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, 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

As stated by Sebesta [14], modern programming languages start around 1958 to 1960 with the development of Algol, Cobol, Fortran and Lisp. Ever since then, thousands of programming languages came into existence as shown by HOPL website[1]. For various considerations, programmers often need to translate projects from one language into another language. For example, as stated by Spenkelink [15], to provide the language and platform independence, he translates Compose* [6] in C# into Compose*/J in Java. To relief the efforts of translating, programmers may use existing migration tools or even implement their own migration tools. For example, Salem *et al.* [2] report their experience of translating the BLUE financial system of the ICT company from Java to C# by the JLCA[2] tool. For another example, to translate db4o[3] from Java to C#, its programmers develop their own migration tool named sharpen[4].

[1] `http://hopl.murdoch.edu.au`
[2] `http://tinyurl.com/2c4coln`
[3] `http://www.db4o.com`
[4] `http://tinyurl.com/22rsnsk`

To translate a source file, a migration tool needs to its structures and its used APIs. As a project typically use thousands of APIs, it is often more difficult to translate used APIs especially for those languages whose structures are similar. In particular, El-Ramly *et al.* [5] conduct an experiment to translate Java programs to C#. One of their learnt lessons is "it becomes very important to develop methods for automatic API transformation". Barry compares C# with Java[5], and claims "although coding in C# is easy for a Java programmer..., the biggest challenge in moving from Java to the .Net Framework is learning the details of another set of class libraries". If not knowing API mapping relations, a migration tool or a programmer cannot translate used APIs correctly. Even when such mapping relations are available, a migration process may introduce defects in translated code since mapped APIs can have different behaviors. As reported by Panesar[6], even most common methods such as `String.subString(int, int)` can have different behaviors between Java and C#. We investigate the mapping relations of existing migration tools, and we confirm that the different behaviors of mapped APIs can cause defects in translated code. In particular, in Java2CSharp[7], one item of mapping files is described in its mapping files as follows:

```
1 package java.lang::System{
2  class java.lang.String :: System:String{
3   method valueOf(Object)  pattern = @1.ToString();
4   ...}}
```

Line 2 of this item describes that the `java.lang.String` class in Java is mapped to the `System.String` class in C#. Line 3 of this item describes that the `java.lang.String.valueOf(Object)` method is mapped to the `System.String.ToString()` method in C#, and `@1` denotes the first parameter of the `valueOf(Object)` method. Based on the preceding mapping relation, Java2CSharp translates a Java code snippet (Lines 5 and 6) into a C# code snippet (Lines 7 and 8) as follows.

```
  Java Code
5 Object obj = ...
6 String value = java.lang.String.valueOf(obj);
  C# Code translated by Java2CSharp
7 Object obj = ...
8 String value = obj.ToString();
```

The translated code snippet compile well, but it has different behaviors with the original Java code snippet. For example, if Line 5 assigns null to `obj`, `value` of Line 6 will be "null". If Line 7 assigns null to `obj`, `value` of Line 8 will not be set to "null" since it throws `NullReferenceException`.

[5] `http://tinyurl.com/26d8xcp`
[6] `http://tinyurl.com/3xpsdtx`
[7] `http://j2cstranslator.sourceforge.net/`

As it throw exceptions, the preceding difference of API mapping relation is relatively easy to detect since programmers often use extreme inputs such as null values as test cases. In particular, sharpen is aware of the differences, and the mapping relation in sharpen is defined as follows:

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

Based on Line 11 of the preceding mapping relation, sharpen translates the Java code snippet (Lines 5 and 6) into a C# code snippet (Lines 7 and 8) as follows.

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

In sharpen, the `getStringValueOf(object)` method is implemented as follows:

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

If Line 13 assigns a null value to `obj`, `value` in Line 14 will also be "null" as expected. By implementing its own mapped C# method, sharpen hides the preceding difference, but it still fails to hide all differences. For example, we find that if Line 5 assigns a `false` boolean value to `obj`, `value` in Line 6 will be "false", but if Lines 7 and 13 assign a `false` boolean value to `obj`, `value` of Line 8 and `value` in Line 14 will both be "False". This difference is relatively difficult to detect, since a programmer typically does not know the internal logic of the method to construct appropriate test cases.

