Exposing Behavioral Differences in Cross-Language API Mapping Relations

Hao Zhong[1], Suresh Thummalapenta[2], and Tao Xie[3]

[1] Laboratory for Internet Software Technologies, Institute of Software, CAS, Beijing, China
[2] IBM Research, Bangalore, India
[3] Department of Computer Science, North Carolina State University, Raleigh, USA
zhonghao@itechs.iscas.ac.cn, surthumm@in.ibm.com,
xie@csc.ncsu.edu

Abstract. Due to various considerations, software vendors often translate their applications from one programming language to another, either manually or with the support of translation tools. Both these scenarios require translation of many call sites of API elements (*i.e.*, classes, methods, and fields of API libraries). API mapping relations, either acquired by experienced programmers or already incorporated in translation tools, are much valuable in the translation process, since they describe mapping relations between source API elements and their equivalent target API elements. However, in an API mapping relation, a source API element and its target API elements may have behavioral differences, and such differences could lead to defects in the translated code. So far, to the best of our knowledge, there exists no previous study for exposing or understanding such differences. In this paper, we make the first attempt to expose and analyze behavioral differences in cross-language API mapping relations. From our result, we summarize eight findings and their implications that can improve effectiveness of translation tools, and also assist programmers in understanding the differences between mapped API elements of different languages during the translation process. Some exposed behavioral differences can indicate defects in translation tools, and four such new defects were confirmed by the developers of those tools.

1 Introduction

Since the inception of computer science, many programming languages (*e.g.*, COBOL, C#, and Java) have been introduced. To survive in competing markets and to address various business requirements, software companies or open source organizations often release variants of their applications in different languages. In total, as stated by Jones [12], nearly one third of the existing applications have versions in more than one language. There are three major factors for such phenomenon of having application variants in multiple languages. First, Application Programming Interface (API) libraries or engines are often released in more than one language to attract programmers in diverse backgrounds. Second, stand-alone applications are released in more than one language to acquire specific features of different programming languages. Finally, the market of mobile platforms is highly competitive, and different mobile platforms typically support different programming languages (*e.g.*, Andriod supports Java, iOS supports Objective C, and Windows Mobile supports C#). To survive in the competitive

market, mobile application vendors tend to release variants of applications in different languages for multiple platforms.

State of the art. To reduce development effort, a natural way to implement an application in a different language is to translate from an existing application. During the translation, programmers need to resolve many differences across two languages. Comparing with limited keywords and code structures, two languages typically have many API elements. As reported by El-Ramly *et al.* [7], it is critical to translate API elements correctly during the translation process. To translate API elements, programmers need to replace call sites of source API elements in one programming language systematically with their target API elements in another programming language, based on their known mapping relations. In practice, many programmers rely on their own understandings to translate API elements, but some experienced programmers also incorporate their understandings in translation tools. For example, programmers of db4o[4] developed a translation tool, call Sharpen, for translating db4o from Java to C#. To reduce the effort of translating API elements, they incorporated hundreds of API mapping relations inside Sharpen. Furthermore, to translate more API elements, researchers [27] proposed various approaches that mine API mapping relations automatically.

Given an entity E_1 (such as API classes, methods, and fields) in a language L_1, and another entity E_2 in another other language L_2, a mapping relation m is a triple $\langle E_1, E_2, f \rangle$ and describes that E_1 and E_2 have the same functionality f. API mapping relations are valuable to translate call sites of API elements, but they can introduce defects in the translated code silently. In an API mapping relation, E_1 and E_2 may have behavioral differences, and such differences can lead to defects. For a mapping relation m, a behavioral difference occurs, when translating between E_1 and E_2 leads to different output values or exceptions given the same input values and method sequences. For example, when translating Lucene.NET[5] from its Java version, a programmer, named Chrisopher Currens, expressed his concerns in an email:[6]

"*It could, also, hide bugs, since it's possible, however unlikely, something could port perfectly, but not behave the same way. A class that has many calls to string.Substring is a good example of this. If the name of the function is changed to the .Net version (.substring to .Substring), it would compile no problems, but they are very different.*"

The following code of Lucene.NET explains the behavioral difference.

