Automated Testing of API Mapping Relations

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

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

ABSTRACT

Application Programming Interface (API) mapping describes mapping relations of APIs provided by two languages. With API mapping, migration tools are able to translate client code that uses APIs from one language into another language. However, given the same inputs, two mapped APIs can produce different outputs. The different behaviors introduces defects in translated code silently since translated code may have no complication errors. In this paper, we propose an approach, called TeMaAPI (**Te**sting **Ma**pping relations of **API**s), that detects different behaviors of mapped APIs via testing. To detect different behaviors, the test oracle of TeMaAPI is that two mapped APIs should produce the same outputs given the same inputs. Based on our approach, we implement a tool and conduct evaluations on migration tools such as Java2CSharp and Tangible C# to Java convertor. The results show that TeMaAPI detects ?? different behaviors between mapped APIs of Java2CSharp and ?? different behaviors between mapped APIs of Tangible C# to Java convertor.

1. INTRODUCTION

Since the inception of computer science, many programming languages (such as Cobol, Fortran and Java) have been introduced to serve specific requirements[1]. For example, Cobol is introduced specifically for developing business applications. In general, software companies or open source organizations often release their applications in different languages to survive in competing markets and to address various business requirements such as platform independence. For example, to achieve language and platform independence, Spenkelink [18] translated the Compose* [7] project in C# to Compose*/J in Java. A recent study [11] shows that nearly one third applications exist in different languages.

In general, translating applications manually from one programming language, referred to L_1, to another language, referred to as L_2, is a tedious and error-prone task, since real-world applications often include thousands of lines of code. To reduce manual effort, programmers use existing automatic translation tools such as JLCA[2] or develop their own translation tools. For example, Salem *et al.* [2] translated the BLUE financial system of the ICT company from Java to C# using the JLCA tool. On the other hand, programmers of db4o[3] developed their own translation tool, called sharpen[4], for translating db4o from java to C#. The primary goal of these tools is to translate an application in one language to another language such that both versions exhibit the same behavior.

```
01: package java.lang::System{
02:   class java.lang.String :: System:String{
03:   method valueOf(Object)  pattern = @1.ToString();
04:   ...}}
```

Figure 1: An example API Mapping relation in Java2CSharp.

```
Java Code
05:  Object obj = ...
06:  String value = java.lang.String.valueOf(obj);
C# Code translated by Java2CSharp
07:  Object obj = ...
08:  String value = obj.ToString();
```

Figure 2: Original Java and its translated C# code.

Automatic translation tools use mappings among APIs from L_1 to L_2, referred to as *API mapping relations*, as a basis for translating applications. Figure 1 shows an example API mapping relation from the API method `String.valueOf()` in Java (L_1) to `String.ToString()` in C# (L_2). Figure 2 shows original Java code and its equivalent C# code translated using an automatic translation tool, called Java2CSharp[5]. These API mapping relations play an important role in preserving the same behavior between the original and translated applications. We hereby use notations App_1 and App_2 to represent applications in languages L_1 and L_2, respectively. Any inconsistencies among these mapping relations could result in different behaviors between App_1 and App_2 applications.

Writing API mapping relations that exhibit the same behavior in both L_1 and L_2 is quite challenging, since existing languages such as Java or C# provide a large number of APIs. Furthermore, APIs with similar names and structures can have different behaviors in different languages. For example, Panesar[6] shows that even common methods such as `String.subString(int, int)` can have different behaviors between Java and C#. Despite the large number of APIs, El-Ramly *et al.* [5] shows that it is essential to develop new methods for translating applications from Java to C#, although both these languages appears to be similar.

To further illustrate the challenges, we next provide real examples that show the behavioral differences among API mappings defined in Java2CSharp. Consider the API mapping and translated

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

[2] `http://tinyurl.com/2c4coln`
[3] `http://www.db4o.com`
[4] `http://tinyurl.com/22rsnsk`
[5] `http://j2cstranslator.sourceforge.net/`
[6] `http://tinyurl.com/3xpsdtx`

```
09: public abstract class Configuration {
10: protected void setUpStringMappings() {
11:   mapMethod("java.lang.String.valueOf",
              runtimeMethod("getStringValueOf"));
12: ...} }

C# Code translated by sharpen
13: Object obj = ...
14: String value = getStringValueOf(obj); ...

15: public static string GetStringValueOf(object value){
16:   return null == value? "null": value.ToString();
17: }
```

Figure 3: An example API mapping relation in sharpen.

