeMAPI detects a behavioral difference between the `skip(long)` method in Java and its mapped C# API methods defined by JLCA.

Besides extending Pex, TeMAPI also extends Randoop [14] to generate invocation sequences. TeMAPI extracts the list of translated wrapper methods without compilation errors, and analyzes generated wrappers for a list of translatable API methods and fields. When generating test cases, TeMAPI limits the search scope of Randoop, so that each generate test case invoke only translatable API methods and fields. For example, a generated JUnit test case is as follows:

```
public void test413() throws Throwable{
  ...
  ByteArrayInputStream var2=new ByteArrayInputStream(...);
  var2.close();
  int var5=var2.available();
  assertTrue(var5 == 1);
}
```

TeMAPI next uses JLCA and translates the generated JUnit test case from Java to C#. As each wrapper method invokes only translatable API invocations, we increase the chance of translating successfully. The translated NUnit test case is as follows:

```
public void test413() throws Throwable{
  ...
  MemoryStream var2 = new MemoryStream(...);
  var2.close();
  long available = var2.Length - var2.Position;
  int var5 = (int) available;
  UnitTool.assertTrue(var5 == 1);
}
```

The preceding JUnit test case gets passed, but the NUnit test case get failed. We find that Java allows programmers to access a stream even if the stream is closed, so the preceding JUnit test case run successfully. C# does not allow such accesses, so the preceding NUnit test case fails with `ObjectDisposedException`. This example motivates our basic idea of generating test cases in one language and translating those test cases into another language for detecting differences among API mapping relations. We next present details of our approach.

3. APPROACH

Given a migration tool between Java and C#, TeMAPI generates various test cases to reveal different behaviors of the tool's API mapping relations. Figure 5 shows the overview of TeMAPI. It is able to test not only mapping relations of a single API invocation but also mapping relations of multiple API invocations.

3.1 Translating Generated client code

Given a migration tool, TeMAPI first extracts its translatable API invocations. It is challenging to extract such invocations directly from migration tools for two factors: (1) different migration

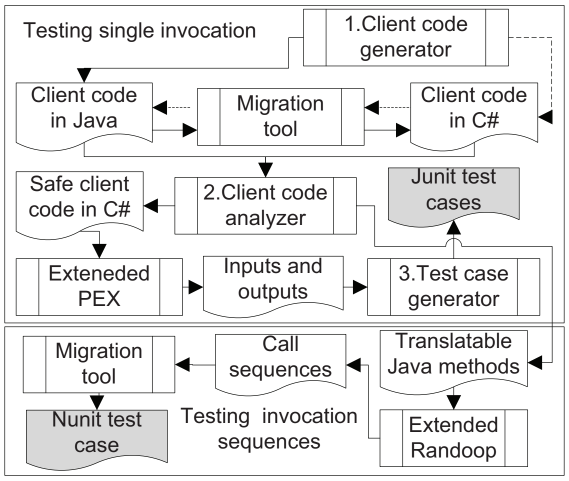

Figure 5: Overview of TeMAPI

tools may follow different styles to describe API mapping relations. For example, as shown in Section 1, the API mapping relations of Java2CSharp are described in its mapping files, but the API mapping relations of sharpen are hard-coded in its source files. (2) commercial migration tools typically hide their API mapping relations in binary files. Due to the two factors, TeMAPI does not extract API mapping relations directly from migration tools, but chooses to analyze translated code of migration tools. In particular, TeMAPI generates a client-code method for each API invocation and analyzes translated results for translatable API invocations. We do not use migration tools to translate existing projects for two considerations: (1) existing projects typically contain quite a small set of API invocations, so many API mapping relations may be not covered; (2) a single method of an existing project may invoke multiple API methods, so it may be difficult to analyze which methods are not mapped. TeMAPI relies on the reflection technique [12] provided by both Java and C# to generate client code for translation.

Static fields. Given a public static field `f` of a class `C` whose type is `T`, TeMAPI generates a getter as follows:

```
public T TestGet|f.name||no|(){ return C.f; }
```

If `f` is not a constant, TeMAPI generates a setter as follows:

```
public void TestSet|f.name||no|(T v){ C.f = v; }
```

Non-static fields. Given a public non-static field `f` of a class `C` whose type is `T`, TeMAPI generates a getter for each constructor `C(T1 p1,..., Tn pn)` of `C` as follows:

```
public T TestGet|f.name||no|(T1 c1,..., Tn cn){
   C obj = new C(c1,..., cn);
   return obj.f; }
```

If `f` is not a constant, TeMAPI generates a setter as follows:

```
public void TestSet|f.name||no|(T1 c1,..., Tn cn){
  C obj = new C(c1,..., cn);
  obj.f = v; }
```

In the preceding code, "`|f.name|`" denotes the name of `f`, and "`|no|`" denotes the corresponding number of generated client-code method.

Static methods. Given a public static method `m(T1 p1,...,Tn pn)` of a class `C` whose return type is `Tm`, TeMAPI generates a client-code method as follows:

```
public Tm Test|m.name||no|(T1 m1,..., Tn mn){
  return C.m(m1,..., mn); }
```

Non-static methods. Given a public non-static method `m(T1 p1,...,Tn pn)` of a class `C` whose return type is `Tm`, TeMAPI generates a client-code method for each constructor `C(Tv pv,..., Tt pt)` of `C` as follows:

```
public Tm Test|m.name||no|(T1 m1,..., Tn mn,
                             Tv cv, ..., Tt ct){
  C obj = new C(cv,..., ct);
  return obj.m(m1,..., mn); }
```

In the preceding code, "`|m.name|`" denotes the name of `m(T1 p1,...,Tn pn)`.