Java code

```
01: protected String add_escapes(String str) {...
02:  String s = "0000" + Integer.toString(ch, 16);
03:  ret = s.substring(s.length() - 4, s.length());
```

Translated C# code

```
04: protected internal String Add_escapes(String str){...
05:  String s = "0000" + System.Convert.ToString(ch, 16);
06:  ret = s.Substring(s.Length - 4, 4);
```

In this example, the second parameter of the `substring(int,int)` method in Line 03 denotes the end index of a return substring, whereas the second parameter of the `Substring(int,int)` method in Line 06 denotes the number of characters in a return substring. To resolve this difference, programmers of Lucene.NET changed the

[4] http://www.db4o.com

[5] https://cwiki.apache.org/LUCENENET/

[6] http://tinyurl.com/88xnf26

arguments accordingly during the translation. Due to a large number of API mapping relations, where some relations are quite complex [19], it is difficult for programmers to know all such behavioral differences of API mapping relations in advance. If programmers translate applications without realizing such behavioral differences, they can introduce defects in the translated code. If developers of translation tools are not aware of such behavioral differences, they may introduce defects in translation tools, and these defects could lead to further defects in the code translated by these tools.

Existing approaches [2, 3, 19, 21] cannot expose behavioral differences in cross-language API mapping relations effectively. For example, our previous work [19] compared API documents for behavioral differences when the APIs evolve, and cannot analyze behavioral differences for cross-language API mapping relations since their documents are fundamentally different. Srivastava *et al.* [21] compared API implementations for their behavioral differences, and their approach cannot analyze many API libraries whose code is not available. Bartolomei *et al.* [2, 3] list challenges to fix behavioral differences between Swing and SWT in the same programming language, but do not handle source code in different languages. To the best of our knowledge, many questions are still open. For example, are such behavioral differences pervasive? What types of behavioral differences are there? Which types of behavioral differences are more common than others? Are behavioral differences easy to be resolved?

Challenges. To answer the preceding questions, we need many API mapping relations for analysis, but it could take much effort for programmers to write them down manually. Instead, we choose to extract mapping relations that are already incorporated in translation tools. To achieve this goal, we have to overcome the following challenges.

Challenge 1. It is challenging to extract API mapping relations from translation tools, since developers of translation tools either use different formats for specifying API mapping relations, or hardcode API mapping relations in their tools' source code.

To address this challenge, instead of extracting API mapping relations directly from translation tools, we analyze translated results for extracting API mapping relations.

Challenge 2. Collected applications under translation may not cover some interesting API elements. In addition, it is difficult to align source API elements and their target API elements for complex mapping relations in collected applications.

To address the challenge, we synthesize the code under translation as test cases generated for all API elements of an API library. To generate test cases, we leverage two state-of-the-art techniques: random testing [17] and dynamic-symbolic-execution-based testing [9, 14, 24]. We generate test cases with simple code structures and minimum API elements, so that if a translated test case fails, it is easy to locate the behavioral difference of the API mapping relation in the failed test case.

Challenge 3. Translated code typically has compilation errors (*e.g.*, due to the API elements that do not fall into the scope of translatable API elements for the translation tool), so it is not feasible to expose behavioral differences via testing directly.

To address the challenge, we extract translatable API elements for a translation tool, and try to generate test cases that use only translatable API elements. For those generated test cases with compilation errors, we filter them out automatically.

Our contribution. This paper makes the following major contributions:

– A tool chain, called TeMAPI, that detects behavioral differences among API mapping relations. With its support, we conduct the first empirical study on behavioral differences of mapped API elements between the J2SE and the .NET frameworks.
– Empirical results showing that behavioral differences are pervasive. We summarize exposed behavioral differences into eight findings, and discuss their implications that are valuable to vendors of translation tools for improving their tools, programmers who use these translation tools, and developers of API libraries for implementing more translatable APIs.
– Empirical results showing that some behavioral differences indicate defects in translation tools, and four defects were confirmed by the developers of translation tools.

Although we focus on cross-language API mapping relations in this paper, our process 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 an application (*e.g.*, upgrading client code with the latest library [10], migrating to alternative APIs [16], or migrating to efficient APIs [13]).

2 Study Setup

Our process has three steps, and is not symmetry across the two languages under translation, since capabilities of translation tools are not symmetry and existing test-generation tools typically work for a single language rather than both languages under translation.