code shown in Figures 1 and 2, respectively. Although the translated code does not include compilation errors, the translated code behaves different from the original code for certain inputs. For example, assigning a `null` value to `obj` in Line 5, assigns the value "`null`" to `value` in Line 6. In contrast, assigning a `null` value to `obj` in Line 7 results in a `NullReferenceException`. To address this behavioral difference, another automatic translation tool, called sharpen, uses a different API mapping relation shown in Lines 09 to 12 in Figure 3. Lines 13 to 17 show the translated C# code for the Java code shown in Figure 2. In particular, Line 16 ensures that the string "`null`" is assigned to `value` in Line 14 when a `null` value is assigned to `obj`. Although, sharpen's API mapping relation addresses the issue with `null`, this relation still includes other behavioral differences. For example, assigning a `false` value to `obj` in Line 5 results in "`false`" value to be assigned to `value` in Line 6. However, assigning a `false` value to `obj` in Line 14 results in "**F**`alse`" value to be assigned to `value` in Line 14. If the string variable `value` is used in a case-sensitive comparison afterwards, the preceding difference could result in different outputs. These differences are relatively difficult to detect, since a programmer typically does not aware of the internal logic of the API method to construct appropriate test cases. These examples show the existence of behavioral differences among API mapping relations and shows the necessity of detecting such differences among APIs.

There exist regression testing approaches [6, 13] that accept two versions of an application and detect behavioral differences between those versions. Although our current problem of detecting behavioral differences among API mapping relations can be considered as a form of regression testing problem, no existing approach can be used to detect such differences. The primary reason is that existing approaches require both the versions under consideration belong to the same language. However, in our context, both versions belong to different languages, making these existing approaches inapplicable. Therefore, to address these preceding issues, we propose a novel approach, called TeMaAPI (**Te**sting **Ma**pping relations of **API**s), that automatically generates tests that detect behavioral differences among API mapping relations.

In particular, TeMaAPI generates test cases on one version App_1 of the application (in a language) and translates those tests into the other language L_2. TeMaAPI next applies translated test cases on the other version App_2 to detect behavioral differences. TeMaAPI addresses two major technical challenges in effectively detecting behavioral differences. (1) Using a naive technique such as generating test cases with `null` values may not be significant in detecting behavioral differences among API mapping relations. Since we focus on object-oriented languages such as Java or C#, to detect behavioral differences, generated test cases need to exercise various object states, which can be achieved using method-call sequences. To address this issue, TeMaAPI leverages two existing state-of-the-art test generation techniques: random [16] and dynamic-symbolic-execution-based [8, 12, 21]. (2) Generating test cases on App_1 and applying those test cases on App_2 may not exercise many behaviors of APIs in App_2, thereby related defects cannot be detected. To address this issue, TeMaAPI uses a round-trip technique that also generates test cases on App_2 and applies them on App_1. We describe more details of our approach and how we address these challenges in subsequent sections.

This paper makes the following major contributions:

- A novel approach, called TeMaAPI, that automatically generates test cases that detect behavioral differences among API mapping relations. Given a translation tool, TeMaAPI detects different behaviors of its all API mapping relations automatically. It is important to detect these different behaviors, since they can introduce defects in translated code silently.
- Test adequacy criteria proposed for generating sufficient test cases to test API mapping. TeMaAPI targets at generating adequate test cases that can reveal all behaviors of APIs to test their mapping relations.
- A tool implemented for TeMaAPI and two evaluations on ?? projects that include ?? mapping relations from Java to C#, and ?? mapping relations from C# to Java. The results show that our tool detects ?? unique defects of mapping relations...

The rest of this paper is organized as follows. Section 2 presents our test adequacy criteria. Section 3 illustrates our approach using an example. Section 4 presents our approach. Section 5 presents our evaluation results. Section 6 discusses issues of our approach. Section 7 presents related work. Finally, Section 8 concludes.

2. TEST ADEQUACY CRITERIA

As stated by Andrews *et al.* [3], a test adequacy criterion is a predict, and a test suite is adequate with respect to a criterion only if all defined properties of the criterion are satisfied by the test suite. In this paper, we define four test adequacy criteria for API mapping relations as follows.

Path criterion. Given an API invocation inv in a language L_1, and its mapped API invocation $\psi(inv)$ in a language L_2, a adequate test suite should cover all internal paths of inv. This criterion ensures that $\psi(inv)$ in L_2 return the same values with inv in L_1 with respect to all paths of inv.