TeMAPI put all generated client code methods for one API class C into one generated class. For a Java-to-C# tools, TeMAPI generates client code in Java as shown by the solid line in Figure 5, and for a C#-to-Java tools, TeMAPI generates client code in C# as shown by the dotted line in Figure 5. When generating methods, TeMAPI ignores generic API methods for simplicity. Besides, it ignores `unsafe` and `delegate` API methods and API methods whose parameters are marked as `out` or `ref` when generating C# methods. Java does not have corresponding keywords, so there are typically not mapped. After TeMAPI generate client-code methods, we translate them using a migration tool under experiments.

After generated code is translated, TeMAPI parses translated code and removes translated methods with compilation errors. For Java code, TeMAPI extends Eclipse's Java compiler to extract methods with compilation errors. For C# code, TeMAPI extracts the list of compilation errors through the automation and extensibility interfaces of Visual Studio. Using remained methods, TeMAPI tests single API invocations (Section 3.2). From remained methods, TeMAPI also analyzes the list of translatable API invocations for a given migration tool. Using the list, TeMAPI further tests API invocation sequences (Section 3.3).

3.2 Testing Single API Invocations

Pex [18] is a white-box test generation tool for .Net based on dynamic symbolic execution. Basically, Pex repeatedly executes a method under test, so that it explores all feasible paths of the method. To reduce the efforts to explore paths, Pex leverage various search strategies. For example, Xie *et al.* [21] propose a search strategy called Fitnex that uses state-dependent fitness values to guide path exploration of Pex.

TeMAPI extends Pex, so that it generates test cases that satisfy the path criterion defined in Section **??**. In particular, for each translated C# client code method, TeMAPI records the inputs and the corresponding output when Pex searches each path. Based on recorded inputs and outputs, TeMAPI generates JUnit test cases to ensure each mapped API invocations produce the same output give the same inputs. For example, TeMAPI records that given a integer whose value is 0, the output of the `TestvalueOf57sm` method in C# is "0". Based on the preceding input and output, TeMAPI generates a JUnit test case for the `testvalueOf57sm` method in Java as follows:

```
public void testvalueOf57sm7(){
  sketch.Test_java_lang_String obj =
      new sketch.Test_java_lang_String();
  int m0 = 0;
  Assert.assertEquals("0", obj.testvalueOf57sm7(m0));}
```

This JUnit test case runs successfully, and the result ensures that the `testvalueOf57sm` method in Java has the same output with the `TestvalueOf57sm` method in C# given the same input.

We find that when Pex searches a path with some specific inputs, the method under test throws exceptions. For example, TeMAPI records that if the input of the `TestvalueOf61sm` method in C# is null, the method throws `NullReferenceException`. TeMAPI also generates a JUnit test case to ensure the `testvalueOf61sm` method in Java also throws a mapped exception. To generate the JUnit test case, TeMAPI first finds the corresponding exceptions in Java by analyzing translated client code with generated code. For example, TeMAPI finds that the `NullReferenceException` class in C# is mapped to the `NullPointerException` class in Java with respect to the API mapping relations of Java2CSharp, so it generates a JUnit test case as follows:

```
public void testvalueOf61sm3(){
  try{
    sketch.Test_java_lang_String obj =
          new sketch.Test_java_lang_String();
    java.lang.Object m0 = null;
    obj.testvalueOf61sm(m0);
  }catch(java.lang.NullPointerException e){
    Assert.assertTrue(true);
    return;
  }
  Assert.assertTrue(false); }
```

This JUnit test case fails since the `testvalueOf61sm` method does not throw any exceptions given a null input. From this failed JUnit test case, TeMAPI detects the different behavior between the `testvalueOf61sm` method in Java and the `TestvalueOf61sm` method in C#. Since the two methods call only one API method, TeMAPI thus detects the different behavior between the `valueOf (Object)` in Java and its mapped C# method.

3.3 Testing Invocation Sequences

Randoop [14] is a test generation tool for Java. It randomly generates test cases based on already generated test cases in a feedback-directed manner. As stated by Tillmann and Halleux [18], the key difference between Pex and Randoop lies in that Pex generates test cases for a single method whereas Randoop generates test cases that involve multiple methods.

TeMAPI extends Randoop, so that it generates test cases that satisfy the sequence criterion defined in Section **??**. As described in Section 3.1, TeMAPI is able to extract translatable API invocations of a given migration tool. TeMAPI restricts the search scope of Randoop, so that it generates JUnit test cases use translatable API invocations. Since generated JUnit test cases use only translatable API invocations of a given migration tool, the migration tool can translate these JUnit test cases in Java into NUnit test cases automatically. We find that some JUnit test cases generated by Randoop fail or produce errors. Before translating, TeMAPI removes all such JUnit test cases, so all JUnit test cases run successfully. If a translated NUnit test case fails or produces errors, the test case may indicate different behaviors of API invocation sequences.

4. EVALUATIONS

We implemented a tool for TeMAPI and conducted evaluations using our tool to address the following research questions:

1. How effectively can existing migration tool translate API invocations (Section 4.1)?
2. How effectively can our approach detect different behaviors of single API invocations (Section 4.2)?
3. How effectively can our approach detect different behaviors of multiple API invocations (Section 4.3)?
4. Can the our combination strategy helps achieve better coverage (Section 4.4)?

