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 elements (*i.e.*, classes, methods, and fields) of L_1 and their mapped API elements of L_2 (as described by API mapping relations) exhibit the same behavior, since any inconsistencies among these mapping relations could result in behavior differences and accumulate to defects in translated applications. Therefore, to detect behavioral differences between API elements 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, 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 API mapping relations. We summarize detected differences as eight findings and their implications. Our approach enables these findings that can improve effectiveness of translation tools, and also assist programmers in understanding the differences among API elements between different languages.

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 [10] shows that nearly one third applications have multiple editions in different languages. A natural way to implement a application in a different language is to translate from existing applications. For example, Lucene.Net was translated from Java Lucene based on its website[2]. As another example, the NeoDatis object database was also translated form Java to C# based on its website[3]. During translation, one primary goal is to provide applications that both exhibit the same behavior.

```
01: private long readLong(ByteArrayInputStream is){
02:  ...
03:  l += ((long) (is.read())) << i;
04: ...}
```

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

```
05: private long ReadLong(ByteArrayInputStream @is){
06:  ...
07:  l += ((long)(@is.Read())) << i;
08:  ...}
```

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 [26], 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 Java 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 elements may occur in many places. To relief efforts, programmers of db4o developed their own translation tool, called sharpen[5], for translating db4o from java to C#. For API translation, sharpen systematically replaces all API elements in Java with equivalent elements in C# to ensure that translated C# applications have the same behaviors with original ones.

In practice, programmers may fail to fix some behavior differences of mapped API elements. These behavior differences are harmful since they may accumulate and cause serious defects in translated applications silently. It is desirable to detect these behavior differences, but existing approaches [12, 14] cannot detect such differences effectively with four major reasons. (1) Existing approaches require both the versions under consideration belong to