Step 1: Synthesizing and analyzing wrappers. For a translation tool that translates one language (L_1) to the other language (L_2), TeMAPI generates wrappers for API elements in L_1. 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. TeMAPI synthesizes a getter for a public static field `T f` of a class `C`:

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

If `f` is not a constant, TeMAPI synthesizes a setter wrapper as well for that field:

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

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

```
public Tr test|m.name||no|sm(T1 p1,..., Tn pn){return C.m(p1,..., pn);}
```

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

Non-static fields. Given a public non-static field `T f` of a class `C`, TeMAPI synthesizes a getter using each constructor `C(C1 c1,..., Cn cn)` of `C` as follows:

```
public T testGet|f.name||no|nfg(C1 c1,..., Cn cn){
   C obj = new C(c1,..., cn);
   return obj.f; }
```

If `f` is not a constant, TeMAPI synthesizes a setter wrapper as well for that field.

```
public void testSet|f.name||no|nfs(T p, C1 c1,...,Cn cn){
  C obj = new C(c1,..., cn);
  obj.f = p; }
```

Non-static methods. Given a public non-static method `Tr m(T1 p1,...,Tn pn)` of a class `C`, TeMAPI synthesizes a wrapper method using each constructor `C(C1 c1, ..., Cn cn)` of `C` as follows:

```
public Tr test|m.name||no|nm(T1 p1,...,Tn pn,C1 c1,...,Cn cn){
  C obj = new C(c1,..., cn);
  return obj.m(p1,..., pn); }
```

For example, for JLCA, TeMAPI synthesizes a wrapper method in Java for the `ByteArrayInputStream.skip(long)` method in Java as follows:

```
public long testskip24nm(byte c1[], long p1){
  ByteArrayInputStream obj = new ByteArrayInputStream(c1);
  return obj.skip(p1);}
```

TeMAPI groups all synthesized wrapper methods for one API class C to one synthesized class. After TeMAPI synthesizes wrapper methods, we use the translation tool under analysis to translate wrapper methods to the other language. For example, JLCA translates the preceding `testskip24nm` method into C# as follows:

```
public virtual long testskip24nm(sbyte[] c1, long p1){
  MemoryStream obj = new MemoryStream(SupportClass.ToByteArray(c1));
  MemoryStream temp_BufferedStream = obj;
  Int64 temp_Int64 = temp_BufferedStream.Position;
  temp_Int64=temp_BufferedStream.Seek(p1,System.IO.SeekOrigin.Current)-temp_Int64;
  return temp_Int64;}
```

TeMAPI extends existing compilers to find wrappers that are translated into the other language without compilation errors (referred to as *safe wrappers* in this paper). In particular, TeMAPI extends Visual Studio for C# code and Eclipse's Java compiler for Java code. From our experiences, translation tools are typically able to translate the simple code structures of synthesized wrappers, and all compilation errors are caused by untranslatable API elements in synthesized wrappers. TeMAPI compares safe wrappers with synthesized wrappers to extract the following two facts:

(1) The one-to-one mapping relations of API elements for the translation tool under analysis. For example, by comparing the first statements of the two `testskip24nm` methods in Java and in C#, TeMAPI extracts the mapping relation between the `ByteArrayInputStream` class in Java and the `MemoryStream` class in C# defined by JLCA, since the two methods declare two local variables with the same name, `obj`. Since translation tools typically do not modify names of variables, TeMAPI extracts such relations by using names. In Step 2, TeMAPI uses such relations to generate test cases in Java when leveraging Pex.

(2) The list of translatable API elements for the translation tool under analysis. For example, by comparing the first statements of the two `testskip24nm` methods in Java and in C#, TeMAPI adds the `ByteArrayInputStream(InputStream)` constructor and the `skip(long)` method in Java to translatable API methods of JLCA, since their corresponding wrapper methods are translated without compilation errors. In Step 3, we use the list to limit the search scope of Randoop.

Step 2: Generating test cases with Pex. Pex [24] uses dynamic symbolic execution [9, 14] for generating test cases that exercise various paths in the code under test. Pex requires adding annotations (*e.g.*, `[TestClass()]`) to code under test for test generation. We use Pex to generate test cases for wrapper methods. In particular, for C#-to-Java translation tools, we use Pex to generate test cases in C# for synthesized wrapper methods in C# that are translated to Java without compilation errors, and for Java-to-C# translation tools, we use Pex to generate test cases in C# for translated wrapper methods in C# without compilation errors. When Pex generates test cases in C#, we set its parameters to allow it to exercise paths in API libraries.