Table 1 shows the subject tools in our evaluation. Column "Name" lists the names of subject tools. In the rest of the paper, we call "VB & C# to Java converter" as converter for short. Column "Provider" lists their companies. Although all these tools are developed by commercial companies, Java2CSharp, sharpen, and Net2Java are all open source. Column "Description" lists the main functionalities of subject tools. We choose these tools as subjects since we find that many programmer recommend these tools in various forums.

All evaluations were conducted on a PC with Intel Qual CPU @ 2.83GHz and 1.98M memory running Windows XP.

Name	Version	Provider	Description
Java2CSharp	1.3.4	IBM (ILOG)	Java to C#
JLCA	3.0	Microsoft	Java to C#
sharpen	1.4.6	db4o	Java to C#
Net2Java	1.0	NetBean	C# to Java
VB & C# to Java converter	1.6	Tangible	C# to Java

Table 1: Subject tools

Type	No	Java2CSharp		JLCA		sharpen	
		M	%	M	%	M	%
sfg	16962	237	1.4%	3744	22.1%	47	0.3%
sfs	0	0	n/a	0	n/a	0	n/a
nfg	832	0	0.0%	121	14.5%	26	2.2%
nfs	823	0	0.0%	79	9.6%	0	0.0%
sm	1175	97	8.3%	198	16.9%	26	2.2%
nm	175400	3589	2.0%	39536	22.5%	1112	0.6%
Total	195192	3923	2.0%	43678	22.4%	1185	0.6%

Table 2: Translation results of Java to C# tools

4.1 Translating Generated Client Code

For Java to C# tools, we use TeMAPI to generate client-code methods for J2SE 6.0[8]. As described in Section 3.1, TeMAPI ignores all generic API methods. Table 2 shows the translation results. Column "Type" lists the types of generated methods. In particular, "sfg" denotes getters of static fields; "sfs" denotes setters of static fields; "nfg" denotes getters of non-static fields; "nfs" denotes setters of non-static fields; "sm" denotes static methods; "nm" denotes non-static methods; and "Total" denotes the sums of all methods. Column "Number" lists numbers of corresponding types of methods. Columns "Java2CSharp", "JLCA", and "sharpen" list the translation results of corresponding migration tools. For these columns, sub-column "M" lists the number of translated methods without compilation errors, and sub-column "%" lists the percentage from translated method without compilation errors to corresponding generated methods.

From the results of Table 2, we find that it is quite challenging for a migration tool to cover all API invocations. One challenge lies in that API invocations are quite large in size. Although JLCA can translate 43678 generated methods, it covers only 22.4% of total generated methods. The other challenge lies in that many API invocations of different languages cannot be accurately mapped. For example, as pointed out by our previous work [24], one API method in one language may be mapped to several API methods in another language. We find that all the three migration tools have techniques to deal with many-to-many mapping relations. In particular, Java2CSharp and sharpen develop their own assemblies and map some API invocations to their implemented assemblies instead of standard C# API invocations. For example, Java2CSharp maps the `java.lang.Class.forName(String)` method in Java to the `ILOG.J2CsMapping.Reflect.Helper.GetNativeType(String)` method in C#, and the latter method is provided by Java2CSharp. JLCA does not implement additional assemblies, but generate additional source code to hide different behaviors. Although all the three migration tools take the different behaviors of mapped API invocations seriously, they do not cover all differences of API mapping relations. For example, when JLCA translates generated code, it generates a report with 1265 warning messages for different behaviors of API mapping relations. One warning message is "Method 'java.lang.String.indexOf' was converted to 'System.String.IndexOf' which may throw an exception". Still, JLCA leaves these differences to programmers, and the report does not tell programmers when such an exceptions is thrown.

For C# to Java migration tools, we use TeMAPI to generate client-code methods for the .Net framework client profile[9]. As described in Section 3.1, besides generic methods, TeMAPI also ignores methods whose parameters are marked with `out` or `ref`. Table 3 shows the translation results. Columns of Table 3 are of the same meanings with the columns of Table 2. TeMAPI generates almost the same size of methods as it generates for J2SE 6.0. From the results of Table 2, we find that both the two tool translate only quite a small portion of API invocations. As described in the wikipedia[10], C# provides many features that Java does not have (*e.g.*, partial class, reference parameters, output parameters, and named arguments). We suspect that a C# to Java migration tool needs take more effects on these issues, so many mapping relations of API invocations are not addressed yet.

[8]`http://java.sun.com/javase/6/docs/api/`

Type	No	Net2Java		converter	
		M	%	M	%
sfg	3223	1	0.0%	3	0.1%
sfs	8	0	0.0%	0	0.0%
nfg	117	0	0.0%	0	0.0%
nfs	115	0	0.0%	0	0.0%
sm	996	22	2.2%	387	38.9%
nm	190376	4	0.0%	6	0.0%
Total	194835	27	0.0%	396	0.2%

Table 3: Translation results of C# to Java tools

Comparing the translation results between Java-to-C# tools and C#-to-Java tools, we find that Java-to-C# tools cover much more API invocations. To fully explore the translation results of Java-to-C# tools, we present the results of the package level in Table 4. Column "Name" lists the names of Java packages. To save space, we omit the prefixes such as "java.", "javax.", and "org." if it does not introduce ambiguity. We also use short names for some packages. In particular, we use "acc." to denote the `javax.accessibility` package, "man." to denote the `javax.management` package, "java. sec." to denote the `java.security` package, and "javax.sec." to denote the `javax.security` package. We also omits 12 packages that are not covered by all the three tools (*e.g.*, the `javax.rmi` package). Other columns of Table 4 are of the same meanings with the columns of Table 2. From the results of Table 4, we find that all the three migration tools cover the `java.io` package, the `java.lang` package, the `java.util` package, and the `java.net` package. The four packages may be quite important for most Java programs. Almost for all packages, JLCA covers more API invocations. In particular, JLCA covers GUI-related packages such as the `java.awt` package and the `javax.swing` package. As a result, JLCA can translate some Java programs with GUI interfaces whereas the other two tools cannot.