Sequence criterion. Given API invocations $inv_1, \ldots, inv_m$ in a language L_1 and its mapped API invocations $\psi(inv_1, \ldots, inv_m)$ in a language L_2, a adequate test suite should cover all states of obj with respect to the field criterion. This criterion ensures that $\psi(inv_1, \ldots, inv_m)$ in L_2 return the same values with $inv_1, \ldots, inv_m$ in L_1 with respect to all call sequences of $inv_1, \ldots, inv_m$.

TeMaAPI targets at generating test cases that satisfy both the path criterion and the sequence criterion for testing API mapping relations.

3. EXAMPLE

We next illustrate the basic steps to detect the different behaviors of the API mapping relation as discussed in Section 1.

Translating Generated client code. First, TeMaAPI generates a client code method for each API method an each API field of the `java.lang.String` class. For example, the generate clinet code for the `valueOf(Object)` method is as follows:

```
public java.lang.String testvalueOf64sm0(Object m0){
  return java.lang.String.valueOf(m0);}
```

After that, we use a migration tool such as Java2CSharp to translate generated client code from Java to C#.

Removing translated client-code methods with compilation errors. A migration tool typically cannot cover all API mapping

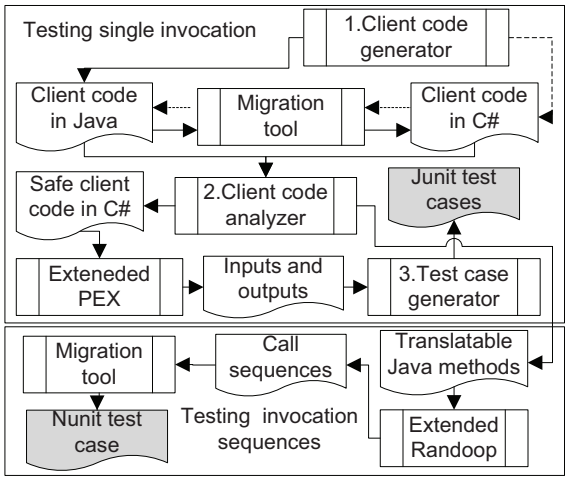

Figure 4: Overview of TeMaAPI

relations, so some translated client-code methods can have compilation errors if their used API methods or API fields are not covered. TeMaAPI parses translated code, and removes all client-code methods with compilation errors. The remaining client-code methods are testable since no compilation errors are left.

In this example, the translated C# `testvalueOf64sm` method is as follows:

```
public System.String TestvalueOf64sm(Object m0) {
  return m0.ToString();}
```

This method contains no compilation errors since Java2CSharp covers the mapping relation of the `valueOf(Object)` method in Java as shown in Figure 1.

Generating and executing test cases to reveal different behaviors. TeMaAPI leverages various techniques to generate test cases for remaining client-code methods. For example, TeMaAPI extends Pex [21], so that it records all inputs and output of each execution. In this example, for the `TestvalueOf64sm` method in C#, one recorded input is a boolean value, and the corresponding output is "False". Based on the input and output, TeMaAPI generates a JUnit test case as follows:

```
@Test
public void testvalueOf64sm0(){
  sketch.Test_java_lang_String obj =
        new sketch.Test_java_lang_String();
  boolean m0 = false;
  Assert.assertEquals("False", obj.testvalueOf64sm(m0));
}
```

This JUnit test case fails since the `testvalueOf64sm` method in Java returns "false" instead of "False". Thus, TeMaAPI detects a different behavior between the `valueOf(Object)` method in Java and its mapped C# API methods in Java2CSharp.

4. APPROACH

Given a migration tool between Java and C#, TeMaAPI generates various test cases to reveal different behaviors of the tool's API mapping relations. Figure 4 shows the overview of TeMaAPI. It is able to test mapping relations of a single API invocation and also mapping relations of multiple API invocations.

4.1 Translating Generated client code

Given a migration tool, TeMaAPI first extracts its validate mapping relations of APIs. It is challenging to extract such mapping relations directly from a migration tool for two factors: (1) different migration tools may follow different styles to describe API mapping relations. For example, as shown in Section 1, the API mapping relations of Java2CSharp are described in its mapping files, but the API mapping relations of sharpen are hard-coded in its source files. (2) commercial migration tools typically hide their API mapping relations in binary files. Due to the two factors, TeMaAPI does not extract API mapping relations directly from a migration tool, but chooses to analyze translated code of a migration tool. We choose to use migration tools to translate simple client code instead of existing projects for two considerations: (1) Existing projects typically use quite a small set of APIs, so many API mapping relations may be not covered; (2) a single method of an existing project may use multiple APIs, so it may be difficult to analyze which APIs are not mapped. For the preceding consideration, TeMaAPI chooses to generate client code instead of using existing client code.