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

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

ICSE '11, May 21-28 2011, Waikiki, Honolulu, Hawaii

```
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.

the same language, but in our context, both versions belong to different languages, making these existing approaches inapplicable. (2) The interfaces of two mapped API elements can be different, and one API element can be mapped to multiple API elements. For example, as shown in Figure 3, JLCA[6] translates the `java.net.DatagramSocket.receive(DatagramPacket)` method in Java into multiple C# elements as shown in Figure 4. (3) Detecting behavior differences from existing applications address the problem partially since applications typically use only a small portion of API elements.

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 three major technical challenges in effectively detecting behavioral differences. (1) It is challenging to extract API mapping relations from translation tools for testing since they may follow different styles to describe such relations. For example, Java2CSharp[7] uses mapping files; sharpen hard codes in source files; and commercial tools such as JLCA typically hide mapping relations in binary files. (2) Using a basic 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 [17] and dynamic-symbolic-execution-based [6, 11, 20]. (3) 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 behavior differences of its all API mapping relations automatically. It is important to detect such differences, since they can introduce defects in translated code silently.
- A tool implemented for TeMAPI and three evaluations on 5 translation tools. The results show the effectiveness of our approach to detect behavior differences of both single API classes and multiple classes.
- A first report on behavior differences of API elements between J2SE and .Net. TeMAPI enables us to present such a report. The report shows that various factors such as null inputs, string values, input ranges, different understanding, exception handling, static values, cast statements, and invocation sequences can lead to behavior differences of mapped API elements. Based on the results, we further analyze their implications from the perspectives of API libraries, translation tools, and programmers.

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 elements. In particular, we use the `java.io.BufferedInputStream` class and the JLCA translation tool as illustrative examples.

Translating synthesized wrapper methods. As described by its documentation[8], the `BufferedInputStream` class in Java has 5 fields, 2 constructors, and 8 methods. To fully explore the behaviors of the class, TeMAPI synthesizes a wrapper method for each field and each method given each constructor. For example, given the `BufferedInputStream(InputStream)` constructor, TeMAPI synthesizes 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. We next uses JLCA to translate synthesized wrapper methods 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 in Java 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, as each translated wrapper method exposes its inputs and output, TeMAPI extends Pex [20] to generate test cases for each

[6] http://tinyurl.com/35o5mo7
[7] http://j2cstranslator.sourceforge.net/
[8] http://tinyurl.com/2bca7vh

wrapper method to test behaviors of wrapped field or method. Each input generated by Pex exercises a unique feasible path in the wrapper method. For example, TeMAPI generates the following Java 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 elements defined by JLCA.

Besides Pex, TeMAPI also extends Randoop [17] to generate invocation sequences. By comparing synthesized wrapper methods with translated wrapper methods without compilation errors, TeMAPI extracts the list of translatable API methods. When generating invocation sequences, TeMAPI limits the search scope, so that each generate test case invoke only translatable API methods. For example, a generated Java test case is as follows:

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

The test case gets passed since Java allows programmers to access the stream even if the stream is closed. If a translation tool does not change its behavior, the translated C# test case should also get passed. We next uses JLCA to translate the generated Java test case from Java to C#. As the Java test uses only translatable API elements, JLCA translates it into a C# test case 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 C# test case get failed since C# does not allow such accesses and it throws `ObjectDisposedException`. TeMAPI thus detects a behavior difference caused by invocation sequences.

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.

3. APPROACH

Given a translation tool between Java and C#, TeMAPI generates various test cases to reveal behavior differences 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 class but also mapping relations of multiple API classes.

3.1 Synthesizing Wrappers for Translation

Given a translation tool, TeMAPI first extracts its API mapping relations. To deal with different styles of translation tools as described in Section 1, TeMAPI does not extract API mapping relations directly from translation tools, but chooses to analyze translated code of translation tools. Translating existing applications

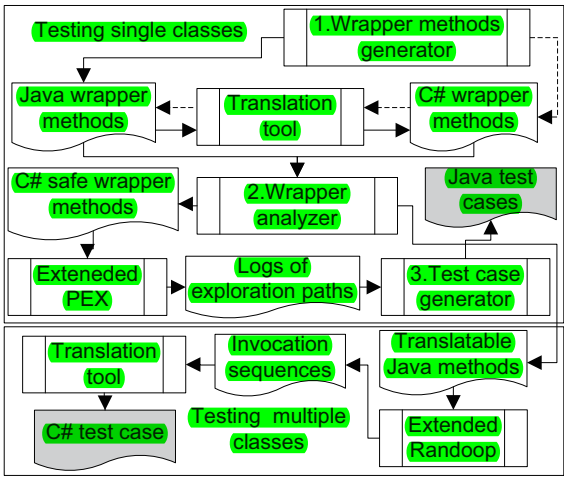

Figure 5: Overview of TeMAPI

address the problem partially for two major reasons: (1) many API mapping relations cannot be covered, since applications typically use a small portion of API elements; (2) in existing applications, a single method often uses multiple API elements. If an API element is translated into multiple elements, it is difficult to extract mapping relations for testing. For the preceding reasons, TeMAPI relies on the reflection technique [13] to synthesize wrapper methods for translation. Each wrapper method uses only one field or method, so TeMAPI is able to extract translatable API elements in the finest level. In particular, TeMAPI synthesizes wrapper methods for static fields and methods as follows:

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

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

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

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

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

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

When TeMAPI synthesizes wrapper methods for non-static fields or methods, it takes constructor into considerations:

Non-static fields. Given a public non-static field `f` of a class `C` whose type is `T`, TeMAPI synthesizes 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 synthesizes a setter as follows:

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

Here, "`|f.name|`" denotes the name of `f`, and "`|no|`" denotes the corresponding number of synthesized wrapper method.

Non-static methods. Given a public non-static method `m(T1 p1,...,Tn pn)` of a class `C` whose return type is `Tm`, TeMAPI synthesizes a wrapper 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); }
```

Here, "|m.name|" denotes the name of m(T1 p1,...,Tn pn).

TeMAPI put all synthesized wrapper methods for one API class C into one synthesized class. For a Java-to-C# tools, TeMAPI synthesizes wrapper methods in Java as shown by the solid line in Figure 5, and for a C#-to-Java tools, TeMAPI synthesizes wrapper methods in C# as shown by the dotted line in Figure 5. When synthesizing, TeMAPI ignores generic methods for simplicity. Besides, for C# code, it ignores unsafe methods, delegate methods, and methods whose parameters are marked as out or ref. Java does not have corresponding keywords, so existing tools typically do not translate them. After wrappers are synthesized, we translate them using a translation tool under analysis.