4.2 Testing Single Invocations

To test different behaviors of single invocations, TeMAPI leverages Pex to search internal paths for C# client-code methods. These methods include the translated C# methods without compilation errors as shown in Table 2 and the generated C# methods that can be translated into Java without compilation errors as shown in Table 3. When Pex searches those paths, TeMAPI records the inputs and output of each iteration. Based on these inputs and outputs, TeMAPI generates JUnit test cases to ensure that the generated methods and the translated methods produce the same outputs given the same inputs. As it requires human interactions to test GUI related API invocations, we filter out GUI related API invocations under the `awt` package and the `swing` package although JLCA is able to translate some GUI related API invocations. In addition,

[9]`http://tinyurl.com/252t2ax`
[10]`http://tinyurl.com/yj4v2m2`

Name	No	Java2CSharp		JLCA		sharpen	
		M	%	M	%	M	%
awt	29199	0	0.0%	8637	29.6%	0	0.0%
bean	1768	20	1.1%	14	0.8%	0	0.0%
io	3109	592	19.0%	1642	52.8%	43	1.4%
lang	5221	1494	28.6%	2377	45.5%	791	15.2%
math	1584	101	6.4%	232	14.6%	0	0.0%
java.net	1990	52	2.6%	482	24.2%	10	0.5%
nio	536	30	5.6%	0	0.0%	0	0.0%
java.rmi	1252	0	0.0%	707	56.5%	0	0.0%
java.sec.	2797	50	1.8%	702	25.1%	0	0.0%
java.sql	3495	20	0.6%	183	5.2%	0	0.0%
text	1068	96	9.0%	321	30.1%	0	0.0%
util	9586	1372	14.3%	1879	19.6%	341	3.6%
acc.	237	1	0.4%	25	10.5%	0	0.0%
activation	538	0	0.0%	165	30.7%	0	0.0%
crypto	625	0	0.0%	263	42.1%	0	0.0%
man.	5380	2	0.0%	0	0.0%	0	0.0%
naming	3565	0	0.0%	1365	38.3%	0	0.0%
javax.sec.	1435	0	0.0%	619	43.1%	0	0.0%
sound	515	0	0.0%	56	10.9%	0	0.0%
swing	102389	10	0.0%	21364	20.9%	0	0.0%
javax.xml	4188	34	0.8%	580	13.8%	0	0.0%
org.omg	8937	0	0.0%	1578	17.7%	0	0.0%
w3c.dom	83	0	0.0%	14	16.9%	0	0.0%
org.xml	897	49	5.5%	473	52.7%	0	0.0%

Table 4: Translation results of package level

when Pex searches methods without return values, we ignore those paths that do not throw any exceptions since we cannot generate JUnit test cases for them. We discuss this issue in Section 5.

We run the generated JUnit test cases, and Table 5 shows the results. Column "Name" lists the name of migration tools. Column "JUnit" lists numbers of generated JUnit test cases. Columns "Error" and "Failure" list number of test cases that end with errors and failures, respectively. Sub-column "M" lists numbers of test cases. Sub-column "%" lists percentages from the numbers of corresponding test cases to the numbers of total generated test cases. From the results of Table 5, we find that totally only about half the generated JUnit test cases get passed. It turns out that TeMAPI is quite effective to detect different behaviors of mapped API invocations since it searches every paths of methods under test. The results also reflect that the API mapping relations defined in JLCA and sharpen are more reliable than other tools since more test cases of the two tools get passed than other four tools.

For tools such as Java2CSharp, JLCA, and sharpen, we further present their testing results of the package level in Table 6. Column "Name" lists names of J2SE packages. For columns "Java2CSharp", "JLCA", and "sharpen", sub-column "R" lists numbers of generated JUnit test cases, and sub-column "%" lists percentages from the test cases end with errors or failures to the total test cases. From the results of Table 6, we find that for the `java.sql` package and the `java.util` package, all the tools suffer relatively high error/failure percentages, and for the `java.lang` package and the `java.math` package, all the tools achieve relatively low error/failure percentages. The results may reflect that some packages between Java and C# are more similar than others, so that they can more easily mapped. We also find that for package the `java.text` package, the `javax.xml` package, and the `org.xml` package, JLCA achieve lower error/failure percentages than other tools. The results indicate that a migration tool can achieve better translation results if they carefully prepare the mapping relations of API invocations.

Table 5 and Table 6 show that many generated JUnit tests do not get passed. To better understand different behaviors of mapped API invocations, we manually inspected 3759 JUnit test cases that end with errors or failures. In particular, for tools such as Java2CSharp, JLCA, and sharpen, we investigate the test cases for the `java.lang` package since TeMAPI generates many test cases for the package as shown in Table 4. For tools such as Net2Java and converter, we inspect all their test cases since TeMAPI test cases in small sizes for them. Our findings are as follows:

Name	JUnit	Error		Failure	
		M	%	M	%
Java2CSharp	15458	5248	34.0%	3261	21.1%
JLCA	33034	8901	26.9%	6944	21.0%
sharpen	2730	662	24.2%	451	16.5%
net2java	352	40	11.4%	261	74.1%
converter	762	302	39.6%	182	23.9%
Total	52336	15153	29.0%	11099	21.2%

Table 5: Results of testing single invocations

Finding 1: 36.8% test cases show the different behaviors caused by null inputs.

