Automated Testing of API Mapping Relations

Hao Zhong[1], Suresh Thummalapenta[2], and Tao Xie[2]

[1] Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing, 100190, China zhonghao@nfs.iscas.ac.cn
[2] 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 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 behavioral differences and defects in translated applications. Therefore, to detect behavioral differences between mapped 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 behavioral differences between mapped API elements from 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 between mapped API elements. We summarize detected differences as eight findings and their implications. Our approach enables us to produce these findings that can improve effectiveness of translation tools, and also assist programmers in understanding the differences between mapped API elements of 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[3]. 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. An empirical study [15] shows that nearly one third applications have multiple versions in different languages. A natural way to implement an application in a different language is to translate from an existing application.

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

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

Fig. 1. A method in the Java version of db4o.

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

Fig. 2. A method in the C# version of db4o.

For example, Lucene.Net was translated from Java Lucene according to its website[4]. As another example, the NeoDatis object database was also translated form Java to C# according to its website[5]. During translation, one primary goal is to ensure that both applications exhibit the same behavior.

Since 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. Robillard [23] points out that it is hard to use API elements, and our previous work [30] shows that the mapping relations between mapped API elements of different languages can also be complicated. In some cases, programmers may fail to find an existing class that has the same behavior in the other language. For example, Figures 1 and 2 show two methods implemented in db4o[6] of its Java version and its C# version, respectively. When translating the Java code shown in Figure 1 to C#, programmers of db4o may fail to find an existing C# class that has the same behaviors with the `ByteArrayInputStream` class in Java, so they implement a C# class with the same name to fix the behavioral difference. Behavioral differences of mapped API elements (*i.e.*, classes, methods, and fields) may occur in many places. To reduce translation effort, programmers of db4o developed their own translation tool, called sharpen[7], 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 the original Java ones.

In practice, as pointed out by Keyvan Nayyeri[8], one of the most common problems is that translated code does not return expected outputs, partially because behavioral differences of mapped API elements are not fully fixed. It is desirable to detect such differences, but existing approaches [14, 21] cannot detect such differences effectively since these existing approaches require that both the versions under consideration belong to the same language, but in our context, the versions belong to different languages, making these existing approaches inapplicable.

[4] `http://lucene.apache.org/lucene.net/`
[5] `http://wiki.neodatis.org/`
[6] `http://www.db4o.com`
[7] `http://developer.db4o.com/Blogs/News/tabid/171/entryid/653/Default.aspx`
[8] `http://dotnet.dzone.com/print/26587`

```
DatagramSocket socket = ...;
DatagramPacket package = ...;
socket.receive(package);
```

Fig. 3. Sample code in Java.

```
UdpClient socket = ...;
IPEndPoint remoteIpEndPoint = ...;
try{
 byte[] data_in = socket.Receive(ref remoteIpEndPoint);
 PacketSupport tempPacket = new PacketSupport(data_in, data_in.Length);
  tempPacket.IPEndPoint = remoteIpEndPoint;
} catch (System.Exception e){...}
PacketSupport package = tempPacket;
```

Fig. 4. Translated C# code by JLCA.

To address the preceding issue, 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 accepts two inputs: a translation tool under analysis and a test-generation tool. Given a translation tool that translates applications from one language L_1 to the other language L_2, TeMAPI generates various test cases to detect behavioral differences among the API mapping relations by effectively leveraging the test-generation tool. TeMAPI next executes translated test cases 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 four major technical challenges in effectively detecting behavioral differences. (1) It is challenging to directly extract API mapping relations from translation tools. The primary reason is that often translation tools either use different formats for specifying API mapping relations or do not explicitly describe these mapping relations. For example, Java2CSharp[9] uses mapping files, sharpen hardcodes relations in source files, and closed source translation tools such as JLCA typically hide mapping relations in binary files. To address this issue and to be independent of the translation tool under analysis, TeMAPI analyzes translated code for extracting those relations. (2) Interfaces of two mapped API elements can be different, and one API element can be mapped to multiple API elements. For example, JLCA[10] translates the `java.net.DatagramSocket.receive(DatagramPacket)` method in Java as shown in Figure 3 to multiple C# elements as shown in Figure 4. To address this issue, TeMAPI uses a technique, called wrapper methods (Section 3.1), that abstracts interface differences among mapping API elements and provides a common interface to effectively apply test-generation tools. (3) 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 [22] and dynamic-symbolic-execution-based [11, 17, 25] ones. (4) Generating test cases on App_1 and applying those test cases on App_2 may not exercise many behaviors

[9] `http://j2cstranslator.sourceforge.net/`
[10] `http://msdn.microsoft.com/en-us/magazine/cc163422.aspx`

of API methods in App_2, thereby not sufficient to detect related behavioral differences. To address this issue, TeMAPI uses a round-trip technique that also generates test cases on App_2 and applies them back on App_1. We describe more details of our approach to address these challenges in subsequent sections.

In this paper, although we present our approach for detecting behavioral differences among API elements defined by mapping relations, our approach is general and can be applied to other software engineering problems where an API needs to be replaced with another API without changing the behavior of the application using that API. For example, our approach can be used to detect behavioral differences during library upgrades [16] or migrating from one library to another library [19].

This paper makes the following major contributions:

- A novel approach, called TeMAPI, that automatically generates test cases to detect behavioral differences among API mapping relations. Given a translation tool, TeMAPI detects behavioral differences of all its API mapping relations automatically. It is important to detect such differences, since they can introduce defects in translated applications silently.
- A tool implemented for TeMAPI and three evaluations on five popular translation tools. The results show the effectiveness of our approach in detecting behavioral differences of mapped API elements between different languages.
- The first empirical comparison on behavioral differences of mapped API elements between the J2SE and .NET frameworks. The comparison reveals 8 behavioral differences between mapped API elements in existing translation tools. Given these findings, venders of translation tools can further improve their tools; programmers who use translation tools can be aware such differences in advance; and developers of API libraries could improve their libraries for translation.