TeMaAPI relies on the reflection technique [14] provided by both Java and C# to generate client code for translation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

TeMaAPI ignores generic methods for simplicity, and organizes all generated client code methods by the corresponding class C. For a migration tool that translates from Java to C#, TeMaAPI generates client code in Java as shown by the solid line of Figure 4, and for a migration tool that translates from C# to Java, TeMaAPI generates client code in C# as shown by the dotted line of Figure 4. When TeMaAPI generates client code in C#, it ignores `unsafe` and `delegate` methods and methods whose parameters are marked as `out` or `ref`. These methods may produce more than one output, and our generated client code has only one output. In addition, Java does not have corresponding keywords, so there are typically no mapped methods in Java for these C# methods. After TeMaAPI generate client-code methods, we translate them using a migration tool under experiments.

After generated code is translated, TeMaAPI parses translated code and removes translated methods with compilation errors. We refer to the remaining methods as safe client code. Safe client code is testable since it has no compilation errors. Based on these methods, TeMaAPI tests single API invocations (Section 4.2). From remaining methods, TeMaAPI analyzes which API invocations can be translated without introducing compilation errors. Thus, TeMaAPI generates a list of translatable API invocations for a given migration tool. Based on the list, TeMaAPI further tests API invocation sequences (Section 4.3).

4.2 Testing Single API Invocations

Pex [21] is a white-box test generation tool for .Net based on dynamic symbolic execution. Basically, Pex repeatedly executes a method under test, so that it explores all feasible paths of the method. To reduce the efforts to explore paths, Pex leverage various search strategies. For example, Xie *et al.* [24] propose a search strategy called Fitnex that uses state-dependent fitness values to guide path exploration of Pex.

TeMaAPI extends Pex, so that it generates test cases that satisfy the path criterion defined in Section 2. In particular, for each translated C# client code method, TeMaAPI records the inputs and the corresponding out of each searched path of Pex to local files. Based on recorded inputs and outputs, TeMaAPI generates JUnit test cases to ensure each mapped API invocations produce the same output give the same inputs. For example, TeMaAPI records that given a integer whose value is 0, the output of the `TestvalueOf57sm` method in C# is "0". Based on the preceding input and output, TeMaAPI generates a JUnit test case for the `testvalueOf57sm` method in Java as follows:

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

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

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

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

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

Table 1: Subject tools

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

4.3 Testing Invocation Sequences

Randoop [16] is a test generation tool for Java. It randomly generates test cases based on already generated test cases in a feedback-directed manner. As stated by Tillmann and Halleux [21], the key difference between Pex and Randoop lies in that Pex generates test cases for a single method whereas Randoop generates test cases that involve multiple methods.

TeMaAPI extends Randoop, so that it generates test cases that satisfy the sequence criterion defined in Section 2. As described in Section 4.1, TeMaAPI is able to extract translatable API invocations of a given migration tool. TeMaAPI restricts the search scope of Randoop, so that it generates JUnit test cases use translatable API invocations. Since generated JUnit test cases use only translatable API invocations of a given migration tool, the migration tool can translate these JUnit test cases in Java into NUnit test cases automatically. We find that some JUnit test cases generated by Randoop fail or produce errors. Before translating, TeMaAPI removes all such JUnit test cases, so all JUnit test cases run successfully. If a translated NUnit test case fails or produces errors, the test case may indicate different behaviors of API invocation sequences.

5. EVALUATIONS

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

1. How effectively can existing migration tool translate API invocations (Section 5.1)?
2. How effectively can our approach detect different behaviors of single API invocations (Section 5.2)?
3. How effectively can our approach detect different behaviors of API invocation sequences (Section 5.3)?

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

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

5.1 Translating Generated Client Code

For Java to C# tools, we use TeMaAPI to generate client-code methods for J2SE 6.0[7]. As described in Section 4.1, TeMaAPI ig-

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

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

Table 2: Translation results of Java to C# tools

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

Table 3: Translation results of C# to Java tools

nores all generic API methods. Table 2 shows the translation results. Column "Type" lists the types of generated methods. In particular, "sfg" denotes getters of static fields; "sfs" denotes setters of static fields; "nfg" denotes getters of non-static fields; "nfs" denotes setters of non-static fields; "sm" denotes static methods; "nm" denotes non-static methods; and "Total" denotes the sums of all methods. Column "Number" lists numbers of corresponding types of methods. Columns "Java2CSharp", "JLCA", and "sharpen" list the translation results of corresponding migration tools. For these columns, sub-column "M" lists the number of translated methods without compilation errors, and sub-column "%" lists the percentage from translated method without compilation errors to corresponding generated methods.