After synthesized code is translated, TeMAPI parses translated code and removes translated wrapper methods with compilation errors. For Java code, TeMAPI extends Eclipse's Java compiler for the list of wrapper methods with compilation errors. For C# code, TeMAPI extracts the list of wrapper methods with compilation errors through the automation and extensibility interfaces of Visual Studio. For Java-to-C# tools, we refer to the remaining C# wrapper methods as safe wrappers. For C#-to-Java tools, we refer to the synthesized C# wrapper methods that can be translated without compilation errors as safe wrappers. As wrappers reveal behaviors of a given class at the finest level and expose interfaces for inputs and outputs, TeMAPI further uses safe wrappers to test behavior differences for single API classes (Section 3.2). Comparing safe wrappers with synthesized wrappers, TeMAPI can extracts the list of translatable API elements for a given translation tool. Using the list, TeMAPI further tests behavior difference for multiple API classes (Section 3.3).

3.2 Testing Single API Classes

Pex [20] 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 exploration strategies. For example, Xie *et al.* [23] 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 for detecting behavior differences of single classes. In particular, for each API class, TeMAPI leverages Pex to explore paths of each its safe wrapper, and records the inputs and the corresponding output when exploring. Based on recorded inputs and outputs, TeMAPI generates Java test cases to ensure each wrapper methods produce the same output give the same inputs. To check equivalence of outputs, TeMAPI checks whether their values are equal for primitive types and arrays, and checks whether each mapped fields are equal for objects. For example, TeMAPI records that given an empty object, the testappend175nm wrapper method in C# return a StringBuilder object whose Capacity field is 16 and Length field is 13, so TeMAPI generates a test case for the corresponding Java wrapper method as follows:

```
public void testappend175nm122(){
  sketch.Test_java_lang_StringBuffer obj =
      new sketch.Test_java_lang_StringBuffer();
  Object m0 = new Object();
  StringBuffer out = obj.testappend175nm(m0);
  Assert.assertEquals(16, out.capacity());
  Assert.assertEquals(13, out.length());
}
```

This test case fails, since here the capacity() method returns 34 and the length() method returns 24. Thus, TeMAPI detects two behavior differences between the java.lang.StringBuffer class in Java and the System.Text.StringBuilder class in C#.

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

We notice that when Pex explores a path with some specific inputs, the method under exploration throws exceptions. For example, during exploring the TestvalueOf61sm wrapper method, TeMAPI records that given a null input, the method throws NullReferenceException, so it generates a test case to ensure the corresponding Java wrapper method also throws a mapped exception. To generate the test case, TeMAPI first finds the corresponding exceptions in Java by analyzing translated wrapper methods with synthesized ones. In this 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 Java 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 test case fails since the testvalueOf61sm method does not throw any exceptions given a null input. From this failed Java test case, TeMAPI detects the behavior difference between the java.lang.String.valueOf(Object) method in Java and the System.Object.ToString() method in C#, since the preceding two wrapper methods are for the two API methods only.

3.3 Testing Multiple Classes

As each wrapper method uses only one field or one method of a given API class, it loses some behavior differences that involve multiple classes. To detect such differences, TeMAPI first extracts translatable API methods by comparing safe wrappers with synthesized ones. For example, as shown in Section 2, the testskip24nm method in C# is a safe wrapper, so all API methods wrapped in the corresponding Java wrapper are translatable. TeMAPI then parses the method and adds its wrapped BufferedInputStream(InputStream) constructor and the BufferedInputStream.skip(long) method into the list of translatable API methods.

Randoop [17] is a test generation tool for Java. It randomly generates invocation sequences based on already generated test cases in a feedback-directed manner. TeMAPI extends Randoop to test behavior differences that involve multiple classes. In particular, TeMAPI restricts the search scope of Randoop, so that its generated test cases use translatable API elements only. After that, TeMAPI runs generated test cases and removes those failed test cases. We translate the remaining test case into C# using the translation tool under analysis. If such translation does not change behaviors, translated C# test cases should also get passed. Thus, TeMAPI is able to detect behavior differences that involve multiple classes, and Section 2 shows a detected one.

4. EVALUATIONS

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

Type	Num	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

Type	Num	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

1. How many API elements are effectively translated by existing translation tools (Section 4.1)?
2. How many behavior differences of single API classes are effectively detected by our approach (Section 4.2)?
3. How many behavior differences of multiple API classes are effectively detected by our approach (Section 4.3)?
4. How much higher statement coverage is achieved by our combination strategy compared to normal strategy (Section 4.4)?