TeMAPI uses the extracted mapping relations of API elements to translate generated test cases from C# into Java. Since test cases generated by Pex typically have limited API elements, extracted one-to-one mapping relations are adequate to translate test cases generated by Pex. Here, an alternative way is to use C#-to-Java translation tools to translate generated test cases. We do not choose this way, since we find that existing C#-to-Java translation tools cannot translate many API elements, and these tools do not support user-defined API mapping relations. Java and C# have different bounds for their literals. For example, the `long m0 = 2147483648` statement compiles well in C#, but it causes a compilation error: "The literal 2147483648 of type `int` is out of range". To resolve this difference, TeMAPI considers literals in C# as strings, and gets their values by corresponding API methods in Java.

To expose behavioral differences, TeMAPI uses two mechanisms for generating test oracles. First, TeMAPI inserts assert statements based on values of public fields or return values of public methods. For example, TeMAPI records that given an empty object, the `testappend175nm` wrapper method in C# returns a `StringBuilder` object whose `Capacity` is 16 and `Length` is 13, so TeMAPI derives a test case for the corresponding wrapper method in Java:

```
public void testappend175nm122(){
  Test_java_lang_StringBuffer obj = new 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 the `capacity()` method returns 34 and the `length()` method returns 24, so TeMAPI detects two behavioral differences. Here, TeMAPI ignores non-primitive or non-public fields of return objects, and thus may miss some behavioral differences that are not easy to be observed.

Second, TeMAPI uses expected assertions as test oracles. For example, when Pex explores a path, the method under exploration could throw exceptions. TeMAPI generates the following test case in Java based on inputs generated by Pex for one feasible path (in the C# wrapper method) that throws exceptions.

```
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 test case in Java 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# code by JLCA.

Step 3: Generating test cases with Randoop. Randoop [17] randomly generates test cases based on already generated test cases in a feedback-directed manner. A wrapper method cannot help effectively generate method sequences in generated test cases, since it has fixed method sequences. To detect behavioral differences with method sequences of a translation tool, instead of generating test cases for wrapper methods, we

use Randoop for API elements directly. Randoop generates test cases for arbitrary methods by default. To generate useful test cases for our purpose, we configure Randoop so that it generates method sequences only for the translatable API methods of the translation tool. Here, in Step 1, TeMAPI extracts translatable API methods of the translation tool with the support of synthesized wrappers.

Passing test cases are much useful to detect behavioral differences. If a passing test case fails after it is translated, it is easy to identify a behavioral difference, since the translated code should have the same behavior. If a translated failing test case fails, it is difficult to infer informative results, since the translation may and may not introduce more defects for causing the failure. Our preliminary study shows that Java-to-C# tools can translate many API elements. After TeMAPI removes all failing test cases, we use the translation tool under analysis to translate generated test cases from Java to C#. For example, TeMAPI generates a test case in Java as follows:

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

JLCA translates the generated test case from Java to C# 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);}
```

The preceding test case in Java passes, but the test case in C# fails. We thus detect a behavioral difference with method sequences.

In our tool chain, wrapper methods play an important role. First, some test generation tools need to instrument code under test. For example, Pex needs to add annotations to code under test. For those API libraries whose code is not available, our tool chain allows test generation tools to instrument wrapper methods for test generation. Second, by comparing translated wrappers with original wrappers, we implement a single technique to extract useful facts by comparing translated code, while different techniques are needed to extract such facts from different translation tools directly. Finally, wrapper methods expose a common interface for all the API elements, and thus help expose behavioral differences of API elements.

3 Empirical Results

In this section, we address the following research questions:

- Are behavioral differences pervasive in cross-language API mapping relations?
- What are the characteristics of behavioral differences concerning inputs and outputs?
- What are the characteristics of behavioral differences concerning method sequences?

In our study, we choose the translation tools in Table 1 as our subjects, since they are popular and many programmers recommend these tools in various forums. For Java-to-C# tools, TeMAPI synthesizes wrapper methods for J2SE 6.0[7], and ignores methods

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

Table 1. Subject tools

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
Converter	1.6	Tangible	C#-to-Java

that include generics, since many translation tools cannot handle generics. For C#-to-Java tools, TeMAPI synthesizes wrapper methods for the .NET 4.0 framework clients[8], and ignores `unsafe`, `delegate`, and generic methods, and also the methods whose parameters are marked as `out` or `ref`. Java does not have these corresponding keywords, so existing translation tools typically do not translate the preceding methods. More details of our empirical results are available at `http://sites.google.com/site/asergrp/projects/temapi`.

Pervasiveness of behavioral differences. Table 2 shows the overall result. For Pex, column "Name" lists the names of translation tools, and column "Number" lists the number of generated test cases in Java. These number largely reflect how many API elements can be translated by corresponding tools. Columns "E-Tests" and "A-Tests" list the number of exception-causing and assertion-failing test cases, respectively. For the two columns, sub-columns "M" and "%" list the number and percentage of these test cases, respectively. For Randoop, column "Method" lists the number of translatable API methods; column "Java" lists the number of passing test cases in Java; and column "C#" lists the number of translated test cases in C# without compilation errors. Here, many test cases are translated with compilation errors for two factors, which are not general or not related with API translation: (1) to prepare input values of translatable API methods, Randoop introduces API elements that are not translatable; (2) the number of compilation errors increases since Randoop produces many redundant code portions. We did not use Randoop to generate test cases for Net2Java and Converter, since the two tools translate too few API elements in C# to generate meaningful method sequences. In total, about half of the generated test cases fail, and the result shows that behavioral differences are pervasive in API mapping relations between Java and C#. The pervasive behavioral differences highlight the importance of our study.

Behavioral differences concerning inputs and outputs. TeMAPI leverages Pex to detect behavioral differences concerning inputs and outputs. As shown in Table 2, when leveraging Pex, more than 20,000 test cases failed. Given this large number of failures, we inspected 3,759 failing test cases selected as follows. For Net2Java and Converter, we inspected all the failing test cases, and for Java2CSharp, JLCA, and Sharpen, we inspected test cases of the `java.lang` package. We selected this package, since it is widely used in Java applications. Our findings are as follows:

Finding 1. 36.8% test cases show behavioral differences with respect to the handling of `null` inputs.

We found that many API methods in Java and their translated API methods in C# have behavioral differences when `null` values are passed as inputs. For example, JLCA translates the `java.lang.Integer.parseInt(String,int)` method in Java to the

[8] `http://msdn.microsoft.com/en-us/library/ff462634.aspx`

Table 2. Overall testing result

Name	Pex					Randoop				
	Number	E-Tests		A-Tests		Method	Java	C#	A-Tests	
		M	%	M	%				M	%
Java2CSharp	15,458	5,248	34.0%	3,261	21.1%	1,996	15,385	2,971	2,151	72.4%
JLCA	33,034	8,901	26.9%	6,944	21.0%	7,060	16,630	1,067	295	27.6%
Sharpen	2,730	662	24.2%	451	16.5%	586	13,532	936	456	48.7%
Net2Java	352	40	11.4%	261	74.1%	n/a	n/a	n/a	n/a	n/a
Converter	762	302	39.6%	182	23.9%	n/a	n/a	n/a	n/a	n/a
Total	52,336	15,153	29.0%	11,099	21.2%	9,642	45,547	4,974	2,902	58.3%

`System.Convert.ToInt32(string,int)` in C#. Given `null` and 10 as inputs, the method in Java throws `NumberFormatException`, but the method in C# returns 0. We notice that translation tools resolve some of these behavioral differences by providing custom functions. For example, `java.lang.String.valueOf(Object)` method in Java and the `System.Object.ToString()` in C# behave differently when a `null` value is passed as input. To resolve this difference, Sharpen translates the method in Java to its own implemented method in C#.

Implication 1. For implementers of API libraries, behaviors for `null` inputs are largely controversial. We suggest that implementers clearly define behaviors of `null` inputs. Our result shows that many such differences are left to programmers. When programmers translate API methods, they should handle `null` inputs carefully.