It is desirable to detect differences of API mapping relations since the differences will potentially introduce defects into client codek, the same inputs, but it is challenging to detect different behaviors of API mapping relations via testing for two factors: (1) APIs are typically quite large in size, so it takes great efforts to write test cases manually for APIs and their mapping relations; (2) Other types of migrations such as library migrations [11] can use existing test cases to ensure the quality of migrated code, but for language migration, translated test cases may also have defects at the first place; (3) It requires many test cases to reveal all behaviors of APIs, and simply generating extreme values such as null values are not sufficient to reveal all API behaviors.

In this paper, we propose an approach, called TeMaAPI (**Te**sting **Ma**pping relations of **API**s), that detects different behaviors of API mapping relations via testing. TeMaAPI generates various test cases and compares testing results of mapped APIs for their different behaviors. This paper makes the following major contributions:

- A novel approach, called TeMaAPI, that detect different behaviors of mapped APIs via testing. Given a migration 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.

Static field criterion. Given a static field f of a class C in a language L_1 and its mapped interface $\psi(C.f)$ in a language L_2, a adequate test suite should cover all values of $C.f$ with respect to the static field criterion. This criterion ensures that all values of f in L_1 is correctly mapped to $\psi(C.f)$ in L_2.

Non-static field criterion. Given a non-static field f of a class C in a language L_1 and its mapped interface $\psi(obj.f)$ 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 all states of obj in L_1 return the same values with $\psi(obj.f)$ in L_2.

Static Method criterion. Given a static method $m(T_1\ p_1, \ldots, T_n\ p_n)$ of a class C in a language L_1 and its mapped interface $\psi(C.m(T_1\ p_1, \ldots, T_n\ p_n))$ in a language L_2, a adequate test suite should cover all paths of $m(T_1\ p_1, \ldots, T_n\ p_n)$ with respect to the static method criterion. This criterion ensures that $m(v_1, \ldots, v_n)$ in L_1 returns the same values with $\psi(C.m(v_1, \ldots, v_n\ ))$ in L_2.

Non-static method criterion. Given a non-static method $m(T_1\ p_1, \ldots, T_n\ p_n)$ of a class C in a language L_1 and its mapped interface $\psi(obj.m(T_1\ p_1, \ldots, T_n\ p_n))$ in a language L_2, a adequate test suite should cover all paths of $m(T_1\ p_1, \ldots, T_n\ p_n)$ with respect to the non-static method criterion. This criterion ensures that $obj.m(v_1, \ldots, v_n)$ in L_1 is correctly mapped to $\psi(obj.m(v_1, \ldots, v_n))$ in L_2.

Round-trip criterion. Given two mapped APIs (an API i_1 in a language L_1, an API i_2 in a language L_2, $\psi(i_1) = i_2$, and $\psi^{-1}(i_2) = i_1$), a adequate test suite should be adequate to test both the mapping relation $\psi(i_1) = i_2$ and the mapping relation $\psi^{-1}(i_2) = i_1$.

3. EXAMPLE

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

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

```
public java.lang.String testvalueOf57(Object m0){
  return java.lang.String.valueOf(m0);}
```

After that, we use a migration tool to translate generated client code from Java to C#. In this example, the translated C# client code is as follows:

```
public System.String TestvalueOf57(Object m0) {
return m0.ToString();}
```

The migration tool translate the client-code method in Java into a client-code method in C#, and the translation introduces no compilation errors.

Removing translated client-code methods with compilation errors. A migration tool typically cannot cover all APIs, so some translated client-code methods can have compilation errors if their

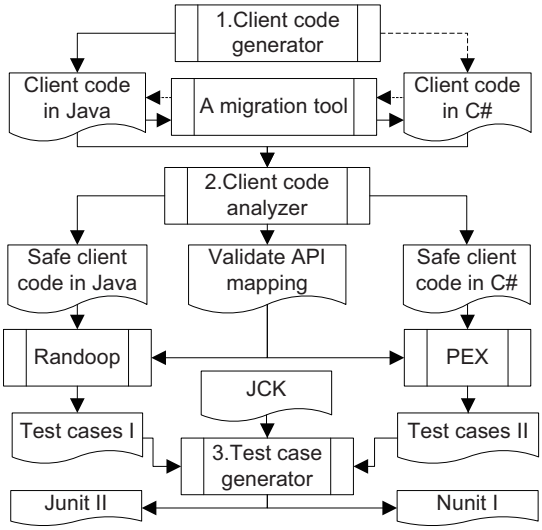

Figure 1: Overview of TeMaAPI

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.