Table 1 shows the subject tools in our evaluation. Column "Name" lists names of subject tools. In rest of the paper, we refer to "VB & C# to Java converter" as *converter* for short. Column "Version" lists versions of subject tools. Column "Provider" lists companies of subject tools. Although all these tools are from commercial companies, Java2CSharp, sharpen, and Net2Java are open source tools. Column "Description" lists main functionalities of subject tools. We choose these tools as subjects, since they are popular and many programmers recommend these tools in various forums.

All evaluations were conducted on a PC with Intel Qual CPU @ 2.83GHz and 2G memory running Windows XP.

4.1 Translating Synthesized Wrappers

This evaluation focuses on the effectiveness of our approach to extract API mapping relations from both open source tools and closed source tools. The results are useful to follow-up steps, and also reveal to what extents can existing translation tools support API translation. For Java-to-C# tools, TeMAPI first synthesized wrapper methods for all classes of J2SE 6.0[9]. During synthesizing, TeMAPI ignored all generic API methods as described in Section 3.1, and Table 2 shows the translation results. Column "Type" lists types of synthesized methods: "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 sum of all methods. Column "Num" lists numbers of corresponding types of methods. Columns "Java2CSharp", "JLCA", and "sharpen" list the translation results of corresponding translation tools, respectively. For these columns, sub-columns "M" and "%" list the number and percentage of translated wrapper methods without compilation errors, respectively.

Our results show that it is quite challenging for a translation tool to cover all API elements, since API elements are quite large in size. Although JLCA can translate 43,678 wrapper methods, these methods cover only 22.4% of total number of wrapper methods. Furthermore, even if an API element is translated, it can be translated to an API element with a different behavior. We observe that programmers of translation tools may already aware of these behavior differences. For example, after JLCA translated synthesized code, it generated a report with many warning messages regarding behavior differences of translated API elements. For example, a warning message is "Method `java.lang.String.indexOf` was converted to `System.String.IndexOf`, which may throw an exception", but the report does not tell programmers when such an exception is thrown or how to deal with that exception. The test cases generated by TeMAPI can help programmers in understanding when such exceptions are thrown.

For C#-to-Java translation tools, TeMAPI first synthesized wrapper methods for all classes of the .NET framework client profile[10]. As described in Section 3.1, besides generic methods, TeMAPI also ignored `unsafe` methods, `delegate` methods, and methods whose parameters are marked with `out` or `ref`. Table 3 shows the translation results. TeMAPI synthesized almost the same size of wrapper methods as synthesized for J2SE 6.0. Table 3 shows that both tools translate only a small number of API elements. One primary reason could be that C# provides many features such as partial class, reference parameters, output parameters, and named arguments, that are not provided by Java[11]. We suspect that a C#-to-Java translation tool needs take these issues into consideration, since many mapping relations of API elements are not addressed yet.

Tables 2 and 3 show that Java-to-C# tools cover much more API elements compared to C#-to-Java tools. To give more insights, we next present more details at the package level regarding the translation results of Java-to-C# tools in Table 4. Column "Name" lists names of Java packages. To save space, we omit the prefixes such as "java.", "javax.", and "org.". We also use short names "acc.", "man.", "java. sec.", and "javax.sec." to represent `javax.accessibility`, `javax.management`, `java.security`, and `javax.security` packages, respectively. Besides, we omit 12 packages that are not covered by all the three tools (*e.g.*, the `javax.rmi` package). Table 4 shows that all the three translation tools cover four packages: `java.io`, `java.lang`, `java.util`, and `java.net`. These four packages seem to be quite important for most Java programs. Almost for all these packages, JLCA covers more API elements than the other two tools. JLCA even 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 Classes

To detect behavior differences of single classes, TeMAPI leverages Pex to explore safe wrapper methods. These methods include both the translated C# wrapper methods without compilation errors (as shown in Table 2) and the synthesized C# wrapper methods that can be translated into Java without compilation errors (as shown in Table 3). During exploration, when Pex generates inputs that exercise a feasible path in the wrapper method, TeMAPI records the inputs and resulting outputs of that path. Based on these inputs and outputs, TeMAPI generates Java test cases to ensure that synthesized wrapper methods and translated wrapper methods produce the same outputs given the same inputs. Since testing GUI related API elements requires human interactions, we filter out GUI related API elements (*i.e.*, the `awt` package and the `swing` package). In addition, when Pex explores methods without return values, we ignore paths that do not throw any exceptions, since we cannot gen-

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

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

[11] `http://tinyurl.com/yj4v2m2`

Name	Num	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 at package level

erate Java related test cases. We discuss this issue in Section 5.