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. Section 5 discusses issues of our approach. Section 6 presents related work. Section 7 concludes.

2 Example

We next explain our TeMAPI approach with an illustrative example. In particular, TeMAPI accepts a translation tool and a test-generation tool as inputs. TeMAPI includes two major steps in detecting behavioral differences among API elements described in mapping relations of the translation tool. We use JLCA (a Java-to-C# translation tool) as an example translation tool, and the `java.io.ByteArrayInputStream` class in Java as an example API element to illustrate these two steps. We also use two existing state-of-the-art test-generation tools Pex [25] and Randoop [22] for generating test cases that detect behavioral differences. In our approach, we use Pex and Randoop, since both these test-generation tools are popular and are shown to be effective in detecting serious defects in industrial code bases. Furthermore, Pex is effective in generating data for primitive types and Randoop is effective in generating method sequences by combining method calls randomly, complementing each other.

Synthesizing and analyzing wrappers. TeMAPI first synthesizes Java wrapper methods for public methods and fields of the example class, and then uses JLCA to translate the wrapper methods to C#. TeMAPI next compares source code of the synthesized wrapper methods with the compilation-error-free translated wrapper methods to extract translatable API elements of the example class. In particular, our example class in Java has five fields, two constructors, and eight methods besides inherited ones[11]. A class can have more than one constructor, and a translation tool may not translate all its constructors. Therefore, to address this issue, TeMAPI includes different constructors in its synthesized wrapper methods instead of simply pushing the receiver object as a parameter of wrapper methods. For example, TeMAPI first identifies `ByteArrayInputStream(byte[])` constructor as translatable, and synthesizes the wrapper method for the `skip(long)` method as follows:

```
public long testskip24nm(long m0, byte c0[]){
  ByteArrayInputStream obj = new ByteArrayInputStream(c0);
  return obj.skip(m0);
}
```

TeMAPI next uses JLCA to translate synthesized wrapper methods from Java to C#. A translation tool typically cannot include mapping relations for all the API elements between two languages, so translated wrapper methods can have compilation errors. TeMAPI parses translated wrapper methods and filters out all methods with compilation errors. For example, below is the translated `testskip24nm` method in C#:

```
public virtual long testskip24nm(long m0, sbyte[] c0){
  MemoryStream obj = new MemoryStream(SupportClass.ToByteArray(c0));
  MemoryStream 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 remove this method, since it does not result in compilation errors.

Generating and Translating test cases. We next explain how TeMAPI leverages Pex to generate test cases using synthesized wrappers. A major advantage of our synthesized wrapper method is that the synthesized wrapper method and the translated wrapper method have the same interface irrespective of method calls within the wrapper method. Therefore, TeMAPI detects behavioral differences between mapped API elements by generating test cases on one language version of wrapper methods and applying those test cases on the other language version. In particular, TeMAPI extends Pex [25] to generate test cases for each remaining C# wrapper method. For the example class, Pex attempts to explore all feasible paths among method calls within the wrapper methods and generates inputs and outputs that exercise various paths. Based on the inputs and output generated for each path, TeMAPI generates a Java test case to check whether the original wrapper method returns the same value as the translated one. For example, TeMAPI generates the following Java test case based on inputs generated by Pex for one feasible path (in the C# wrapper method) that throws exceptions (see Section 3.2 for the details of generating Java test cases).

[11] `http://download.oracle.com/javase/6/docs/api/java/io/ByteArrayInputStream.html`

```
public void testskip24nm36(){
  try{
     Test_java_io_ByteArrayInputStream obj = new Test_java_io_ByteArrayInputStream();
     long m0 = java.lang.Long.valueOf("2147483648").longValue();
     byte[] c0 = new byte[0];
     obj.testskip24nm(m0,c0);
     Assert.assertTrue(false);
  }catch(java.lang.Exception e){
     Assert.assertTrue(true);
  }
}
```

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

We next describe how TeMAPI generates test cases using Randoop [22]. TeMAPI does not generate invocation sequences from wrappers directly, since each wrapper method includes a fixed simple invocation sequence. Instead, TeMAPI uses translatable API methods in Step 1, and limits the scope of Randoop to those methods while generating invocation sequences. 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 access to the stream even if the stream is closed. TeMAPI next uses JLCA to translate the generated Java test case from Java to C#. Since the Java test case uses only translatable API elements, JLCA translates the test case to 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;
  AssertTrue(var5 == 1);
}
```

In contrast to the Java test case, the C# test case gets failed, since C# does not allow such access to the stream and throws `ObjectDisposedException`. TeMAPI thus detects a behavioral difference with invocation sequences. This example motivates our basic idea of generating test cases in one language and translating those test cases to another language for detecting differences among API mapping relations.

3 Approach

We next describe our TeMAPI approach that accepts two tools as inputs: a translation tool under analysis and a test-generation tool. TeMAPI effectively leverages the test-generation tool to generate test cases that detect behavioral differences of API mapping relations defined by the translation tool. TeMAPI addresses two major issues in detecting behavioral differences of API mapping relations. First, it is challenging to extract API mapping relations defined by the translation tool. The primary reason is that often translation tools either use different formats for specifying API mapping relations or do not explicitly describe these mapping relations. In the later case, API mapping relations are hard coded within the source code of translation tools such as sharpen. To address this issue and to be independent of the translation tool under analysis, TeMAPI does not extract API mapping relations directly from translation tools. Instead, TeMAPI analyzes translated code for extracting those relations. In particular, TeMAPI constructs *wrapper* methods for each API element in a language and translates those wrapper methods to another language to extract the API mapping relations. Section 3.1 presents more details on how wrapper methods are constructed from API elements. These wrapper methods also help identify the API elements that can be translated by the translation tool under analysis, since often translation tools translate only a limited set of API elements [30].