Finding 2. 22.3% test cases show differences among returned `string` values.

We found that two mapped methods typically return different `string` values. For example, each class in Java has a `toString()` method, and each class in C# has a `ToString()` method. Many translation tools map the two API methods, but the return values of the two methods are different in most cases. Besides the preceding two methods, many API classes declare methods such as `getName` or `getMessage`, and these methods also return `string` values that are quite different. Overall, we found that none of the five tools resolves this category of behavioral differences.

Implication 2. Although a method in Java and a method in C# have the same functionality, the two methods can return different `string` values. Programmers should be cautious while using these values, since they are typically different across languages.

Finding 3. 11.5% test cases show the behavioral differences of input domains.

We found that API methods in Java and their mapped API methods in C# can have different input domains. For 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. As another 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 the method in Java to the `System.Boolean.Parse(String)` method in C#. The C# method throws `FormatException` given the same input since it checks for illegal inputs.

Implication 3. Programmers should be cautious while dealing with methods whose arguments are close to minimum or maximum values of respective data types, since the

ranges of these values can be different between different languages. Cook and Dage [5] pointed out that an updated API method in a single programming language can also have different input domains. Adopting their approach may help deal with different input domains across languages.

Finding 4. 10.7% test cases show behavioral differences with respect to implementations.

We found that API libraries in different languages may have different implementations of the same functionalities. For example, we found that, unlike C#, Java considers "\" as an existing directory. Such differences can also indicate defects in translation tools. For 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`. As another 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 -2147483648 as input, the method in Java returns "80000000", but the method in C# returns "\080000000". Four behavioral differences including the preceding two were confirmed as defects by developers of Java2CSharp[9].

Implication 4. Implementers of API libraries can have different understandings on functionalities of specific methods. Some of such differences reflect different natures of different languages, and some other differences indicate defects in translation tools. Programmers should learn the natures of different programming languages (*e.g.*, different definitions of paths and files) to figure out such differences.

Finding 5. 7.9% test cases show behavioral differences with respect to handling of exceptions.

We found that some mapped API methods throw unmapped exceptions. For example, the `java.lang.StringBuffer.insert(int,char)` method in Java throws `ArrayIndexOutofBoundsException`, when indexes are out of bounds. Java2CSharp translates the method in Java to the `System.Text.StringBuilder.Insert(int, char)` method that throws `ArgumentOutOfRangeException` when indexes are out of bounds. Java2CSharp translates `ArrayIndexOutofBoundsException` in Java to `IndexOutOfRangeException` in C#. As Java and C# both allow unchecked exceptions, translated code can fail to catch corresponding exceptions.

Implication 5. Implementers of API libraries may design different exception handling mechanisms. This category of differences is quite challenging to be resolved for translation tools. When programmers translate `try-catch` statements, they should be aware of these differences. Otherwise, exception handling code may not be invoked in the translated version or may even become a dead code.

Finding 6. 2.9% test cases show the behavioral differences caused by constants.

We found that mapped constants may have different values. For example, the `java.lang.reflect.Modifier` class in Java has many constants to represent modifiers (*e.g.*, `FINAL`, `PRIVATE`, and `PROTECTED`). Java2CSharp translates these constants to the constants of the `ILOG.J2CsMapping.Reflect` class in C#. Between the two classes, constants such as `VOLATILE` and `TRANSIENT` are of different values. Sometimes dif-

[9] `http://tinyurl.com/3z45c5c`

ferent values reflect different settings of two languages. For example, translation tools often translate the `java.lang.Double.MAX_VALUE` field in Java to `System.Double.MaxValue` field in C#. The value of the former is 1.7976931348623157E+308, and the value of the latter is 1.79769313486232E+308.

Implication 6. Implementers of API libraries may store different values in constants, even if two constants have the same name. The different values sometimes reflect different settings such as different bounds of data types between two languages. Programmers should be aware of these differences while using constants.

The remaining 7.9% failing test cases are related to the API methods that 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. Translation tools often map the two methods. Since each object has a unique hash code, the two methods of two receiver objects return different values. Since these exposed behavioral differences are false, they can be considered as false positives or limitations of our work.

Behavioral differences concerning method sequences. TeMAPI leverages Randoop to detect behavioral differences concerning method sequences. After browsing translated test cases, we notice that some translated test cases have compilation errors, even if these test cases use only translatable API elements.

Finding 7. API classes in Java and API classes in C# can have different inheritance hierarchies, and the difference can lead to compilation errors.

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

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