Table 5 shows the results of executing generated Java test cases. Column "Name" lists names of translation tools. Column "Num" lists the number of generated Java test cases. Columns "E-Tests" and "A-Tests" list the number of exception-causing and assertion-failing test cases. These test cases reflect defects related to behavior differences. Among these two columns, sub-columns "M" and "%" list the number and percentages of these test cases. Table 5 shows that only about half of the generated Java test cases are passed. Among the five tools, sharpen includes the lowest number of "E-Tests" and "A-Tests". It seems that programmers of sharpen put great efforts to fix behavior differences. The percentage of JLCA is also relatively low. The results are comparable, since JLCA translates much more API elements than the other tools. In total, about 50% of test cases are failed. These results show the effectiveness of TeMAPI, since these test cases represent behavior differences.

For Java2CSharp, JLCA, and sharpen, we further present their testing results at 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 Java test cases, and sub-column "%" lists percentages of failing test cases (including exception-causing and assertion-failing). Table 6 shows that for the `java.sql` and `java.util` packages, all tools suffer from relatively high percentages of failing test cases, and for the `java.lang` and `java.math` packages, all tools include relatively low percentage of failing test cases. These results may reflect that some packages between Java and C# are more similar than the others, so they can be more easily mapped. We also find that for the `java.text`, `javax.xml`, and `org.xml` packages, JLCA includes the lowest percentage of failing test cases among the five tools. The results indicate that a translation tool can achieve better translation results if the programmers carefully prepare mapping relations of API elements.

Tables 5 and 6 show that a high percentage of generated Java test cases are failed. To better understand behavior differences of mapped API elements, we inspected 3759 failing Java test cases. For tools Net2Java and converter, we inspect all failing test cases, whereas for Java2CSharp, JLCA, and sharpen, we inspect test cases

Name	Num	E-Tests		A-Tests	
		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 classes

generated for the `java.lang` package, due to a large number of failing test cases. We next present our findings ranked based on the number of failing test cases.

Finding 1: 36.8% test cases show the behavior differences caused by `null` inputs.

We find that Java API methods and their translated C# API methods can have behavior differences when `null` values are passed as inputs. In some cases, a Java API method can accept `null` values, but its translated C# API method throws exceptions. One such example is shown in Section 2 (*i.e.*, the `skip(long)` method in Java and its mapped C# methods). In some other cases, a Java API method throws exceptions with a `null` value, but its translated C# API method can accept `null` values. For example, JLCA translates the `java.lang.Integer.parseInt(String,int)` method in Java to the `System.Convert.ToInt32(string,int)` in C#. If the inputs of the Java method are `null` and 10, it throws `NumberFormatException`, but given the same inputs, the output of the C# method is 0. We notice that some translation tools can fix some differences caused by `null` inputs. For example, to fix the behavior difference of `null` inputs for the `valueOf(Object)` method as shown in Section 3.2, sharpen translates the method to its self-developed C# method, and thus fix the difference.

Implication 1: Although implementers of API libraries in different languages can come to agreements on functionalities of many API methods, behaviors for `null` inputs are typically controversial. Some translation tools such as sharpen try to fix those differences, however, many such differences are still left to programmers as shown in our results. Therefore, programmers should be careful when inputs are `null`.

Finding 2: 22.3% test cases show the behavior differences caused by stored string values.

We find that string values stored in fields between Java classes and their translated C# classes are typically different. This difference ranks as the second, since each Java class has a `toString()` method and each C# class also has a `ToString()` method. Many translation tools map the two API methods, but the return values of the two methods are quite different in many cases. Besides, 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. Overall, we find that all the five tools do not fix this difference.

Implication 2: String fields of mapped classes in different languages typically store different values, but existing translation tools do not fix those differences. Programmers should not rely on these values, since they are typically different across languages.

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

We find that API methods in Java seldom check whether their inputs are illegal or out of range, whereas API methods in C# often do. For example, the `java.lang.Boolean.parseBoolean`

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

`(String)` method in Java does not check for illegal inputs, and returns `false` given an illegal input such as "`test`". Java2CSharp translates it into the `System.Boolean.Parse(String)` method in C#. The C# method throws `FormatException` given the same input since it checks for illegal inputs. As another example, the `java.lang.Double.shortValue()` method in Java accepts values that are larger than `Short.MAX_VALUE` (32,767). JLCA maps the Java method to the `Convert.ToInt16(double)` method in C#. The C# method throws `OverflowException` when values are larger than 32767 since it checks whether inputs are too large.