Second, exposing behavioral differences among API elements that are defined by API mapping relations poses testability issues. For example, the interface of an API method in one language could be different from the interface of its mapped API method or an API method in one language can be mapped to multiple API methods in the other language, resulting in controllability issues. On the other hand, generating test oracles is also challenging, resulting in observability issues. TeMAPI addresses these issues by using wrapper methods as an abstraction. In particular, these wrapper methods expose a common interface for API elements that are defined by API mapping relations, thereby addressing controllability issues. Furthermore, to address observability issues, TeMAPI uses return values of wrapper methods or exceptions being thrown as test oracles. We next describe more details on how wrapper methods are constructed and later describe how TeMAPI leverages two existing state-of-the-art test-generation tools Pex [25] and Randoop [22] for generating test cases that detect behavioral differences[12].

3.1 Synthesizing and Analyzing Wrappers

Given a translation tool under analysis, TeMAPI first extracts its API mapping relations. To deal with different formats of translation tools as described in Section 1, TeMAPI does not extract API mapping relations directly from translation tools, but analyzes translated code for such relations. TeMAPI includes a wrapper generator that generates wrappers for API elements in both Java and C#. For static fields and static methods, the two wrapper generators of TeMAPI use the following rules to synthesize wrapper methods. In the synthesized code below, "`|f.name|`" denotes the name of a field `f`; "`|m.name|`" denotes the name of a method `m`; and "`|no|`" denotes the id of the synthesized wrapper method.

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|sfg(){ return C.f; }
```

[12] TeMAPI is general and any other test-generation tool such as Java PathFinder [1] can be easily leveraged to generate test cases due to the abstraction provided by the wrapper methods.

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

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

Static methods. Given a public static method `m(T1 m1,...,Tn mn)` of a class `C` whose return type is `Tm`, TeMAPI synthesizes a wrapper method as follows:

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

When TeMAPI synthesizes wrapper methods for non-static fields or methods, TeMAPI takes constructors into considerations.

Non-static fields. Given a public non-static field `f` of a class `C` whose type is `T`, TeMAPI synthesizes a getter using each constructor `C(T1 c1,..., Tn cn)` of `C` as follows:

```
public T testGet|f.name||no|nfg(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|nfs(T v, T1 c1,..., Tn cn){
  C obj = new C(c1,..., cn);
  obj.f = v;
}
```

Non-static methods. Given a public non-static method `m(T1 m1,...,Tn mn)` of a class `C` whose return type is `Tm`, TeMAPI synthesizes a wrapper method using each constructor `C(Tv cv,..., Tt ct)` of `C` as follows:

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

TeMAPI groups all synthesized wrapper methods for one API class C to one synthesized class. When synthesizing, TeMAPI ignores generic methods, since the translation tools cannot handle generics. Furthermore, when synthesizing wrappers for C# API classes, TeMAPI ignores `unsafe` methods, `delegate` methods, and methods whose parameters are marked as `out` or `ref` besides generic methods. Java does not have these corresponding keywords, so existing translation tools typically do not translate the preceding methods. After wrapper methods are synthesized, TeMAPI uses the translation tool under analysis to translate wrapper methods to the other language.

Our synthesized wrappers are quite simple in their structures. Therefore, existing translation tools are typically able to translate their structures correctly. Still, some translated wrappers can have compilation errors, since existing translation tools cannot translate some API elements. We use *safe wrappers* to refer to the wrappers that are translated without compilation errors. To identify safe wrappers, TeMAPI extends Visual Studio and Eclipse's Java compiler for C# and Java code, respectively. Comparing source code of safe wrappers with source code of synthesized wrappers, TeMAPI extracts one-to-one mapping relations of API classes for the tool under analysis. For example, by comparing the first statements of the `testskip24nm` method in Java and in C# as shown in Section 2, TeMAPI extracts the mapping relation between the `ByteArrayInputStream` class in Java and the `MemoryStream` class in C# defined by JLCA. Through analysis, TeMAPI also extracts translatable API methods for the translation tool under analysis. In the preceding example, TeMAPI adds `BufferedInputStream(InputStream)` constructor and the `skip(long)` method in Java to translatable API methods of JLCA, since their belonging wrapper method is translated from Java to C# without compilation errors.

3.2 Generating and Translating Test Cases

Using Pex. We next describe how TeMAPI generates test cases that detect behavioral differences using Pex. Pex uses a technique, called dynamic symbolic execution [11, 17], for systematically exploring the code under test and generates test cases that exercise various paths in the code under test. In particular, TeMAPI applies Pex on safe wrappers in C# language, since Pex handles .NET code.

To generate test cases for Java to C# translation tools, TeMAPI synthesizes wrappers in Java and translates those wrappers into C#. TeMAPI next generates test cases on wrappers in C# using Pex. TeMAPI captures the return values of wrapper methods or exceptions being thrown and uses them as test oracles in generated test cases. TeMAPI next translates generated test cases to Java to detect behavioral differences. On the other for C# to Java translation tools, TeMAPI synthesizes wrappers in C# and generates test cases on those wrappers. TeMAPI next translates both synthesized wrapper methods and generated test cases to Java to detect behavioral differences.

We next describe the technical challenges addressed by TeMAPI while translating test cases generated in C# to Java. When translating test cases to Java, TeMAPI refers to the extracted mapping relations of API elements (Section 3.1) to translate C# values into Java values. In a few cases, values of some primitive types can cause compilation errors. Therefore, TeMAPI replaces them with corresponding method invocations. For example, the `long m0 = 2147483648` statement causes a compilation error with a message: "The literal 2147483648 of type `int` is out of range". Here, TeMAPI uses a method invocation to replace this statement in the `testskip24nm` test case. Pex includes some heuristics for generating data factories for non-primitive types[13]. For these non-primitive types, TeMAPI refers to extracted mapping relations of API elements to translate them. To check equivalence of outputs, TeMAPI checks whether their values are equal for primitive types and arrays, and checks whether each mapped field is equal for objects. For example, TeMAPI records that given an empty object, the `testappend175nm` wrapper method in C# returns a `StringBuilder` object whose `Capacity` field is 16 and `Length` field is 13, so TeMAPI derives a test case for the corresponding Java wrapper method as follows:

```
public void testappend175nm122(){
  Test_java_lang_StringBuffer obj = new Test_java_lang_StringBuffer();
  Object m0 = new Object();
```

[13] In our future work, we plan to further enhance these data factories using our previous work [24].

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

Table 1. Subject tools

```
  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 behavioral differences between the `java.lang.StringBuffer` class in Java and its translated Class in C#.