From the results of Table 2, we find that it is quite challenging for a migration tool to cover all API invocations. One challenge lies in that APIs are quite large in size. Although JLCA can translate 43678 generated methods, it covers only 22.4% of total generated methods. The other challenge lies in that many API invocations of different languages cannot be accurately mapped. For example, as pointed out by our previous work [27], one API method in one language may be mapped to several API methods in another language. We find that all the three migration tools have techniques to deal with the differences. In particular, Java2CSharp and sharpen develop their own assemblies and map some API invocations to their implemented assemblies instead of standard C# API invocations. For example, Java2CSharp maps the `java.lang.Class.forName(String)` method in Java to the `ILOG.J2CsMapping.Reflect.Helper.GetNativeType(String)` method in C#, and the latter method is provided by Java2CSharp. JLCA does not implement additional assemblies, but generate additional source code to hide different behaviors. For example, TeMaAPI generates one method for `java.awt.Dialog.add(Component m0,int m1, Frame c0)`:

```
public Component testadd79nm(Component m0,int m1,Frame c0){
  java.awt.Dialog obj = new java.awt.Dialog(c0);
  return obj.add(m0,m1);
}
```

JLCA translates the preceding method as follows:

```
@Test
public virtual Control testadd79nm(Control m0, int m1,
                                   Form c0){
  Form obj = SupportClass.DialogSupport.CreateDialog(c0);
  Control temp_Control;
  temp_Control = m0;
  obj.Controls.Add(temp_Control);
  if (m1 != -1)
    obj.Controls.SetChildIndex(temp_Control, m1);
  return temp_Control;
}
```

JLCA maps the API method in Java into many API methods in C#. Besides adding a temporary variable named `tmp_Control`, JLCA generates a class named `SupprtClass` where more API invocations are introduced.

Although all the three migration tools take the different behaviors of mapped APIs seriously, they do not cover all differences of API mapping relations. For example, when JLCA translates generated code, it generates a report with 1265 warning messages for different behaviors of API mapping relations. One warning message is "Method 'java.lang.String.indexOf' was converted to 'System.String.IndexOf' which may throw an exception". Still, the report does not tell programmers when such an exceptions will be thrown, and the rest of our evaluation targets at the problem.

For C# to Java migration tools, we use TeMaAPI to generate client-code methods for the .Net framework client profile[8]. As described in Section 4.1, besides generic methods, TeMaAPI also ignores methods whose parameters are marked with `out` or `ref`. Table 3 shows the translation results. Columns of Table 3 are of the same meanings with the columns of Table 2. From the results of Table 2, we find that TeMaAPI generates almost the same size of methods as it generates for J2SE 6.0, and both the two tool translate only quite a small size of API invocations. As described in the wikipedia[9], C# provides many features that Java does not have (*e.g.*, partial class, reference parameters, output parameters, and named arguments). We suspect that a C# to Java migration tool needs to deal with these differences, so many mapping relations of APIs are not addressed yet.

Comparing the translation results between Java-to-C# tools and C#-to-Java tools, we find that Java-to-C# tools cover much more API invocations. To fully explore the translation results of Java-to-C# tools, we present the results of the package level in Table 4. Column "Name" lists the names of Java packages. To save space, we omit the prefixes such as "java.", "javax.", and "org." if it does not introduce ambiguity. We also use short names for some packages. In particular, we use "acc." to denote the `javax.accessibility` package, "man." to denote the `javax.management` package, "java.sec." to denote the `java.security` package, and "javax.sec." to denote the `javax.security` package. Other columns of Table 4 are of the same meanings with the columns of Table 2. From the results of Table 4, we find that all the three migration tools cover the `java.io` package, the `java.lang` package, and the `java.util` package. The three packages may be quite important for most Java programs. Almost for all packages, JLCA covers more API invocations. In particular, JLCA covers GUI-related packages such as the `java.awt` package and the `javax.swing` package. As a result, JLCA can translate some Java programs with GUI interfaces whereas the other two tools cannot. Still, we find that all the three tools do not cover 15 packages. As most of these packages are related to networks, all the three tools will fail to translate a program that use network APIs.

In summary, we find that APIs are quite large in size, and migration tools typically cover only a small set of API invocations. Existing migration tools already notice the differences between mapped APIs, but many differences are left unsolved.