We find that Java API invocations and their mapped C# API invocations can have different behaviors when inputs are null values. In some cases, a Java API method can accept null values, but its mapped C# API method will throw exceptions given a null value. One such example is shown in Section 1. In other cases, a Java API method will throw exceptions given a null value, but its mapped C# API method can accept null values. For example, JLCA maps the `java.lang.Integer.parseInt(String,int)` method in Java to the `System.Convert.ToInt32(string,int)` in C#. TeMAPI detects that when the inputs of the C# method are null and 10, its output is 0. Given the same inputs, the Java method throws a `NumberFormatException`.

Implication 1: Although programmers may come to agreements on functionalities of API invocations, the behaviors for null values are typically controversial. Programmers or migration tool should deal with null values carefully across Java and C#.

Finding 2: 22.3% test cases show the different behaviors caused by stored string values.

We find that string values between Java API invocations and their mapped C# API invocations are typically different. For example, each Java class has a `toString()` method inherited from the `java.lang.Object` class, and each C# class also has a `ToString()` method inherited from the `System.Object` class. Many migration tools map the two API methods, but the return values of the two methods are quite different in many cases. For another example, many API classes declare methods like `getName` or `getMessage`. These methods also return string values that can be quite different. In particular, we find that the `Message` fields of exceptions in C# often return informative messages. One such message is "Index was outside the bounds of the array" provided by the `System.IndexOutOfRangeException.Message` field. On the other hand, exceptions in Java often provide only null messages.

Implication 2: String values including names are typically different between Java API invocations and their mapped C# API invocations. Programmers should not rely on these string values unless migration tools can hide the differences.

Finding 3: 11.5% test cases show the different behaviors caused by illegal inputs or inputs out of ranges.

We find that Java methods often do not check whether their inputs are legal or out of range, whereas C# methods typically do. For example, Java2CSharp maps the `java.lang.Boolean.parseBoolean(String)` method in Java to the `System.Boolean.Parse(String)` method in C#. Given a string whose value is "test", the Java method return false without checking its format, whereas the C# method throws `FormatException` since it considers "test" as illegal inputs. For another example, the `java.lang.Double.`

Name	Java2CSharp		JLCA		sharpen	
	R	%	R	%	R	%
bean	17	82.4%	18	33.3%	0	n/a
io	4155	67.8%	6981	58.0%	33	1.4%
lang	3480	37.5%	4431	26.1%	1753	29.3%
math	561	4.3%	1629	1.5%	0	n/a
java.net	438	25.1%	3941	47.8%	9	44.4%
nio	27	48.1%	0	n/a	0	n/a
java.rmi	0	n/a	884	32.6%	0	n/a
java.sec.	45	55.6%	828	35.6%	0	n/a
java.sql	260	88.1%	1465	91.0%	0	n/a
text	566	61.5%	374	18.2%	0	n/a
util	5519	60.8%	6177	70.2%	935	62.4%
acc.	1	0.0%	0	n/a	0	n/a
activation	0	n/a	694	53.9%	0	n/a
crypto	0	n/a	298	24.2%	0	n/a
man.	2	0.0%	0	n/a	0	n/a
naming	0	n/a	1569	40.6%	0	n/a
javax.sec.	0	n/a	683	45.9%	0	n/a
sound	0	n/a	66	36.4%	0	n/a
javax.xml	110	71.8%	628	45.9%	0	n/a
org.omg	0	n/a	1842	45.9%	0	n/a
w3c.dom	0	n/a	18	33.3%	0	n/a
org.xml	277	70.0%	483	27.3%	0	n/a

Table 6: Testing results of package level

`shortValue()` method in Java accepts values that are larger than `Short.MAX_VALUE` (32767). JLCA maps the Java method to the `Convert.ToInt16(double)` method in C#, but the C# method throw `OverflowException` when values are larger than 32767.

Implication 3: Programmers should be aware of the different input ranges of API methods between Java and C#. As C# API methods typically check ranges of input, C# programmers may not check ranges of inputs themselves, and thus introduce potential defects in translated Java code.

Finding 4: 10.7% test cases show the different behaviors caused by different understanding or implementation.

We find that Java developers and C# developers may have different understanding or implementation for mapped API methods. For example, according to their documents, the `java.lang.StringBuffer.capacity()` method in Java returns "the current capacity of the String buffer", and the `System.Text.StringBuilder.Capacity` field in C# can return "the maximum number of characters that can be contained in the memory allocated by the current instance". JLCA maps the method in Java to the field in C#, and we find that in many cases they are of different values. For sample, given a string whose value is "0", the `capacity()` in Java returns 0, but the `Capacity` field in C# is 16. We notice that some such differences may indicate defects in mapped methods. For example, Java2CSharp maps the `java.lang.Integer.toHexString(int)` method in Java to the `ILOG.J2CsMapping.Util.IlNumber.ToString(int,16)` method in C#. Given a integer whose value is -2147483648, the Java method returns "80000000", but the C# method returns "\080000000". As the result of the Java method seems to be right, we suspect that the mapped C# method may have some defects to deal with the value.

Implication 4: Although programmers can come to agreement on functionalities of many API methods, they may have different understanding on functionalities of specific methods. Such differences may indicate defects in mapped API methods.