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 in C#, TeMAPI records that given a `null` input, the method throws `NullReferenceException`, so TeMAPI derives a test case to ensure that the corresponding Java wrapper method also throws a mapped exception. To derive the test case, TeMAPI 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 TeMAPI derives a Java test case as follows:

```
public void testvalueOf61sm3(){
  try{
    Test_java_lang_String obj = new 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 gets failed since the `testvalueOf61sm` method in Java does not throw any exceptions given a null input. From this failing test case, TeMAPI detects the behavioral difference between the `java.lang.String.valueOf(Object)` method in Java and the `System.Object.ToString()` method in C#, since the two wrapper methods invoked by the two preceding test cases are for these two API methods. When the translation tool under analysis is from C# to Java, wrapper methods generated for C# are used for deriving C# test cases for Java code.

Using Randoop. Despite of its benefits to detect behavioral differences, a wrapper method cannot help effectively generate test invocation sequences, since it has fixed invocation sequences (*e.g.*, a constructor first and then the public method). To address this issue, TeMAPI extends Randoop [22] for Java to generate invocation sequences in Java for all translatable API methods of the translation tool under analysis. Randoop

Type	Number	Java2CSharp	JLCA	sharpen
getters of static fields	16,962	237	3,744	47
setters of static fields	0	0	0	0
getters of non-static fields	832	0	121	0
setters of non-static fields	823	0	79	0
static methods	1,175	97	198	26
non-static methods	175,400	3,589	39,536	1,112
Total	195,192	3,923	43,678	1,185

Table 2. Translation results of Java-to-C# tools

randomly generates test cases based on already generated test cases in a feedback-directed manner. In particular, when using Randoop to generate test cases, TeMAPI limits its search scope to translatable API methods. TeMAPI runs generated test cases, and removes all failing test cases. TeMAPI next uses the translation tool under analysis to translate generated Java test cases to the other language C#. If translated code has the same behaviors, translated test cases should also get passed. If not, TeMAPI detects behavioral differences. Randoop-generated Java test cases also include `assert` statements to assert behaviors of Java code. Thus, TeMAPI does not need to generate assertion statements as it does with Pex. Section 4.3 shows such detected behavioral differences that are related to invocation sequences (*e.g.*, the `test423` test cases).

4 Evaluations

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

1. How many API elements can be translated by existing translation tools (Section 4.1)?
2. How many behavioral differences are effectively detected by generating C# test cases for testing Java code (Section 4.2)?
3. How many behavioral differences are effectively detected by by generating Java test cases for testing C# code (Section 4.3)?
4. How many behavioral differences are detected with and without TeMAPI's internal techniques (Section 4.4)?

Table 1 shows subject tools in our evaluations. Column "Name" lists names of these tools. We use *converter* to denote the "VB & C# to Java converter" for short. Java2CSharp, sharpen, and Net2Java are open source tools, and JLCA and converter are closed source tools. Column "Version" lists versions of subject tools. Column "Provider" lists companies of these tools. Column "Description" lists main functionalities of these 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. More details of our evaluation results are available at `https://sites.google.com/site/asergrp/projects/temapi`.

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 for subsequence steps, and also to show the effectiveness of existing translation tools. For Java-to-C# tools, TeMAPI first synthesized wrapper methods for all classes of J2SE 6.0[14]. Most classes in J2SE support both generic programming and non-generic programming (*e.g.*, `java.util.ArrayList`). During synthesizing, our approach focuses on only non-generic programming as described in Section 3.1. Table 2 shows the translation results. Column "Type" lists types of synthesized methods, and Row "Total" denotes the sum of all methods. Column "Number" lists numbers of corresponding types of methods. Columns "Java2CSharp", "JLCA", and "sharpen" list the translation numbers of translated wrapper methods without compilation errors for each tool, respectively. Although these tools support mapping for only some APIs, these supported APIs are the most commonly used, and these tools' significant utility/popularity are reflected by their high download counts.

Our results show that JLCA is able to translate much more API elements than the other two tools. Still, even if an API element is translated, it can be translated to API elements with behavioral differences. We observe that developers of translation tools may already aware of some behavioral differences. For example, after JLCA translated synthesized code, it generated a report with many warning messages regarding behavioral differences of translated API elements. For example, a warning message was "Method `java.lang.String.indexOf` was converted to `System.String.IndexOf`, which may throw an exception", but the report does not describe when such an exception is thrown or how to deal with that exception. TeMAPI complements the problem, and detects that the Java method does not check whether inputs are out of ranges as the C# method does. For example, given an empty string `str`, the `str.indexOf("", -1)` statement in Java returns 0, whereas the `str.IndexOf("", -1)` statement in C# throws `ArgumentOutOfRangeExcpetion`.