Since the preceding two API classes in Java are related through inheritance, the test case has no compilation errors. JLCA translates the test case from Java to C# as follows:

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

Since the two translated C# classes do not have the inheritance relation, the translated test case in C# has compilation errors.

Implication 7. A source API library and its target API library can design different inheritance hierarchies of classes. It is quite difficult for translation tools to resolve this category of behavioral differences. When programmers translate code, they should be aware of such differences. For example, when they translate cast statements, they should double check whether the target API elements have a similar inheritance hierarchy.

TeMAPI removed those translated test cases with compilation errors. Among remaining test cases, for each translation tool, we investigated only the first 100 failing test cases. The percentages are as follows:

Finding 8. 3.4% test cases fail for method sequences.

We found that random method sequences can violate specifications of API libraries. One category of such specification is described in our previous work [28]: closed resources should not be manipulated. Java sometimes allows programmers to violate such specifications although return values can be meaningless. Besides method sequences that are related to specifications, we found that field accessibility also leads to failures of test cases. For example, a generated test case in Java is as follows:

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

JLCA translates it to C# as follows:

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

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

Implication 8. Legal method sequences can become illegal after translation, since the target language may be more strict to check method sequences, and other factors such as field accessibility can also cause behavioral differences. In most of such cases, programmers should deal with these differences themselves.

Remaining test cases failed for the following reasons: 45.0% for input domains, 34.0% for `string` values, 5.3% for different implementations, 4.0% for exception handling, 3.0% for `null` inputs, 2.0% for values of constants, and 0.3% for random values. The remaining 3.0% test cases fail, due to that translation tools 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`.

When generating test cases, Pex generates one test case for each feasible path, whereas Randoop uses a feedback-guided random strategy. As a result, the category distribution revealed by Pex more reflects the category distribution of unique behavioral differences than the category distribution revealed by Randoop, since each test case generated by Pex typically reflects a unique behavior.

Threats to validity. The threats to internal validity include human factors for inspecting behavioral differences. To reduce these threats, we re-ran those failing test cases, and inspected those test cases carefully. This threat could be further reduced by involving more third-party members for inspecting the detected differences.

The threats to external validity of our evaluation include the representativeness of the subject translation tools, selected programming languages (Java and C#), and the selected package for inspection. In future work, this threat could be reduced by including more translation tools and inspecting test cases that are related to other packages. The threats to external validity of our evaluation also include unexplored behaviors of APIs. Taneja *et al.* [22] proposed an approach that generates test cases for database applications via mock objects. Thummalapenta *et al.* [23] proposed an approach that

mines method sequences from real code for test generation. In future work, we plan to leverage their approaches and to integrate more testing tools (*e.g.*, JPF [25]), so that our work can detect more behavioral differences.

4 Discussion and Future Work

Improving translation tools and detecting related defects. Our previous work [27] mines unknown mapping relations from existing projects with variants in different languages. In future work, we plan to extend our previous work [27] to resolve some detected behavioral differences. In addition, when programmers write code in an unfamiliar language, they may follow idioms of their familiar languages. This practice can lead to defects, since our findings show that differences between two programming languages can be subtle. In future work, we plan to propose approaches that leverage our findings to detect such defects as well.

Surveying programmers with language-migration experiences. When migrating legacy systems, many programmers choose to translate applications manually from scratch. Although they know many API mapping relations, they may not develop any translation tools, and our process cannot detect behavioral differences in their known mapping relations residing in their minds. In future work, we plan to conduct a survey to collect API mapping relations from those experienced programmers. In addition, we plan to conduct a survey to investigate whether these programmers are aware of behavioral differences exposed by us, and if they are, how they deal with such differences.

Analyzing translation of more programming languages. To improve the potential impact of our work, we could analyze translation of more programming languages. Ravitch *et al.* [18] proposed an approach that generates bindings to expose low-level languages to high-level languages. In future work, we plan to adapt their wrappers, so that we can analyze translation of more programming languages, even if the two languages under analysis are fundamentally different.

5 Related Work

