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

Migration tools can help translate projects from one language into another language. For example, as stated by Patrick Roemer[1], a migration tool named sharpen translates most of db4o[2] engine core code and its unit test suites from Java into C#. As existing projects typically use many Application Programming Interfaces (APIs), a migration tool should know the mapping relations of APIs between two languages, so that it can translate projects with API callsites correctly. However, two mapped APIs may have different behaviors, and thus can introduce defects in translated client code silently. For example, in Java2CSharp[3], one mapping relation of APIs 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   ...}}
```

[1] http://tinyurl.com/29kw78e
[2] http://www.db4o.com/
[3] http://j2cstranslator.sourceforge.net/

Based on Line 3 of 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`.

The developers of sharpen are 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, Java2CSharp 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)` is implemented as follows:

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

If Line 13 assigns null to `obj`, `value` will also be "null". Still, we find that the mapping relation defined by sharpen does not hide all differences between the `java.lang.String.valueOf(Object)` method and its mapped C# API. For example, if Line 5 assign a `false` boolean value to `obj`, `value` of Line 6 will be "false", but if Lines 7 and 13 assign a `false` boolean value to `obj`, `value` of Line 8 and `value` of Line 14 will both be "False". This difference will potentially introduce defects into client code since it produces different outputs given the same inputs.

It is challenging to detect different behaviors of API mapping relations via testing for two factors: (1) as shown in our previous work [18], translated code often cannot be tested directly since it typically contains compilation errors; (2) it requires techniques to generate adequate test cases for APIs whose code is often not

available. 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. 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. The different behaviors can introduce defects in translated code silently.
- Test adequacy criteria proposed for generating sufficient test cases to test API mapping. For the testing criteria, TeMaAPI introduces a feedback loop when it generates test cases .
- 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.* [2], 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 reveal 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 reveal 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 reveal 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 reveal 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:

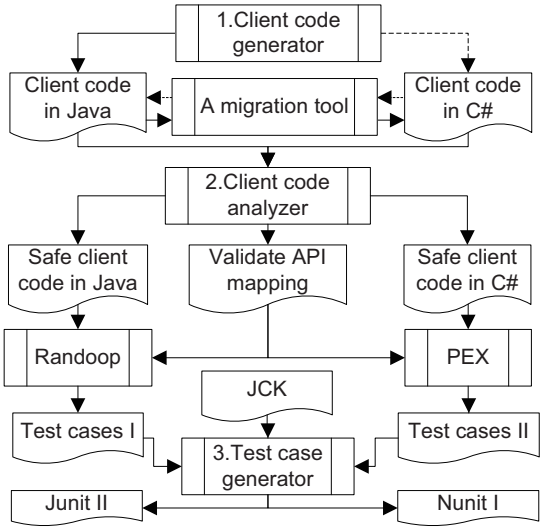

Figure 1: Overview of TeMaAPI

```
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 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. If we use a migration tool to translate existing projects, many API mapping relations may be not covered, and it may be difficult to analyze the translated code for validate mapping relations. For the preceding consideration, TeMaAPI chooses to generate client code instead of using existing client code.

TeMaAPI relies on the reflection technique [7] 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[4] 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, 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 Generating and Executing Testing Cases

In the final step, TeMaAPI generates test cases to detect different behaviors of API mapping relations. For each safe method in Java, we use Randoop [9] to generate its test cases. For each safe method in C#, we use Pex [13] 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[5] or Nunit[6] 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;
```

[4] http://www.eclipse.org/jdt/
[5] http://www.junit.org/
[6] http://www.nunit.org/

```
  }
  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.

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 Java Compatibility Kit (JCK)[7] 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 all APIs of J2SE. As JCK is released under read-only source license[8], we cannot generates Nunit test cases based on inputs and outputs by executing JCK. Instead, TeMaAPI reads source files of JCK, and translates API related test cases into Junit. For example, a test method for `java.lang.Strng.endWith(String)` in JCK is as follows:

```
public Status String0044(){
  String testCaseID = "String0044";
  String s1 = "endsWith Test"; //step Create a String
  String s2 = " Test";         //step Create a suffix
  ref.println("s1 = " + s1);
  ref.println("s2 = " + s2);
  if( s1.endsWith(s2) )        //step check result
  {
     return Status.passed( "OKAY" );
  }
     return Status.failed( testCaseID
                           + " copyValueOf failed" );
  }}
```

Based on the test method of JCK, TeMaAPI generates a Junit method case as follows:

```
@Test
public void String0044(){
  String testCaseID = "String0044";
  String s1 = "endsWith Test"; //step Create a String
  String s2 = " Test";         //step Create a suffix
  System.out.println("s1 = " + s1);
  System.out.println("s2 = " + s2);
  if( s1.endsWith(s2) )        //step check result
  {
      Assert.assertTrue(true);
      return;
  }
     Assert.fail();
     return;
  }}
```

JCK implements its own classes to assert whether a test case passes. To translate the preceding method of JCK into a Junit method, TeMaAPI follows the following steps:

Step 1: adding the `@Test` annotation to the method.

Step 2: the return type of the method `Status` → `void`.

Step 3: `return Status.passed(...)` → `Assert.assertTure(true); return;` and `return Status.failed(...)` → `Assert.fail(); return;`.

Some test methods of JCK are so straightforward as the previous one. 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;
   }
 else{
     Assert.fail();
     return;
 }
}
```

Compared with the original test method in JCK, the translated method does not use the `FileRT` class and the `AllPermissionSM` class. After the preceding process, TeMaAPI removes those test methods who use APIs out of the validate API mapping relations. As the `FileRT` class and `AllPermissionSM` class are both defined in JCK, their mapping relations are unlikely in the validate API mapping relations. Removing them increases the chance to translate a Junit test case into a Nunit test case. The remaining test cases can be translate by a migration tool from Java to C# since their used APIs are all in the validate mapping relations 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. How effectively can our approach detect different behaviors of API mapping relations for migration tools that translate code from CSharp to Java (Section 5.2)?

In our evaluations, we introduce Java2CSharp and C# to Java converter[9] as subjects. We choose Java2Charp since it is a relatively mature tool developed by ILOG[10] (now part of IBM). We choose C# to Java converter since it is a commercial tool developed by Tangible Software[11].

[7] `https://jck.dev.java.net/`
[8] `http://tinyurl.com/33x9fo6`
[9] `http://tinyurl.com/24o6jpc`
[10] `http://www.ilog.com/`
[11] `http://www.tangiblesoftwaresolutions.com`

5.1 From Java to CSharp

5.1.1 Generating client code

5.1.2 Analyzing client code

5.1.3 Detecting different behaviors

5.2 From CSharp to Java

5.2.1 Generating client code

5.2.2 Analyzing client code

5.2.3 Detecting 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 [10], researchers proposed various approaches [5, 8, 14, 15, 17] to automate the process. However, all these approaches focus on the syntax or structural differences between languages. Deursen *et al.* [14] 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.* [4]'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 [6] proposed an approach that captures and replays API refactoring actions to update the client code. Xing and Stroulia [16] proposed an approach that recognizes the changes of APIs by comparing the differences between two versions of libraries. Balaban *et al.* [3] 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, 11, 12, 19, 20] 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] 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.

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

[4] M. El-Ramly, R. Eltayeb, and H. Alla. An experiment in automatic conversion of legacy Java programs to C#. In *Proc. AICCSA*, pages 1037–1045, 2006.

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

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

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

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

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

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

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

[12] 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.

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

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.