For C#-to-Java translation tools, TeMAPI first synthesized wrapper methods for all the classes of .NET framework client profile[15]. As described in Section 3.2, besides generic methods, TeMAPI also ignored `unsafe` methods, `delegate` methods, and methods whose parameters are marked with `out` or `ref`. TeMAPI synthesized almost the same size of wrapper methods as it synthesized for J2SE. Table 3 shows the translation results. We find that both tools translate only a small number of API elements. One primary reason could be that C# provides many features such as partial classes, reference parameters, output parameters, and named arguments, that are not provided by Java[16]. We suspect that a C#-to-Java translation tool needs to deal with these issues, so many mapping relations of API elements are not addressed yet.

Tables 2 and 3 show that the Java-to-C# tools are able to translate much more API elements compared to the 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 12

[14] `http://java.sun.com/javase/6/docs/api/`
[15] `http://msdn.microsoft.com/en-us/library/ff462634.aspx`
[16] `http://en.wikipedia.org/wiki/Comparison_of_Java_and_C_Sharp`

Type	Number	Net2Java	converter
getters of static fields	3,223	1	3
setters of static fields	8	0	0
getters of non-static fields	177	0	0
setters of non-static fields	115	0	0
static methods	996	0	6
non-static methods	190,376	94	387
Total	194,835	99	396

Table 3. Translation results of C#-to-Java tools

packages that are not translated by all the three tools (*e.g.*, the `javax.rmi` package). Table 4 shows that all the three translation tools can translate the `java.io`, `java.lang`, `java.util`, and `java.net` packages. These four packages seem to be quite important for most Java programs. Almost for all these packages, JLCA translates more API elements than the other two tools. JLCA can also handle GUI-related packages such as the `java.awt` package and the `javax.swing` package, and can translate some Java programs with GUI interfaces whereas the other two tools cannot translate such programs.

4.2 Generating and Translating Test Cases with Pex

To detect behavioral differences through wrappers, TeMAPI leverages Pex to explore safe wrappers. 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 to 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 return the same outputs given the same inputs. Since testing GUI related API elements requires human interactions, we filter out these elements (*i.e.*, the `awt` package and the `swing` package). In addition, when Pex explores methods without return values, TeMAPI ignores paths that do not throw any exceptions, since it cannot generate 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 "Number" lists numbers of generated Java test cases. Columns "E-Tests" and "A-Tests" list numbers of exception-causing and assertion-failing test cases. For the 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 behavioral 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 behavioral 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

Name	Java2CSharp	JLCA	sharpen
java.awt	0	8,637	0
java.bean	20	14	0
java.io	592	1,642	43
java.lang	1,494	2,377	791
java.math	101	232	0
java.net	52	482	10
java.nio	30	0	0
java.rmi	0	707	0
java.security	50	702	0
java.sql	20	183	0
java.text	96	321	0
java.util	1,372	1,879	341
javax.accesibility	1	25	0
javax.activation	0	165	0
javax.crypto	0	263	0
javax.management	2	0	0
javax.naming	0	1,365	0
javax.security	0	619	0
javax.sound	0	56	0
javax.swing	10	21,364	0
javax.xml	34	580	0
org.omg	0	1,578	0
w3c.dom	0	14	0
org.xml	49	473	0

Table 4. Java-to-C# translation results of package level

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 percentages of failing test cases. This result may reflect that some packages between Java and C# are more similar than the others, so they can be more easily translated. 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 result indicates that a translation tool can achieve better translation results if its developers carefully prepare API mapping relations.

Tables 5 and 6 show that a high percentage of generated Java test cases are failed. To better understand behavioral differences between mapped API elements, we inspected 3,759 failing Java test cases. For Net2Java and converter, we inspect all failing test cases, whereas for Java2CSharp, JLCA, and sharpen, we inspect test cases generated for the `java.lang` package, due to a large number of failing test cases. Each failing test case reflects one unique behavioral difference, and Figure 5 shows the distribution of found unique behavior differences. We next present our findings ranked by percentages of failing test cases.

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

Name	Number	E-Tests		A-Tests	
		M	%	M	%
Java2CSharp	15,458	5,248	34.0%	3,261	21.1%
JLCA	33,034	8,901	26.9%	6,944	21.0%
sharpen	2,730	662	24.2%	451	16.5%
net2java	352	40	11.4%	261	74.1%
converter	762	302	39.6%	182	23.9%
Total	52,336	15,153	29.0%	11,099	21.2%

Table 5. Results of generating C# test cases for Java code

We find that many Java API methods and their translated C# API methods have behavioral 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 other cases, a Java API method throws exceptions given a `null` input, 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 C# method returns 0. We notice that translation tools can fix some differences caused by `null` inputs. For example, to fix the behavioral difference of `null` inputs for the `valueOf(Object)` method as shown in Section 3.2, sharpen translates the method to its own method, and fixes 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 these 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 behavioral differences caused by stored `string` values.