Finding 5: 7.9% test cases show the different behaviors caused by exception handling.

We find that two mapped API methods can throw exceptions that are not mapped. For example, when indexes are out of bounds, the `java.lang.StringBuffer.insert(int,char)` method in Java throws `ArrayIndexOutofBoundsException`. Java2CSharp maps the methods to the `StringBuilder.Insert(int,char)` method in C# that throws `ArgumentOutOfRangeException` when indexes are out of bounds. As Java2CSharp maps `ArrayIndexOutofBoundsException` in Java to `IndexOutOfRangeException` in C#, the mapped C# method may fail to catch exceptions when indexes are out of bounds.

Name	API	JUnit	NUnit	Failure	
				M	%
Java2CSharp	1996	15385	2971	2151	72.4%
JLCA	7060	16630	1067	295	27.6%
sharpen	586	13532	936	456	48.7%
Total	9642	45547	4504	2813	62.5%

Table 7: Results of testing multiple invocations

Implication 5: Even if two methods are of the same functionality, they may produce exceptions that are not mapped. Programmers should be careful to deal with exception handling, unless migrations tools can hide the differences.

Finding 6: 2.8% test cases show the different behaviors caused by static values.

We find that mapped static fields may have different values. For example, the `java.lang.reflect.Modifier` class has many static fields to represent modifiers (*e.g.*, FINAL, PRIVATE and PROTECTED). Java2CSharp maps these fields to the fields of the `ILOG.J2CsMapping.Reflect` class. Although most mapped fields of the two class are of the same values, we find that fields such as VOLATILE and TRANSIENT are of different values. In addition, we find that different values sometimes reveal different ranges of data types. For example, `java.lang.Double.MAX_VALUE` in Java is 1.7976931348623157E+308, and `System.Double.MaxValue` in C# is 1.79769313486232E+308. Although the difference is not quite large, it can cause serious defects if a program needs highly accurate calculation results.

Implication 6: Programmers should be aware of that static fields may have different values even if they has the same names. As these differences reveal that Java and C# may define different bounds for data types, programmers should also be aware of these different bounds if they need highly accurate results of extremely large or small calculation results.

The rest 7.9% test cases fail since API methods can produce random values. For example, the `java.util.Random.nextDouble()` method in Java generates random double values. For another example, each Java class has a `hashCode()` method inherited from the `java.lang.Object` class, and each C# class has a `GetHashCode()` inherited from the `System.Object` class. Both the two methods return a hash code for the current object, so migration tools such as JLCA map the two methods. For these methods, TeMAPI can find their different behaviors of inputs. For example, converter maps the `System.Random. Next(int)` method in C# to the `java.util.Random.nextInt(int)` method in Java. Given a integer whose value is 0, the C# method return 0, but the Java method throws `IllegalArgumentException` with a message: "n must be positive". However, as these methods generate outputs randomly, we cannot conclude they have different behaviors even if corresponding JUnit test cases all fail, and we discuss this issue in Section 5.

4.3 Testing Multiple Invocations

To test different behaviors of API invocation sequences, TeMAPI leverage Randoop to generate test cases that involve multiple API invocations. For each Java-to-C# tools, TeMAPI first analyzes the translation results as shown in Table 2 for the list of translatable API invocations in Java. When generating test cases, TeMAPI extends Randoop, so that each generated test case use only translatable API invocations. Randomly generated invocation sequences may not reflect API usages in true practice. We discuss this issue in Section 5. We find that Randoop can generate failure test cases or even test cases with compilation errors. TeMAPI removes those test cases, so that the remaining test cases all get passed. After that, we use the corresponding migration tool to translate the remaining test cases from Java to C#. As the remaining JUnit test cases all get passed, translated NUnit test cases should also get passed.

Table 7 shows the results. Column "API" lists sizes of translatable API invocations. Column "JUnit" lists numbers of JUnit test cases that run successfully. Column "NUnit" lists numbers of NUnit test cases. We notice that many JUnit test cases are not successfully translated into NUnit test cases. We find that two factors that are not general or not related with API migration: (1) some generated JUnit test cases use classes defined by Randoop; (2) some code structures are complicated to translate, and we further discuss this issue in Section 5. Besides the two factors, we find one general factors for API translation.

Finding 7: Many translated test cases have compilation errors since Java classes and their mapped C# classes have different inheritance relations.

We find that Java API classes can have quite different inheritance relations with their mapped C# API classes, and thus introduce compilation errors in cast statements. For example, a JUnit test case is as follows:

```
public void test87() throws Throwable{
  ...
  StringBufferInputStream var4=...;
  InputStreamReader var10=
    new InputStreamReader((InputStream)var4, var8);
}
```

JLCA translates the JUnit test case into a NUnit test case as follows:

```
public void test87() throws Throwable{
  ...
  StringReader var4=...;
  StreamReader var10=
    new StreamReader((Stream)var4, var8);
}
```

As the two Java classes are subclass and parent class, the Java test case runs successfully, whereas the two C# class have no such relations so translated C# method has compilation errors.

Implication 7: Programmers should use cast statements carefully since classes of two languages typically have different inheritance relations.

Column "Failure" lists failed NUnit test cases. We do not list numbers of test cases with errors since NUnit does not separate errors from failures as JUnit does. Sub-column "M" lists numbers of test cases, and sub-column "%" lists percentages from failed test cases to total test cases.