5.2 Testing Single Invocations

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

Name	No	Java2CSharp		JLCA		sharpen	
		M	%	M	%	M	%
awt	29199	0	0.0%	8637	29.6%	0	0.0%
bean	1768	20	1.1%	14	0.8%	0	0.0%
io	3109	592	19.0%	1642	52.8%	43	1.4%
lang	5221	1494	28.6%	2377	45.5%	791	15.2%
math	1584	101	6.4%	232	14.6%	0	0.0%
java.net	1990	52	2.6%	482	24.2%	10	0.5%
nio	536	30	5.6%	0	0.0%	0	0.0%
java.rmi	1252	0	0.0%	707	56.5%	0	0.0%
java.sec.	2797	50	1.8%	702	25.1%	0	0.0%
java.sql	3495	20	0.6%	183	5.2%	0	0.0%
text	1068	96	9.0%	321	30.1%	0	0.0%
util	9586	1372	14.3%	1879	19.6%	341	3.6%
acc.	237	1	0.4%	25	10.5%	0	0.0%
activation	538	0	0.0%	165	30.7%	0	0.0%
activity	252	0	0.0%	0	0.0%	0	0.0%
annotation	19	0	0.0%	0	0.0%	0	0.0%
crypto	625	0	0.0%	263	42.1%	0	0.0%
imageio	1261	0	0.0%	0	0.0%	0	0.0%
model	103	0	0.0%	0	0.0%	0	0.0%
man.	5380	2	0.0%	0	0.0%	0	0.0%
naming	3565	0	0.0%	1365	38.3%	0	0.0%
javax.net	285	0	0.0%	0	0.0%	0	0.0%
print	1534	0	0.0%	0	0.0%	0	0.0%
javax.rmi	45	0	0.0%	0	0.0%	0	0.0%
script	195	0	0.0%	0	0.0%	0	0.0%
javax.sec.	1435	0	0.0%	619	43.1%	0	0.0%
sound	515	0	0.0%	56	10.9%	0	0.0%
javax.sql	640	0	0.0%	0	0.0%	0	0.0%
swing	102389	10	0.0%	21364	20.9%	0	0.0%
tools	11	0	0.0%	0	0.0%	0	0.0%
transaction	264	0	0.0%	0	0.0%	0	0.0%
javax.xml	4188	34	0.8%	580	13.8%	0	0.0%
ietf	189	0	0.0%	0	0.0%	0	0.0%
org.omg	8937	0	0.0%	1578	17.7%	0	0.0%
w3c.dom	83	0	0.0%	14	16.9%	0	0.0%
org.xml	897	49	5.5%	473	52.7%	0	0.0%

Table 4: Translation results of package level

To test different behaviors of single invocations, TeMaAPI leverages Pex to search internal paths for C# client-code methods. These methods include the translated C# methods without compilation errors as shown in Table 2 and the generated C# methods that can be translated into Java without compilation errors as shown in Table 3. When Pex searches those paths, TeMaAPI records the inputs and output of each iteration. Based on these inputs and outputs, TeMaAPI generates JUnit test cases to ensure that the generated methods and the translated methods produce the same outputs given the same inputs. As it requires human interactions to test GUI related APIs, we filter out GUI related APIs such as awt and swing although some migration tools such as JLCA are able to translate some GUI related APIs. In addition, when Pex searches methods without return values, we ignore those paths that do not throw any exceptions since we cannot generate JUnit test cases for them.

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

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

Table 5: Results of testing single invocations

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

Table 5 and Table 6 show that many generated JUnit tests do not get passed. Still, we notice that even if a JUnit test case does not get passed, we do not have fully confidence that it is caused by different behaviors of mapped APIs. For example, some API methods such as `java.util.Random.nextDouble()` generate random values. As a result, we find that its corresponding JUnit test case fails, but we cannot conclude that these methods have different behaviors.

To better understand different behaviors of mapped API invocations, we manually inspected 3759 JUnit test cases that end with errors or failures. In particular, for tools such as Java2CSharp, JLCA, and sharpen, we investigate the test cases for the `java.lang` package since TeMaAPI generates many test cases for the package as shown in Table 4. For tools such as Net2Java and converter, we inspect all their test cases since TeMaAPI test cases in small sizes for them. Our findings are as follows:

Finding 1: 37.2% test cases show the different behaviors caused by null inputs or outputs.

We find that Java API invocations and their mapped C# API invocations can have different behaviors when inputs are null values. In some cases, a Java API method can accept null values, but its mapped C# API method will throw exceptions given a null value. One such example is shown in Section 1. In other cases, a Java API method will throw exceptions given a null value, but its mapped C# API method can accept null values. For example, JLCA maps the `java.lang.Integer.parseInt(String,int)` method in Java to the `System.Convert.ToInt32(string,int)` in C#. TeMaAPI detects that when the inputs of the C# method are null and 10, its output is 0. Given the same inputs, the Java method throws a `NumberFormatException`. We also find that given the same inputs, a method may produce null outputs, whereas its mapped method will not. For example, converter maps the `System.Collections.Queue.ToArray()` in C# to the `java.util.LinkedList.toArray()` method in Java. Given an empty list, the C# method produce a null value, whereas the Java method produce an empty array.