We find that `string` values stored in fields between Java classes and their mapped 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. In addition, 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 in C#. On the other hand, exceptions in Java often provide only `null` messages. Overall, we find that none of the five tools fixes this difference.

Implication 2: Mapped `String` fields 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 behavioral differences caused by illegal inputs or inputs out of ranges.

Name	Java2CSharp		JLCA		sharpen	
	R	%	R	%	R	%
java.bean	17	82.4%	18	33.3%	0	n/a
java.io	4,155	67.8%	6,981	58.0%	33	39.4%
java.lang	3,480	37.5%	4,431	26.1%	1,753	29.3%
java.math	561	4.3%	1,629	1.5%	0	n/a
java.net	438	25.1%	3,941	47.8%	9	44.4%
java.nio	27	48.1%	0	n/a	0	n/a
java.rmi	0	n/a	884	32.6%	0	n/a
java.security	45	55.6%	828	35.6%	0	n/a
java.sql	260	88.1%	1,465	91.0%	0	n/a
java.text	566	61.5%	374	18.2%	0	n/a
java.util	5,519	60.8%	6,177	70.2%	935	62.4%
javax.accessibility	1	0.0%	25	16.0%	0	n/a
javax.activation	0	n/a	694	53.9%	0	n/a
javax.crypto	0	n/a	298	24.2%	0	n/a
javax.management	2	0.0%	0	n/a	0	n/a
naming	0	n/a	1,569	40.6%	0	n/a
javax.sec.	0	n/a	683	29.4%	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	1,842	36.3%	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. Results of Table 5 at package level (three tools)

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(String)` method in Java does not check for illegal inputs, and returns `false` given an illegal input such as "`test`". Java2CSharp translates it to 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 32,767. JLCA translates the Java method to the `Convert.ToInt16(double)` method in C#. The C# method throws `OverflowException` when values are larger than 32,767 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. When programmers migrate from one language to another, they should check whether the new language follows a high standard or not.

Finding 4: 10.7% test cases show the behavioral differences caused by different understandings.

We find that implementers of API libraries may have different understandings for mapped API methods in different languages. Two such examples are shown in Section 3.2 (*i.e.*, the `capacity()` method and the `length()` method). In some cases, such differences reflect different natures between languages. For example, we find

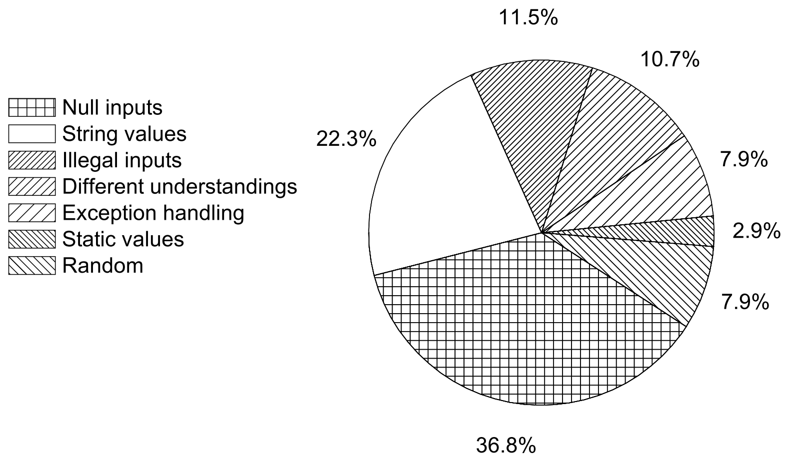

Fig. 5. Distribution of found unique behavioral differences with Pex

that Java considers "\" as existing directories, but C# considers it not. In some other cases, we find that such differences can 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 -2147483648, the Java method returns "80000000", but the C# method returns "\080000000". As another example, Java2CSharp translates the `Character.isJavaIdentifierPart(char)` method in Java to the `ILOG.J2CsMapping.Util.Character.IsCSharpIdentifierPart(char)` method in C#. Given an input "\0", the Java method returns `true`, but the C# method returns `false`. These two behavioral differences were confirmed as defects by developers of Java2CSharp after we reported the detected defects.

Implication 4: Implementers can have different understanding on functionalities of specific methods. Some such differences reflect different natures of different languages, and some other differences indicate defects in translation tools. Programmers should test their translated code carefully since this type of differences is difficult to figure out.

Finding 5: 7.9% test cases show the behavioral differences caused by exception handling.

We find that two mapped API methods can throw exceptions that are not mapped. For example, the `java.lang.StringBuffer.insert(int,char)` method in Java throws `ArrayIndexOutofBoundsException` when indexes are out of bounds, and Java2CSharp translates the method to the `System.Text.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: Implementers of API libraries may design quite different exception handling mechanisms. This type of differences is quite challenging to fix for translation tools. Even if two methods are of the same functionality, programmers should notice that these methods may produce exceptions that are not mapped.

Name	Method	Java	C#	A-Tests	
				M	%
Java2CSharp	1,996	15,385	2,971	2,151	72.4%
JLCA	7,060	16,630	1,067	295	27.6%
sharpen	586	13,532	936	456	48.7%
Total	9,642	45,547	4,974	2,902	58.3%

Table 7. Results of generating Java test cases for C# code

Finding 6: 2.9% test cases show the behavioral differences caused by static values.

We find that mapped static fields may have different values. For example, the `java.lang.reflect.Modifier` class in Java 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 in C#. Although most values of the mapped fields are the same, 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, the `java.lang.Double.MAX_VALUE` field in Java is 1.7976931348623157E+308, but the `System.Double.MaxValue` field 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 these fields can have different values. The results also reveal that data types between Java and C# can have different bounds. Programmers should be aware of this situation if they need highly accurate results.

The remaining 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 behavioral differences of their 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 returns 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 behavioral differences even if their outputs are different. We discuss this issue further in Section 5.