Generating and executing test cases to reveal different behaviors. We leverage various techniques to generate test cases for remaining client-code methods. TeMaAPI targets at generate test cases that can satisfy the round-trip criterion, and comparing outputs of generated test cases for different behaviors given the same inputs. In this example, TeMaAPI detects that when an object is assigned to a null value, a boolean value, or a double value, the outputs are different.

After the preceding steps, TeMaAPI detects the different behaviors between `java.lang.String.valueOf(Object)` in Java and its mapped C# API. We next present details of preceding steps.

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 1 shows the overview of TeMaAPI.

4.1 Generating 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 [9] 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 1, 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 1. When TeMaAPI generates client code in C#, it ignores `unsafe` and `delegate` methods and methods whose parameters are marked as `our` or `ref`. 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.

4.2 Analyzing Generated Methods

Translated code typically contain many compilation errors since a migration tool typically cannot cover mapping relations of all APIs. TeMaAPI then analyzes translated code for validate API mapping relations of the migration tool. To achieve this, TeMaAPI first remove all translated methods with compilation errors. For translated methods in Java, TeMaAPI implements a Eclipse plug-in that uses on Eclipse JDT compiler[8] for the list of compilation errors. For translated methods in C#, TeMaAPI implements a Visual Studio.Net add-in to retrieve the list of compilation errors from the error-list view of Visual Studio.Net. Both Eclipse JDT compiler and Visual Studio.Net cannot list all methods with compilation errors in a single build. After each iteration of removing methods,

[8] `http://www.eclipse.org/jdt/`

TeMaAPI re-build these methods until it removes all methods with compilation errors.

After methods with compilation errors are removed, TeMaAPI compares generated code with translated code for the validate API mapping relations of a migration tool. Based on translated code and validate API mapping, TeMaAPI removes generated methods whose corresponding translated methods have compilation errors. We refer to those removing client-code methods as safe methods.

4.3 Finding Different Behaviors

In the final step, TeMaAPI generates test cases to detect different behaviors of API mapping relations. An alternative approach is to use existing test cases in two languages. For example, lucene[9] has both a Java version and a C# version. It is feasible to use these test cases to reveal some different behaviors, but such test cases typically cover only a small set of APIs. Some test suites such as Java Compatibility Kit (JCK)[10] cover most APIs of a language. However, translating such a test suite from one language into another language may introduce many compilation errors and defects. A test method may use many APIs, so even if the API under test can be translated correctly, the test method cannot be translated correctly since other APIs are not mapped. As a result, we choose to translating existing test suites as a supplement of our approach.

4.3.1 Generating Test Cases

For each safe method in Java, we use Randoop [12] to generate its test cases. For each safe method in C#, we use Pex [18] to generate its test cases. TeMaAPI then executes generated test cases, and records the inputs, the output, and the thrown exception of each test case as a file.

Based on the file, TeMaAPI generates Junit[11] or Nunit[12] test cases to ensure each mapped API produce the same output give the same inputs. For example, Pex generates a test case whose input is `m0 = false` for the `TestvalueOf57` method in C# as shown in Section 3, and after executing the output of the test case is "False". Based on the input and the output of this test case, TeMaAPI generates a Junit test case as follows:

```
@Test
public void testvalueOf64zhh0(){
  sketch.Test_java_lang_String obj =
                    new sketch.Test_java_lang_String();
  boolean m0 = false;
  Assert.assertEquals("False", obj.testvalueOf64(m0));}
```

This Junit test case fails since the preceding `testvalueOf64zhh` method produces "false" instead of "False". From this failed Junit test case, TeMaAPI detects that the `java.lang.String.valueOf(Object)` method in Java has different behaviors with its mapped C# methods if inputs are boolean values.

In some cases, executing a test case does not produce outputs but exceptions. For example, Pex also generates a test case whose input is `m0 = null` for the `TestvalueOf57` method in C# as shown in Section 3, after executing it throws `NullReferenceException`. TeMaAPI finds that the `NullPointerException` class in C# is mapped to the `NullPointerException` class in Java in the validate API mapping relations, and generates a Junit test case based on the preceding mapping relation and input as follows:

```
@Test
public void testvalueOf64zhh3(){
try{
  sketch.Test_java_lang_String obj =
                    new sketch.Test_java_lang_String();
  boolean m0 = null;
  obj.testvalueOf64(m0);}
}catch(java.lang.NullPointerException e){
    Assert.assertTrue(true);
    return;
}
Assert.assertTrue(false);}
```

This Junit test case also fails since given a null input, the preceding `testvalueOf64` method does not throw any exceptions. From this failed Junit test case, TeMaAPI detects that the `java.lang.String.valueOf(Object)` method in Java has different behaviors with its mapped C# methods if inputs are null pointers.