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

Table 6: Testing results of package level

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

Finding 2: 22.2% test cases show the different behaviors caused by stored string values.

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

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

Finding 3: 8.1% test cases show the different behaviors caused by exception handling.

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

Implication 3: Even if two methods are of the same functionality, they may produce exceptions that are not mapped. Programmers should be careful to deal with exception handling, unless migrations tools can hide the differences.

Finding 4: 3.9% test cases show the different behaviors caused by ranges of parameters.

We find that Java methods and their mapped C# methods may have different ranges for parameters. In particular, we find that Java API methods often do not check whether values are out of ranges. For example, the `java.lang.Double.shortValue()` method in Java accepts values that are larger than short values, but its mapped C# methods such as the `Convert.ToInt16(double)` method throw `OverflowException` when values are larger than short values. For another example, Java2CSharp maps the `java.lang.Boolean.parseBoolean(String)` method in Java to the `System.Boolean.Parse(String)` method in C#. Given a string whose value is "test", the Java method return false without checking its format, but the C# method throws `FormatException`.

Implication 4: Programmers should be aware of the different input ranges of API methods between Java and C#. As C# API methods typically check ranges of input, C# programmers may not check ranges of inputs themselves, and thus introduce potential defects in translated Java code.

Finding 5: 3.5% test cases show the different behaviors caused by static values.

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

Implication 5: Programmers should be aware of that static fields may have different values even if they has the same names. As these differences reveal that Java and C# may define different bounds for data types, programmers should also be aware of these different bounds if they need highly accurate results of extremely large or small calculation results.

Finding 6: 17.1% test cases show the different behaviors caused by different understanding or implementation.

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

Implication 6: Although programmers can come to agreement on functionalities of many API methods, they may have different

Name	NUnit	Failure	
		M	%
Java2CSharp	2971	2151	72.4%
JLCA	1067	295	27.6%
sharpen	936	456	48.7%
Total	4504	2813	62.5%

Table 7: Results of testing invocation sequences

understanding on some methods. In some cases, such differences may indicate defects in mapped API methods.

Finding 7: 8.0% test cases end with failures since corresponding API methods produce random values.

Besides the preceding mentioned methods whose functionalities are to generate random values, we find that other methods can also generate values. For example, each Java class has a `hashCode()` method inherited from the `java.lang.Object` class, and each C# class has a `GetHashCode()` inherited from the `System.Object` class. Both the two methods return a hash code for the current object, so migration tools such as JLCA map the two methods. As such code is generated randomly, we cannot conclude they have different behaviors although corresponding JUnit test cases all fail. To test such mapped API methods, we plan to introduce other test oracles in future work. For example, we can generate many test cases and check whether mapped methods generate random values of the same ranges.

In summary, TeMaAPI effectively detects many different behaviors of mapped single API invocations between Java and C#. We inspect some detected differences, and we find that various factors such as null values, string values, exception handling, ranges of parameters, static values, and different understanding can cause differences of mapped single API invocations.

5.3 Testing Invocation Sequences

To test different behaviors of API invocation sequences, TeMaAPI leverage Randoop to generate test cases that involve multiple API invocations. For each Java-to-C# tools, TeMaAPI first analyzes the translation results as shown in Table 2 for the list of translatable APIs in Java. When generating test cases, TeMaAPI extends Randoop, so that each generated test case use only translatable APIs. We find that Randoop can generate failure test cases or even test cases with compilation errors. TeMaAPI removes those test cases, so that the remaining test cases all get passed. After that, we use the corresponding migration tool to translate the remaining test cases from Java to C#. As the remaining JUnit test cases all get passed, translated NUnit test cases should also get passed.