Implication 3: API methods across languages may follow different standards to check their inputs for different considerations. If a tool translates code from a low standard to a high standard (*e.g.*, Java to C#), it can add extra code to satisfy the high standard. If a tool translates code from a high standard to a low standard (*e.g.*, C# to Java), it will be difficult to fix the difference. When programmers migrate from one language to another, they should check whether the new language follow a high standard or not.

Finding 4: 10.7% test cases show the behavior differences caused by different understanding.

We find that Java programmers and C# programmers may have different understanding for mapped API methods. Two such examples are shown in Section 3.2 (*i.e.*, the `capacity()` method and the `length()` method). Besides, we notice that some differences of this type may indicate defects in translation tools. For example, Java2CSharp translates the `java.lang.Integer.toHexString(int)` method in Java to the `ILOG.J2CsMapping.Util.IlNumber.ToString(int,16)` method in C#. Given an integer whose value is -2147483648, the Java method returns "80000000", but the C# method returns "\080000000". As another example, Java2CSharp maps the `Character.isJavaIdentifierPart(char)` method in Java to the `ILOG.J2CsMapping.Util.Character.IsCSharpIdentifierPart(char)` method in C#. Given a input whose value is `\0`, the Java method returns `true`, but the C# method returns `false`. These two behavior differences were confirmed as defects by programmers of Java2CSharp.

Implication 4: Although implementers can come to agreement on functionalities of many API methods, they may have different understanding on functionalities of specific methods. Some differences of this type are confirmed as defects in translation tools. Programmers should test their translated code carefully since this type of differences are difficult to figure out.

Finding 5: 7.9% test cases show the behavior differences 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 translates the method 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 fails to catch exceptions when indexes are out of bounds.

Implication 5: Implementer of API libraries may design quite different exception handling mechanisms. This type of differences are quite challenging to fix for translation tools. Even if two methods are of the same functionality, programmers should notice that they may produce exceptions that are not mapped.

Finding 6: 2.8% test cases show the behavior differences 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 translates 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: Implementers of API libraries may store different values in static fields. Even if two static fields have the same names, programmers should be aware of that they can have different values. The results also reveal that data types between Java and C# can have different boundaries. Programmers should be aware of this if they need highly accurate results.

The rest 7.9% failing test cases are related to the API methods that can return random values or values that depend on time. For example, the `java.util.Random.nextInt()` method returns random values, and the `java.util.Date.getTime()` method returns the number of milliseconds since Jan. 1st, 1970, 00:00:00 GMT. As another example, each Java class has a `hashCode()` method, and each C# class has also a `GetHashCode()` method. Both the methods return a hash code for the current object, so translation tools such as JLCA map the two methods. Since a hash code is randomly generated, the two methods typically return different values. For these methods, TeMAPI can detect their behavior differences of inputs. For example, converter translates the `System.Random.Next(int)` method in C# to the `java.util.Random.nextInt(int)` method in Java. Given an integer value 0, the C# method return 0, but the Java method throws `IllegalArgumentException` with a message: "n must be positive". However, since these methods return values randomly, we cannot conclude that they have behavior differences even if their outputs are different. We discuss this issue further in Section 5.

4.3 Testing Multiple Classes

To test behavior differences involving multiple classes, TeMAPI leverages Randoop to generate test cases, given the list of trans-

Name	T	Java	C#	A-Tests	
				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 classes

latable API methods. In this evaluation, we focus on Java-to-C# tools only, since C#-to-Java tools translate only a few API elements as shown in Table 2. For each Java-to-C# tool, TeMAPI first extracted the list of translatable API methods using the technique as described in Section 3.3. When generating test cases, TeMAPI extends Randoop, so that each generated test case use only translatable API methods. Randomly generated invocation sequences may not reflect API usages in true practice, and we discuss this issue in Section 5. Among generated test cases, TeMAPI translates only passing test cases from Java to C#.