From the results of Table 7, we find that JLCA achieves better results than other tools since its percentage is the lowest. For each tool, we further investigate the first 100 failed test cases, and we find that 93.3% failed test cases are accumulated by the found factors as shown in Section 4.2: 45.0% for ranges of parameters, 34.0% for string values, 5.3% for different understanding, 4.0% for exception handling, 3.0% for null values, 2.0% for values of static fields, and 0.3% for random values. We find that random strategy affect the distribution. For example, as invocation sequences are random, inputs of many methods are out of range or illegal. Java API methods typically do not check inputs, so generated JUnit run

Class	Pex	Randoop	TeMAPI
ManagerNotRecognizedException	100%	100%	100%
ManagerNotSupportedException	100%	100%	100%
SaxAttributesSupport	78%	74%	80%
XmlSaxDefaultHandler	100%	94%	100%
XmlSAXDocumentManager	29%	17%	29%
XmlSaxLocatorImpl	83%	100%	100%
XmlSaxParserAdapter	100%	100%	100%
XmlSourceSupport	100%	56 %	100%

Table 8: Results of testing coverage

successfully, but translated NUnit test cases fail with various exceptions since C# API methods typically check inputs. Besides those found factors, we find additional two factors as follows:

Finding 8: 3.3% test cases fail because of invocation sequences.

We find that random invocation sequences can violate specifications of API libraries. One type of such specification is described in our previous work [26]: closed resources should not be manipulated. Java sometimes allow programmers to violate such specifications although the return values can be meaningless. One such example is shown in Section 2. Besides invocation sequences that are related to specifications, we find that field accessibility also leads to failures of test cases. For example, a generated JUnit test case is as follows:

```
public void test423() throws Throwable{
  ...
  DateFormatSymbols var0=new DateFormatSymbols();
  String[] var16=new String[]...;
  var0.setShortMonths(var16);
}
```

JLCA translates the JUnit test case into a NUnit test case as follows:

```
public void test423() throws Throwable{
  ...
  DateTimeFormatInfo var0 =
  System.Globalization.DateTimeFormatInfo.CurrentInfo;
  String[] var16=new String[]...;
  var0.AbbreviatedMonthNames = var16;
}
```

The `var0.AbbreviatedMonthNames = var16` statement fails with `InvalidOperationException` since a constant value is assigned to `var0`.

Implication 8: When translating, programmers should check carefully whether they violate speculations of libraries and whether invocation sequences affect accessibility of fields.

The rest 3.0% test cases fail since mapped methods are not implemented. In particular, Java2CSharp maps API invocations in Java to C# API invocations that are not implemented yet. For example, Java2CSharp maps the `java.io.ObjectOutputStream` class in Java to the `ILOG.J2CsMapping.IO.IlObjectOutput-Stream` class in C# that is not implemented yet, and such mapping relations lead to `NotImplementException`. As this difference introduces no compilation errors, programmers should test translated projects carefully to ensure each API method is called.

4.4 Coverage

Test coverage is a common criterion to measure the adequacy of test cases [27]. To investigate whether our combination strategy helps achieve better coverage from Java to C#, we conduct an experiment on JLCA, and Table 8 shows the results. Is trick to extract coverage of API methods, especially in our context where most mapping relations are between J2SE and .NetFramework. For example, we find that coverage tools such as PartCover[11] rely on

[11] `http://partcover.blogspot.com/`

the `JITCompilationStarted` method[12] for notifications of called methods, and thus fail to extract coverage for many methods in .NetFramework since usually no notifications are received when these method are called. As a result, we choose the `org.xml` package in Java as the subject. JLCA generates eight classes as shown in Table 8, and translates some classes of the `org.xml` package in Java to the eight C# classes. Column "Class" shows the names of the eight classes. Column "Pex" lists achieved coverage if leveraging only Pex. Column "Randoop" lists achieved coverage if leveraging only Randoop. We limit the search scope of Randoop to the `org.xml` package, and translate generated Java test cases into C# using JLCA As Pex searches feasible paths and Randoop relies on random strategy, Pex achieves better coverage than Randoop except the `XmlSaxLocatorImpl` class. We find that Pex can fail to generate non-null values for some interfaces. For example, the parameter of the `XmlSaxLocatorImpl(XmlSaxLocator)` constructor is an interface. Pex generates only null inputs for the constructor, but Randoop casts a value to the interface. As a result, Randoop achieves better coverage on this class than Pex. Still, both Pex and Randoop do not achieve high coverage for some classes such as the `XmlSAXDocumentManager` class. We find that some methods of the class cannot be covered unless it reads a file. As both Pex and Randoop generates filenames randomly, these methods are not covered by either tool. Column "TeMAPI" lists achieved coverage if combining Pex and Randoop. We find that the combination achieves the best results for all classes.

To investigate whether our combination strategy helps achieve better coverage from C# to Java, we conduct an experiment on converter, and Table 9 shows the results. For similar consideration, we select several classes (*i.e.*, the `System.Collections.ArrayList` class, the `System.Collections.Hashtable` class, the `System.Collections.Queue` class, and the `System.Collections.Stack` class). Table 9 shows coverage of their translated classes in Java. The four Java classes are decompiled by JAD[13]. To compile the four Java classes, we fix defects introduced during decompiling, and change their package names. Column "Class" shows the names of the four Java classes. Column "Pex" lists achieved coverage if leveraging only Pex. We use TeMAPI to generates Java test cases when Pex searches feasible paths. Column "Randoop" lists achieved coverage if leveraging only Randoop. From the perspective of the four Java classes, Pex also generates test cases randomly since it does not search their feasible paths. As a result, the achieved coverage shown in Table 9 are also half to half. Column "TeMAPI" lists achieved coverage if combining Pex and Randoop. We also find that the combination achieves the best results.