API translation. API translation is an important aspect of language migration. (1) Language-to-language migration. Song and Tilevich [20] proposed an enhanced specification to improve source-to-source translation approaches. Zhong *et al.* [27] mined API mapping relations from existing applications in different languages to improve API translation. (2) Library update migration. Henkel and Diwan [10] proposed an approach that captures API refactoring actions to update client code with the latest APIs. Xing and Stroulia [26] proposed an approach that recognizes the changes of APIs by comparing the differences between two versions of libraries. Meng *et al.* [15] proposed an approach that mines API mapping relations from revision histories of API libraries. Balaban *et al.* [1] proposed an approach to migrate code when mapping relations of libraries are available. (3) Migrating from one API library to alternative libraries. Dig *et al.* [6] proposed *CONCURRENCER* that translates sequential API elements to concurrent API elements in Java. Nita and Notkin [16] proposed twinning to automate the process given that API mapping is specified. (4) Migrating to more efficient APIs. Kawrykow *et al.* [13] proposed an approach that compares client code with API implementation code, and thus allows programmers to choose more efficient APIs. Our work

detects behavioral differences between mapped API elements, and the results can also help the preceding approaches translate applications resulting in fewer defects.

API comparison. Shi *et al.* [19] compared API documents for behavioral differences when APIs evolve. Their approach is not suitable to compare API libraries in different languages, where API documents are typically fundamentally different. Hou and Yao [11] analyzed such behavioral differences for the intents behind API evolution. Srivastava *et al.* [21] proposed an approach that compares API implementations for their behavioral differences, and cannot analyze the .NET frameworks whose code is not available. Bartolomei *et al.* [2, 3] reported their experiences on implementing wrappers between SWT and Swing. The preceding approaches compare APIs in a single language. Our work complements the preceding approaches by exposing behavioral differences of API elements in different languages.

Language comparison. Researchers conducted various empirical comparisons on languages. Garcia *et al.* [8] presented a comparison study on six languages to reveal differences with respect to generics. Cabral and Marques [4] compared exception handling mechanisms between Java and .NET programs. To the best of our knowledge, no previous work systematically compares behavioral differences of API elements from different languages. Our work enables us to produce such a comparison study, complementing the preceding empirical comparisons.

6 Concluding Remarks

Behavioral differences among API elements of different languages are pervasive and could introduce defects in the translated code. Often, programmers are not aware of these differences either due to a large number of mapping relations or due to the fact that differences happen only for specific input values. In this paper, we presented the first empirical study that exposes behavioral differences among API elements between Java and C#. Our results can help improve existing translation tools and also assist programmers to better understand API behavioral differences between different languages.

Acknowledgments

Hao Zhong's work is supported by the National Natural Science Foundation of China No. 61100071. Tao Xie's work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNF-0958235, CNS-1160603, and an NSA Science of Security Lablet Grant, as well as the National Science Foundation of China No. 61228203.

References

1. I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proc. 20th OOPSLA*, pages 265–279, 2005.
2. T. Bartolomei, K. Czarnecki, and R. Lammel. Swing to SWT and back: Patterns for API migration by wrapping. In *Proc. ICSM*, pages 1–10, 2010.
3. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm. Study of an API migration for two XML APIs. In *Proc. 2nd SLE*, pages 42–61, 2009.
4. B. Cabral and P. Marques. Exception handling: A field study in Java and. NET. *Proc. 21st ECOOP*, pages 151–175, 2007.

5. J. Cook and J. Dage. Highly reliable upgrading of components. In *Proc. 21st ICSE*, pages 203–212, 1999.
6. D. Dig, J. Marrero, and M. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *Proc 31st ICSE*, pages 397–407, 2009.
7. 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.
8. 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.
9. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.
10. J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proce. 27th ICSE*, pages 274–283, 2005.
11. D. Hou and X. Yao. Exploring the intent behind API evolution: A case study. In *Proc. WCRE*, pages 131–140, 2011.
12. T. Jones. *Estimating software costs*. McGraw-Hill, Inc. Hightstown, NJ, USA, 1998.
13. D. Kawrykow and M. P. Robillard. Improving API usage through automatic detection of redundant code. In *Proc. ASE*, pages 111–122, 2009.
14. S. Koushik, M. Darko, and A. Gul. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.
15. S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *Proc. 34th ICSE*, pages 353–363, 2012.
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. T. Ravitch, S. Jackson, E. Aderhold, and B. Liblit. Automatic generation of library bindings using static analysis. In *Proc. PLDI*, pages 352–362, 2009.
19. L. Shi, H. Zhong, T. Xie, and M. Li. An empirical study on evolution of API documentation. In *Proc. FASE*, pages 416–431, 2011.
20. M. Song and E. Tilevich. Enhancing source-level programming tools with an awareness of transparent program transformations. In *Proc. 24th OOPSLA*, pages 301–320, 2009.
21. V. Srivastava, M. Bond, K. McKinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple API implementations. In *Proc. 32nd PLDI*, pages 343–354, 2011.
22. K. Taneja, Y. Zhang, and T. Xie. MODA: Automated test generation for database applications via mock objects. In *Proc. 26th ASE*, pages 289–292, 2010.
23. S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proc. ESEC/FSE*, pages 193–202, 2009.
24. N. Tillmann and J. De Halleux. Pex: white box test generation for. NET. In *Proc. 2nd TAP*, pages 134–153, 2008.
25. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.
26. Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.
27. 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.
28. 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.