4.3.2 Translating Existing Test Cases

Each generated client-code method uses only one fields or methods provided by API libraries, and may lose some complicated behaviors even if test cases satisfy the round-trip criterion. To test those complicated behaviors, we introduce JCK that covers many complicated behaviors of Java APIs. JCK is a test suite provided by Sun to ensure compatibility of Java platforms, and it covers most standard APIs of J2SE. However, JCK implements many internal classes to collect the results of executed test cases. If a migration tool cannot correctly translate one of these classes, all translated test cases may have compilation errors or defects. In addition, JCK is released under read-only source license[13], so many such internal classes are not shipped and it has many compilation errors. To increase the chance of migrating JCK, TeMaAPI first replaces those internal classes with the classes of Junit. For example, one test method for `java.io.File.delete()` in JCK is as follows:

```
public Status File0037(){
  String testCaseID = "File0037";
  ...
  FileRT method = new FileRT(testCaseID) {
   public Status run() {
     File f = null;
     f = new File(workdir, testCaseID);
     ...
     if (f.delete()) { // Try to delete
       if (!f.exists()) { // Does it exist?
         return Status.passed("OKAY");
       }else{
         return Status.failed(...);
       }
     else{
         return Status.failed(...);
     }
  }
   return AllPermissionSM.testRun(...);
}
```

After the preceding three steps, TeMaAPI further replaces the statement starts with `FileRT` with the body of the `run` method, and removes the last statement. The translated code is as follows:

```
public void File0037(){
  String testCaseID = "File0037";
  ...
  File f = null;
  f = new File(workdir, testCaseID);
  ...
  if (f.delete()) { // Try to delete
    if (!f.exists()) { // Does it exist?
      Assert.assertTrue(true);
      return;
   }else{
      Assert.fail();
      return;
```

[9] http://lucene.apache.org
[10] http://jck.dev.java.net
[11] http://www.junit.org/
[12] http://www.nunit.org/
[13] http://tinyurl.com/33x9fo6

Tool	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: Subjects

```
    }
  else{
      Assert.fail();
      return;
  }
 }
```

Compared with the original test method in JCK, the translated method does not use the three internal classes: `Status`, `FileRT`, and `AllPermissionSM`.

After the preceding process, for a migration tool, TeMaAPI further removes methods that use any APIs outside its defined mapping relations. The remaining methods can be translated from Java to other languages since it does not use any APIs outside of the migration tool.

5. EVALUATIONS

We implemented a tool for TeMaAPI and conducted two evaluations using our tool to show the effectiveness of our approach. In our evaluations, we address the following two research questions:

1. How effectively can our approach detect different behaviors of API mapping relations for migration tools that translate code from Java to CSharp (Section 5.1)?
2. What are the impacts of found different behaviors on migration tools (Section 5.2)?

Table 1 shows the migration tools used as subjects in our evaluation. Among these tools, JLCA and VB & C# to Java Converter are commercial tools, and other three tools are all open source tools. All evaluations were conducted on a PC with Intel Qual CPU @ 2.83GHz and 1.98M memory running Windows XP.

5.1 Detecting different behaviors

5.2 Analyzing results

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 [13], researchers proposed various approaches [7, 10, 19, 20, 22] to automate the process. However, all these approaches focus on the syntax or structural differences between languages. Deursen *et al.* [19] proposed an approach to identify objects in legacy code. Their approach uses these objects to deal with the differences between object-oriented and procedural languages. As shown in El-Ramly *et al.* [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 [8] proposed an approach that captures and replays API refactoring actions to update the client code. Xing and Stroulia [21] 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, 16, 17, 23, 24] 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] C. Garcia. *Compose* A Runtime for the. Net Platform*. PhD thesis, University of Twente, 2003.

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

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

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

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

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

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

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

[14] R. Sebesta. *Concepts of programming languages*. Addison-Wesley New York, 2002.

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

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

[17] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proc. 7th ESEC/FSE*, pages 193–202, 2009.

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

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

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

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

[22] 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.

[23] 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.

[24] 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.