Table 7 shows the results. Column "T" lists sizes of translatable API methods in Java. Column "Java" lists numbers of passing test cases in Java. Column "C#" lists numbers of translated test cases in C#. We notice that many Java test cases are not successfully translated into C# for two factors that are not general or not related with API migration: (1) to prepare inputs of translatable API methods, Randoop introduces API methods that are not translatable; (2) some code structures are complicated to translate, and we further discuss this issue in Section 5. Besides, our finding is as follows:

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

We find that Java API classes can have different inheritance hierarchies with their translated C# classes, and thus introduce compilation errors. For example, many compilation errors are introduced by type cast statements, and such an example is as follows:

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

Since the preceding two Java API classes are related through inheritance, the test case gets passed. JLCA translates the Java test case into a C# test case as follows:

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

Since the two translated C# classes have no inheritance relations, the translated C# test case has compilation errors.

Implication 7: It seems to be too strict to require that implementers of API libraries in different languages follow the same inheritance hierarchy, and it is also quite difficult for translation tools to fix this behavior difference. Programmers should deal with this difference carefully.

Column "A-Tests" lists the number and percentage of failing C# test cases. Table 7 shows that JLCA achieves the best results among the five tools. For each tool, we further investigate the first 100 failing test cases. We find that 93.3% failing test cases are due to the same factors described 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` inputs, 2.0% for values of static fields, and 0.3% for random values. We find that random strategy of generating invocation sequences affects the distribution. For example, as invocation sequences are random, inputs

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

of many methods are out of range or illegal. Java API methods typically do not check for illegal inputs, therefore, these test cases get passed, but translated C# test cases fail since C# API methods typically check for illegal inputs. Besides the preceding finding, we find an additional finding described 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 [27]: 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 (*i.e.*, the `test413` test case). Besides invocation sequences that are related to specifications, we find that field accessibility also leads to failures of test cases. For example, a generated Java test case is as follows:

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

JLCA translates the Java test case into a C# 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: Legal invocation sequences may become illegal after translation. The target language may be more strict to check invocation sequences, and other factors such as field accessibility can also cause behavior differences. In most cases, programmers should deal with the difference themselves.

The rest 3.0% of failing test cases since translation tools such as Java2CSharp translate API elements in Java to C# API elements that are not implemented yet. For example, Java2CSharp maps the `java.io.ObjectOutputStream` class in Java to the `ILOG.J2CsMapping.IO.IlObjectOutputStream` class in C# that is not yet implemented, and such translations lead to `NotImplementException`. The evaluation in Section 4.2 does not detect this difference since the specific exception is not mapped.

4.4 Coverage

Test coverage is a common criterion to measure the adequacy of test cases [28]. To investigate whether our combination strategy helps achieve better structural coverage such as statement coverage from Java to C#, we conduct an evaluation on JLCA. Table 8 shows the results. Column "Class" shows names of the subject C# classes. JLCA generates the eight classes, and translates some classes of the `org.xml` package in Java to the eight C# classes. We choose only the `org.xml` package in Java as the subject since it is tricky to extract coverage for internal classes of J2SE and .Net. Column "Pex"

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

lists achieved coverage if leveraging only Pex (C#). Column "Randoop" lists achieved coverage if leveraging only Randoop (Java). As Pex explores feasible paths in a systematic manner and Randoop uses random strategy, Pex achieves better coverage than Randoop except for 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 (*e.g.*, the `XmlSAXDocumentManager` class), since covering some methods requires file interactions. As both Pex and Randoop generate filenames randomly, these methods are not covered by either tool. Column "TeMAPI" lists the coverage achieved by 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 evaluation on converter, and Table 9 shows the results. For similar consideration, we select four 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[12]. 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 generate Java test cases when Pex explores feasible paths. Column "Randoop" lists achieved coverage if leveraging only Randoop. Column "TeMAPI" lists the coverage achieved by combining Pex and Randoop. We also find that the combination achieves the best results.

In summary, by using a combination strategy, TeMAPI achieves higher coverage than without using the combination strategy.

4.5 Summary

In summary, we find that API elements are quite large in size, and translation tools typically cover only a small set of API elements. Although existing translation tools already notice behavior differences of mapped API elements, many differences are not fixed. To detect behavior differences, our approach combines random testing with dynamic-symbolic-execution-based testing, and achieves higher coverage than using only one technique. Therefore, our approach enables us to find that various factors such as `null` inputs, `string` values, ranges of inputs, different understanding, exception handling, and static values that could lead to behavior differences for single API classes. The preceding factors can accumulate to behavior differences of multiple API elements. Besides, TeMAPI detects that other factors such as type cast statements and invocation sequences can also lead to behavior differences of multiple API classes.