Table 7 shows the results. Column "NUnit" lists numbers of NUnit test cases. Column "Failure" lists failed NUnit test cases. We do not list numbers of test cases with errors since NUnit does not separate errors from failures as JUnit does. Sub-column "M" lists numbers of test cases, and sub-column "%" lists percentages from failed test cases to total test cases. We find that JLCA achieves better results than other tools since its percentage is the lowest. For each tool, we further investigate the first 100 failed test cases, and we find that 93.3% failed test cases are accumulated by the found factors as shown in Section 5.2. In particular, 45.0% for ranges of parameters, 34.0 for string values, 5.3% for different understanding, 4.0% for exception handling, 3.0% for null values, 2.0% for values of static fields, and 0.3% for random values. We find that the search strategy of Randoop for building invocation sequences affect the distribution. For example, as Randoop generates invocation sequences randomly, many values are out of range or illegal. Java API methods typically do not check inputs, so generated JUnit run successfully. However, as C# API methods typically check inputs, so corresponding NUnit test cases fail with various exceptions. Besides those found factors that cause different behaviors of mapped APIs as follows:

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

We find that Java methods often do not check invocation sequences, whereas C# methods often do. As a result, a NUnit test case may fail because of illegal invocation sequences. For example, a generated JUnit test case is as follows:

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

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

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

Java allows programmers to access a stream even if the stream is closed, so the preceding JUnit test case run successfully. C# does not allow such accesses, so the preceding NUnit test case fails with `ObjectDisposedException`.

We also find that in some cases, single invocations get passed but their invocation sequences can fail. For example, a generated JUnit test case is as follows:

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

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

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

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

Implication 8: When translating from Java to C#, programmers should check carefully whether they violate speculations of libraries. One such specification describes that closed streams should not be manipulated. When translating from C# to Java, programmers should be careful since Java often does not check illegal invocation sequences as C# does.

Finding 9: 3.0% test cases end with failures since mapped API methods are not implemented.

We find that some migration tools such as Java2CSharp maps APIs in Java to C# APIs that are not implemented yet. For example, Java2CSharp maps the `java.io.ObjectOutputStream` class in Java to `ILOG.J2CsMapping.IO.IlObjectOutputStream` in C#, and the C# class is not implemented yet. As as result, corresponding NUnit test cases fail with `NotImplementException`. As the

Java exception for the C# exception is not defined in Java2CSharp, TeMaAPI cannot generate JUnit test cases for them based on the technique as described in Section 4.2.

Implication 9: As this difference introduces no compilation errors, programmers should test translated projects carefully to ensure each API method is called.

In summary, besides the factors listed in Section 5.2, we find more factors based on the results of testing invocation sequences. The results confirm that even if single invocations have no different behaviors, invocation sequences can have different behaviors.

6. DISCUSSION AND FUTURE WORK

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

Aligning client code.

7. RELATED WORK

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

Language migration. To reduce manual efforts of language migration [17], researchers proposed various approaches [9,15,22,23, 26] to automate the process. However, all these approaches focus on the syntax or structural differences between languages. Deursen *et al.* [22] proposed an approach to identify objects in legacy code. Their approach uses these objects to deal with the differences between object-oriented and procedural languages. As shown in El-Ramly *et al.* [5]'s experience report, existing approaches support only a subset of APIs for language migration, making the task of language migration a challenging problem. In contrast to previous approaches, our approach automatically mines API mapping between languages to aid language migration, addressing a significant problem not addressed by the previous approaches and complementing these approaches.

Library migration. With evolution of libraries, some APIs may become incompatible across library versions. To address this problem, Henkel and Diwan [10] proposed an approach that captures and replays API refactoring actions to update the client code. Xing and Stroulia [25] 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 client code when mapping relations of libraries are available. In contrast to these approaches, our approach focuses on mapping relations of APIs across different languages. In addition, since our approach uses ATGs to mine API mapping relations, our approach can also mine mapping relations between API methods with different parameters or between API methods whose functionalities are split among several API methods in the other language.

Mining specifications. Some of our previous approaches [1, 19, 20, 28, 29] focus on mining specifications. MAM mines API mapping relations across different languages for language migration, whereas the previous approaches mine API properties of a single language to detect defects or to assist programming.

8. CONCLUSION

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

9. REFERENCES

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

[2] S. Y. Al-Agtash, T. Al-Dwairy, A. El-Nasan, B. Mull, M. Barakat, and A. Shqair. Re-engineering BLUE financial system using round-trip engineering and Java language conversion assistant. In *SERP*, pages 657–663, 2006.

[3] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for uml design models. *Software Testing, Verification and Reliability*, 13(2):95–127, 2003.

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

[5] 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.

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

[7] C. Garcia. *Compose* A Runtime for the. Net Platform*. PhD thesis, University of Twente, 2003.

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

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

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

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

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

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

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

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

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

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

[18] R. Spenkelink. Porting Compose* to the Java Platform. Master's thesis, University of Twente, 2007.

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

[20] 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.

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

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

246–255, 1999.

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

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

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

[26] 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.

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

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