4.3 Generating and Translating Test Cases with Randoop

To test behavioral differences involving invocation sequences, TeMAPI leverages Randoop to generate test cases, given the list of translatable API methods. In this evaluation, we focus on the Java-to-C# tools only, since the 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 3.2. When generating test cases, TeMAPI extends Randoop, so that each generated test case uses 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, we translate only passing test cases from Java to C#.

Table 7 shows the results. Column "Method" lists sizes of translatable API methods for the three tools. Column "Java" lists the numbers of passing test cases in Java. Column "C#" lists the numbers of translated test cases in C#. We notice that many Java test cases are not successfully translated to 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) the numbers of compilation errors can be magnified since Randoop produces many redundancies (Section 4.4 shows an example of produced redundancies). Our findings are 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 to 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 behavioral difference. Programmers should deal with this difference carefully.

Column "A-Tests" of Table 7 lists numbers and percentages of failing C# test cases. We find that JLCA achieves the lowest percentages among the five tools. For each tool, we further investigated its first 100 failing test cases. Figure 6 shows the distribution of found behavioral differences with Randoop. We find that 93.6% 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. When generating test cases, Pex explores feasible paths, whereas Randoop uses a feed-back random strategy. As a result, the distribution of Figure 5 is different from the distribution of Figure 6. The distribution of Figure 5 is more reasonable since each

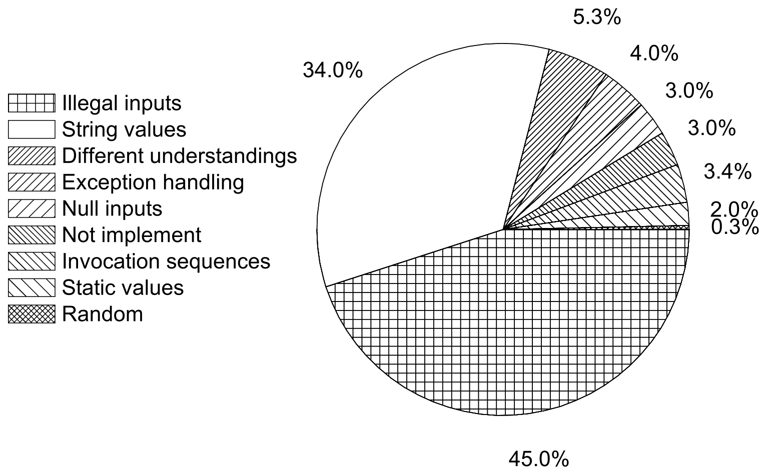

Fig. 6. Distribution of found behavioral differences with Randoop

feasible path reflects a unique behavior, and each failing test case reflects a unique behavioral difference, whereas the distribution of Figure 6 is affected by redundancies in test cases generated by Randoop. Still, Randoop's random strategy helps find an additional behavioral difference as follows:

Finding 8: 3.4% test cases fail because of invocation sequences.

We find that random invocation sequences can violate specifications of API libraries. One type of such specifications is described in our previous work [31]: closed resources should not be manipulated. Java sometimes allows programmers to violate such specifications although 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 to a C# test case as follows:

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

In the preceding translated test case, the last statement throws `InvalidOperationException` since a constant value is assigned to `var0`.

Implication 8: Legal invocation sequences can become illegal after translation. The target language may be more strict to check invocation sequences, and other factors such

Class	Method	Pex	Randoop	Combination	Percent
ParserAdapter	23	8	2	9	39.1%
AttributeListImpl	19	7	3	7	36.8%
AttributesImpl	31	15	11	18	58.1%
XMLReaderAdapter	23	8	2	9	39.1%
LocatorImpl	17	4	0	4	23.5%
DefaultHandler	26	4	0	4	15.4%
HandlerBase	23	4	1	5	21.7%
InputSource	15	4	0	4	26.7%
NamespaceSupport	15	5	2	6	40.0%
SAXException	15	5	1	5	33.3%
SAXParseException	19	6	1	6	31.6%
SAXNotSupportedException	15	5	1	5	33.3%
SAXNotRecognizedException	15	5	1	5	33.3%
Total	256	80	25	87	34.0%

Table 8. Results with and without TeMAPI's internal techniques

as field accessibility can also cause behavioral differences. In most cases, programmers should deal with the difference themselves.

The remaining 3.0% test cases get failed since translation tools such as Java2CSharp translate API elements in Java to C# API elements that are not implemented yet. For example, Java2CSharp translates 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 Significance of TeMAPI's Internal Techniques

To investigate the significance of TeMAPI's internal techniques, we use JLCA as the subject tool, and the `org.xml` package in Java as the subject package for detecting behavioral differences. For each class of the package, we compare the number of distinct translatable methods with behavioral differences when we use TeMAPI with and without its internal techniques, and Table 8 shows the results. Column "Class" shows the names of classes in Java that can be translated to C# by JLCA. Column "Method" lists the numbers of translatable methods of each class. These methods include inherited ones. Columns "Pex", "Randoop", and "Combination" list numbers of found distinct translatable methods with behavioral differences when TeMAPI uses only Pex, only Randoop, and both Pex and Randoop, respectively. With only Pex, 483 test cases were generated, and 132 test cases failed. With only Randoop, 1200 test cases were generated in Java, and all these test cases got passed. After translation, all translated test cases in C# had no compilation errors, and 1168 C# test cases got failed. We inspected these failing test cases, and we found that test cases generated by Pex are more effective to reveal behavioral differences than test cases generated by Randoop, since for test cases, Pex explores feasible paths whereas Randoop generates randomly. Although more test cases were generated by Randoop fail than by Pex, these failing test cases do not reveal any new methods with behavioral differences since these failing test cases are redundant. For example, we found that 1151 test cases generated by Randoop all have the same invocation sub-sequence as follows:

```
SaxAttributesSupport var25 = new SaxAttributesSupport();
System.Int32 var26 = 1;
System.String var27 = var25.GetLocalName((int) var26);
Assert.IsTrue(var27 == null);
```

In this sub-sequence, JLCA translates the `AttributeListImpl.getName(int)` method in Java to the `SaxAttributesSupport.GetLocalName(int)` method in C#. The translation makes the assertion fail since the C# method does not return `null` given an empty attribute as the Java method does. Besides redundancies, each test case generated by Randoop uses many API elements, and each test case generated by Pex focuses on only one field or method within a synthesized wrapper method. As a result, it takes much more efforts to locate a method with behavioral differences from failing test cases generated by Randoop than by Pex.

From the results of Table 8, we find that Pex alone can detect most behavior differences.

However, the combination of the two techniques helps TeMAPI detect more methods with behavioral difference. Besides the behavioral differences that involve invocation sequences, we also find that Pex can fail to explore paths that are too complicated. Randoop complements Pex to generate test cases for detecting behavioral differences of such methods since Randoop generates test cases randomly. Column "%" lists percentages from "T" to "M". We find that behavioral differences of mapped API methods are quite common since about one third methods have such differences.

4.5 Summary

In summary, we find that API elements are quite large in size, and translation tools typically can translate only a small portion of API elements. Although existing translation tools already notice behavioral differences of mapped API elements, many differences are not fixed. To detect behavioral differences, our approach combines random testing with dynamic-symbolic-execution-based testing, and achieves to detect more behavioral differences than with single techniques. Our approach enables us to present the first empirical comparison on behavioral differences of API mapping relations between Java and C#. We find that various factors (*i.e.*, `null` inputs, `string` values, ranges of inputs, different understanding, exception handling, static values, type cast statements, and invocation sequences) could lead to behavioral differences of mapped API elements between different languages.

4.6 Threats to Validity

The threats to external validity include the representativeness of the subject tools. Although we applied our approach on five popular 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 behavioral differences from failing test cases. To reduce these threats, we inspected those test cases carefully. The threat could be further reduced by involving more parties 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 behavioral difference of mapped API elements. Although our approach detected many behavioral differences, it may fail to reveal all behaviors. To detect more behavioral differences, some directions seem to be promising: (1) we can rely on 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) other tools such as CUTE [17] and JPF [26] may help generate more test cases to reveal more behaviors. We plan to explore these directions in future work. In addition, our approach does not cover some types of API elements (*e.g.*, abstract classes and protected elements). To test these elements, we plan to extend our wrappers in future work (*e.g.*, generating a concrete wrapper class for each abstract class). In our project website, we released all synthesized and translated wrappers, so that researchers can also employ other static/dynamic techniques to detect more behavioral differences.

Testing translation of other programming languages. Some programming languages may have much more different code structures than Java and C# do, and existing translation tools between these programming languages may fail to translate some different code structures. Daniel *et al.* [7] propose an approach that tests refactoring engines by comparing their refactored results given the same generated abstract syntax trees. In future work, we plan to adapt their approach to test translation tools by comparing their translation results given the same code structures as inputs. As pointed out by Waters [27], when a source language is fundamentally different from its target language, programmers may even have to abstract a source program and to re-implement its target program from scratch. Still, API differences are important for programmers to avoid related defects when they re-implement a target program. Canfora *et al.* [6] propose an approach that can wrap functionalities of legacy system as services. To detect such differences, we plan to adapt their wrappers to test mapped API elements between two fundamentally different languages in future work.

Improving translation tools and detecting related defects. Our evaluation reveals where existing tools fail to fix behavioral differences between mapped API elements. To improve existing translation tools, we plan to mine better mapping relations and to propose corresponding translation techniques for fixing these differences. In addition, found behavioral differences are potential to introduce defects in translated client code. In future work, we plan to conduct empirical studies to investigate whether found behavior differences really introduced defects in translated code, and to propose corresponding detection techniques if such defects are found.

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 [12], Cobol to Hibol [27], Fickle to Java [2], Cobol to Java [18], Smalltalk to C [29], and Java to C# [9]). El-Ramly *et al.* [9] point out that API translation is an important part of language translation, and our previous work [30] mines API mapping relations from existing applications in different languages to improve the process. Besides language migration, other processes also involve API translation. For example, programmers often need to update applications with the latest version of API libraries, and a new version may contain breaking changes. Henkel and Diwan [13] proposed an approach that captures and replays API refactoring actions to update the client code. Xing and Stroulia [28] proposed an approach that recognizes the changes of APIs by comparing the differences between two versions of libraries. Balaban *et al.* [4] proposed an approach to migrate code when mapping relations of libraries are available. As another example, programmers may translate applications to use alternative APIs. Dig *et al.* [8] propose *CONCURRENCER* that translates sequential API elements to concurrent API elements in Java. Nita and Notkin [20] propose twinning to automate the process given that API mapping is specified. Our approach detects behavioral differences between mapped API elements, and the results help the preceding approaches translate applications with fewer defects.

Language comparison. To reveal differences between languages, researchers conducted various empirical comparisons on various languages. Garcia *et al.* [10] present a comparison study on six languages to reveal their differences of supporting generic programming. Cabral and Marques [5] compare exception handling mechanisms between Java and .NET programs. Appeltauer *et al.* [3] compare eleven context-oriented programming languages (*e.g.*, Lisp) for their designs and performances. To the best of our knowledge, no previous work systematically compares behavioral differences of API elements from different languages. Our approach enables us to produce such a comparison study, complementing the preceding empirical comparisons.

7 Conclusion

Translated applications can exhibit behavioral differences from the original applications due to inconsistencies among API mapping relations. In this paper, we proposed an approach, called TeMAPI, that detects behavioral differences of mapped API elements via testing. For our approach, 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 behavioral differences between mapped API elements. We further analyze these differences and their implications. Our approach enables such findings that can help improve existing translation tools and help programmers better understand differences between different languages such as Java and C#.

References

[1] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *Proc. TACAS*, pages 134–138, 2007.

[2] D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, and E. Zucca. A provenly correct translation of Fickle into Java. *ACM Transactions on Programming Languages and Systems*, 29(2):13, 2007.

[3] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A comparison of context-oriented programming languages. In *Proc. COP co-located with ECOOP 2009*, pages 1–6, 2009.

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

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

[6] G. Canfora, A. R. Fasolino, G. Frattolillo, and P. Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81(4):463–480, 2008.

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

[8] D. Dig, J. Marrero, and M. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *Proc 31st ICSE*, pages 397–407, 2009.

[9] 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.

[10] 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.

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

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

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

[14] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *Proc. 3rd ICST*, pages 137–146, 2010.

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

[16] D. Kawrykow and M. P. Robillard. Improving api usage through automatic detection of redundant code. In *Proc. ASE*, pages 111–122, 2009.

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

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

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

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

[21] A. Orso, M. Harrold, D. Rosenblum, G. Rothermel, M. Soffa, and H. Do. Using component metacontent to support the regression testing of component-based software. In *Proc. ICSM*, pages 716–725, 2001.

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

[23] M. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Software*, pages 27–34, 2009.

[24] 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.

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

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

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

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

[29] 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.

[30] 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.

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