4.6 Threats to Validity

The threats to external validity include the representativeness of the subject tools. Although we applied our approach on five widely used translation tools, our approach is evaluated only on these limited tools. This threat could be reduced by introducing more subject tools in future work. The threats to internal validity include human factors for inspecting behavior differences from failing test cases. To reduce these threats, we inspected those test cases 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.

Detecting more behavior difference. Although our approach detected many behavior differences, it fails to reveal all behaviors since it does not cover all feasible paths. To detect more behavior differences, some directions seem promising. (1) We can test side effects or mock objects to test methods without return values. (2) To test API methods that return random values, we can check the distribution of their returned values. (3) To test methods that need to read files, we can generate test cases based on Java Compatibility Kit[13] where standard call sequences and files are prepared. (4) Other tools such as Cute [11] and JPF [21] may help generate more test case. We plan to explore these directions in future work.

Testing more API elements. Although our approach covered a majority of API elements, some API elements such as abstract classes and interfaces are not covered. To test behaviors of these API elements, we plan to extend our wrappers in future work. For example, for each abstract class, we can synthesize a subclass as wrapper class for it, and then we can test its behaviors.

Testing translation of code structures. As shown in our evaluations, translation tools may fail to translate if code structures are complicated. We notice that other translation tools encounter with similar problems. For example, 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 translation tools by comparing the translation results given the same code structures.

6. RELATED WORK

Our approach is related to previous work on areas as follows:

API translation. To reduce efforts of language translation, researchers proposed various approaches to automate the process (*e.g.*, JSP to ASP [7], Cobol to Hibol [22], Cobol to Java [15], Smalltalk to C [25], and Java to C# [4]). El-Ramly *et al.* [4] point out that API translation is an important part of language translation, and our previous work [26] mines API mapping relations from existing applications to improve the process. Besides language migration, others processes also involve API translation. For example, programmers often need to update applications with the latest version of API libraries, and the new version may contain breaking changes. Henkel and Diwan [8] proposed an approach that captures and replays API refactoring actions to update the client code. Xing and Stroulia [24] proposed an approach that recognizes the changes of APIs by comparing the differences between two versions of libraries. Balaban *et al.* [1] proposed an approach to migrate client code when mapping relations of libraries are available. As another example, programmers may translate applications from using one API library to another. Nita and Notkin [16] propose twinning to automate the process given that API mapping is specified. Our approach detects behavior differences between mapped API elements, and the results help preceding approaches translate applications without introducing new defects.

[12] `http://www.varaneckas.com/jad`

[13] `http://jck.dev.java.net`

Language comparison. To reveal differences across Languages, researchers conducted various empirical comparisons on different languages. Garcia *et al.* [5] present a comparison study on six languages to reveal their differences of supporting generic programming. Prechelt [19] compares runtime performances of seven languages. Hericko *et al.* [9] compare performance and accumulative size of serialization between Java and C#. Cabral and Marques [2] compare exception handling mechanism between Java and .Net programmers. Paul and Evans [18] compare Java and .Net on their security policies. To the best of our knowledge, no previous work systematically compares behavior differences of mapped API elements since API elements are typically quite large in size. Our approach enables us to present such a report, complementing the preceding empirical studies.

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 behavior differences due to inconsistencies among mapping relations. In this paper, we proposed an approach, called TeMAPI, that detects behavior differences of mapped API methods via testing. TeMAPI targets at generating test cases that covers all feasible paths and sequences to reveal behavior differences 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] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proc. 20th OOPSLA*, pages 265–279, 2005.

[2] B. Cabral and P. Marques. Exception handling: A field study in Java and. Net. *Proc. 21st ECOOP*, pages 151–175, 2007.

[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. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proc. 18th OOPSLA*, pages 115–134, 2003.

[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 *Proce. 27th ICSE*, pages 274–283, 2005.

[9] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, and A. Zivkovic. Object serialization analysis and comparison in Java and .NET. *ACM SIGPLAN Notices*, 38(8):44–54, 2003.

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

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

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

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

[14] W. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

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

[16] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *Proc, 32nd ICSE*, pages 205–214, 2010.

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

[18] N. Paul and D. Evans. Comparing Java and. NET security: Lessons learned and missed. *Computers & Security*, 25(5):338–350, 2006.

[19] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

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

[21] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

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