In summary, by combing Pex and Randoop, TeMAPI achieve higher coverage than with single tools.

4.5 Summary

In summary, we find that API invocations are quite large in size, and migration tools typically cover only a small set of API invocations. Although existing migration tools already notice different behaviors of mapped API invocations, many differences are left unsolved. In particular, TeMAPI detects that various factors such as null values, string values, ranges of inputs, different understanding, exception handling, and static values can lead to different behaviors for single API invocations. The preceding factors can accumulate to different behaviors of multiple API invocations. Besides, TeMAPI detects that other factors such as cast statements and invocation sequences can also lead to different behaviors of multiple API invocations.

[12]`http://tinyurl.com/2gy2nqk`
[13]`http://www.varaneckas.com/jad`

Class	Pex	Randoop	TeMAPI
HashTable	23%	15%	26%
LinkedList	32%	26%	37%
ArrayList	18%	25%	31%
Stack	21%	55%	55%

Table 9: Results of testing coverage

4.6 Threats to Validity

The threats to external validity include the representativeness of the subject tools. Although we applied our approach on 5 widely used migration tools, our approach is evaluated only on these limited tools. Other tools may perform better in API migration. This threat could be reduced by more subject tools in future work. The threats to internal validity include human factors for determining factors of different behaviors. To reduce these threats, we inspected detected differences carefully. The threat could be further reduced by introducing more researchers to inspect detected differences.

5. DISCUSSION AND FUTURE WORK

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

Testing API methods with no return values or random return values. If an API method returns no values and throws no exceptions, TeMAPI cannot generate any test cases for them to detect differences of their API mapping relations since it relies on comparing return values for detecting differences. We find that the FAQ of JUnit[14] also discusses a related issue. In this page, Dave Astels suggests to test side effects or to introduce mock objects for methods without return values. We plan to follow his advice in future work. In addition, we find that some API methods return random values, so we cannot conclude their mapped API methods have different behaviors even if corresponding JUnit test cases fail. To test such API methods, we plan to introduce other test oracles in future work. For example, we can generate many test cases and check the distribution of generate random values.

Extracting invocation sequences from existing compatibility kit. Although randomly generated invocation sequences reveal some different behaviors, these random invocation sequences may not reflect API usages in true practice. We notice that Java provides the Compatibility Kit (JCK)[15] to ensure the compatibility of Java provided by different vendors. Besides other Java specifications, JCK defines many typical invocation sequences of API methods with their inputs and expected outputs. In future work, we plan to extract invocation sequences from such compatibility kit, and detect different behaviors based on extracted invocation sequences.

Testing code structures for migration tools. We find that migration tools fail to translate some JUnit test cases since code structures are complicated. Developers of migration tools already notice some such code structures. For example, Java2CSharp lists some unsupported code structures in its website[16]. It may be desirable if some approaches can detect these complicated code structures, so that migration tools can improve their translation capabilities. Daniel *et al.* [3] propose an approach that tests refactory engines by comparing their refactored results given the same generated abstract syntax trees. The idea inspires our future work to testing code structures for migration tools by comparing the translation results given the same generated code snippets.

[14]`http://tinyurl.com/2ccsl5q`
[15]`http://jck.dev.java.net`
[16]`http://tinyurl.com/27z7qrj`

6. 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 [15], researchers proposed various approaches [7, 13, 19, 20, 23] to automate the process. However, all these approaches focus on the syntax or structural differences between languages. Deursen *et al.* [19] 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.* [4]'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 [8] proposed an approach that captures and replays API refactoring actions to update the client code. Xing and Stroulia [22] 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, 16, 17, 25, 26] 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.

7. CONCLUSION

API Mapping relations serve as a basis for automatic translation tools to translate applications from one language to another. However, original and translated applications can exhibit different behaviors due to inconsistencies among mapping relations. In this paper, we proposed an approach, called TeMaAPI, that detects different behaviors of mapped API methods via testing. TeMaAPI targets at generating test cases that covers all feasible paths and sequences to reveal different behaviors of both single methods and method sequences. We implemented a tool and conducted three evaluations on five translation tools to show the effectiveness of our approach. The results show that our approach detects various differences between mapped API methods. We further analyze these differences and their implications. We expect that our results can help improve existing translation tools and help programmers better understand differences of Java and C#.

8. 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] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proc. 6th ESEC/FSE*, pages 185–194, 2007.

[4] 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.

[5] R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In *Proc. 6th ESEC/FSE*, pages 549–552, 2007.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.

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

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

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

[10] S. Koushik, M. Darko, and A. Gul. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[11] T. Kunal and T. Xie. Diffgen: Automated regression unit-test generation. In *Proc. 23rd ASE*, pages 407–410, 2008.

[12] P. Maes. Concepts and experiments in computational reflection. In *Proc. OOPSLA*, pages 147–155, 1987.

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

[14] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. 29th ICSE*, pages 75–84, 2007.

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

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

[17] 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.

[18] N. Tillmann and J. De Halleux. Pex: white box test generation for. NET. In *Proc. 2nd TAP*, pages 134–153, 2008.

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

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

[21] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. 39th DSN*, pages 359–368, 2009.

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

[23] 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.

[24] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proc. 32nd ICSE*, pages 195–204, 2010.

[25] 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.

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

[27] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):427, 1997.