Evaluation and Criticism on Black-Box Testing Based Identity Verification of Java and Kotlin Code

20140847 Gunou Park, 20150348 Jihoon Park, 20150645 Sungmin Im, 20170360 Ukho Shin

June 22, 2021

Abstract

Since its emergence, Kotlin has become a very popular language widely used across various fields. It is widely used and tends to replace Java code in many areas like Android application development. Due to its full interoperability with Java and therefore ease for substituting Java, migration from Java to Kotlin happens very frequently nowadays. Therefore, we need a way to ensure semantic identity of the original Java code and migrated Kotlin code. In this report, we suggest a methodology of using black-box testing for comparing generated test suite outputs to confirm the code identity, and provide criticism on the methodology. We have implemented a program to simplify the process, and used it to evaluate our methodology.

1 Introduction

Kotlin is a programming language developed by JetBrains in 2011. It is designed to be fully interoperable with Java, so there is almost no work needed to call Java method or class from Kotlin file, and vice versa. It has become very popular due to its convenience on industrial level coding and its supplement on Java language. As Kotlin became Google's preferred language for Android development, it has become one of the most fast growing language in the world.

Due to its interoperability with Java, it is very easy to migrate an existing Java project to Kotlin. One can convert each file in a project one by one from Java to Kotlin, not the whole project. Therefore, it is very common in the industry where they have both Java and Kotlin files in their project.

Although most developers may simply convert Java class to Kotlin class, it becomes very crucial to ensure that the migrated Kotlin class works exactly the same as Java class. This is very important especially if they are migrating a project in service, because the migration could cause bugs if not carefully done. Many companies verify this process by code reviews, but it becomes really difficult to detect the difference in files written in two different languages, especially if the file is very big.

For that, we are suggesting a methodology to ensure semantic identity of original Java code and migrated Kotlin code, using black-box testing for comparing generated test suite outputs of each code to confirm the code identity. We implemented a program to simplify the process. It will generate test suites which can cover original Java code, and execute tests on both Java and Kotlin class to compare those outputs as black-box testing. By evaluating the result in various cases, we are going to evaluate and criticize on the methodology.

2 Experiments

We use Evosuite to generate test suites. Evosuite is an automatic test suite generation tool for Java. It generates optimized JUnit tests covering different coverage criteria, like lines, branches, outputs, and mutation testing from the given Java class file. The tests are in the form of Java code, but it can be run over Kotlin class too. The generated JUnit tests always pass on the original Java class, we only need to consider the result running on the Kotlin class.

If all tests are passed on the Kotlin class, it means two codes are identical or there's only negligible difference between. Or if a failure exists, it means there is a difference between migrated Kotlin code and original Java code so the migration was unsuccessful and should be rectified. The program parses the execution output of Evosuite to simplify the process.

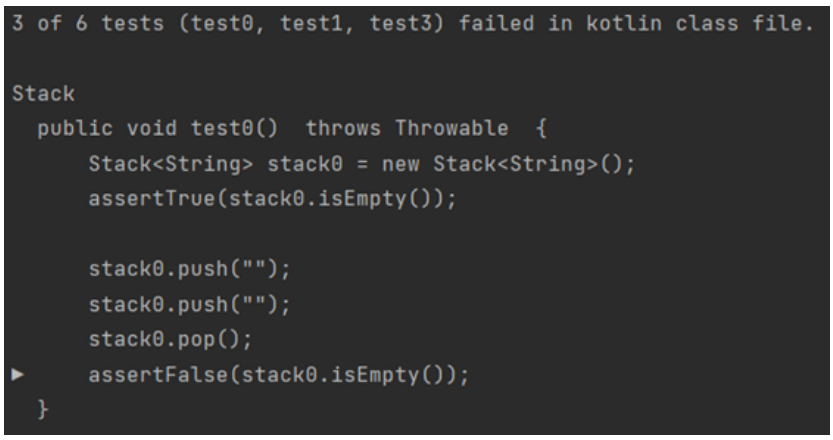

Figure 1: Execution example. The result shows how many tests have failed and which assert caused the failure.

If there's a failure in migration, the program will show the number of failed tests and which tests were failed. Also, it would annotate the failed lines of the tests for debugging, making it easy to detect which part of the Kotlin code is mutated. For example in Figure1, we can get the error occurred in isEmpty() function after 2 push and 1 pop. We can modify the code based on the information and can re-run the program to check whether the modification works correctly, repeating until all test-suites are passed.

3 Evaluation

As described, our method seeks to find and pinpoint differences in existing Java code and newly written Kotlin code by using the same test cases written in Java with JUnit. Therefore, the validity and effectiveness of our method can be evaluated by two major criteria, running the code successfully, and detecting differences with generated test cases.

3.1 Basic flow and conditionals

Before evaluating the effectiveness of our method with domain specific features and characteristics, the fundamental basis of our method, testability, had to be verified. We established the basis of our methods by executing our method with source code containing simple logical flow and conditionals. Cases such as the Circle series and the Person series (1, 2) pass test cases for both Java and Kotlin for logical flow, proving that simple classes with method and constructor calls can be tested well under this scheme. A little more complex programs with sorting algorithms and data structure implementations, such as MaxHeap and MergeSort cases, have also proven to work seamlessly. We verified that conditional mutants are killed, and also some obvious cases that fail from JUnit where the symbols such as method names do not match.

Library compatibility can also be tested through our method. As Java and Kotlin all run under the JVM as compiled bytecode, they can reference the same libraries regardless of the base language the library is written in, but may eventually use different libraries that are more native to the specific language when programmed. As verified with the Circle test cases, our method can also test if the referenced libraries execute and provide the same results by evaluating the results of the functions.

3.2 Cross-language inheritance

Language migration, especially for a large code base, are often done incrementally. With object oriented implementations such as inheritance and method overloading, it is crucial to be able to convert only a part of the code in the inheritance chain. The IMAPClient series employs the Apache Commons' Network source, where there are inheritances between the classes. It was possible to convert only the IMAPClient class into Kotlin while keeping the inheritances with classes written in Java, and have the

(a)

```java
public class Circle {
    private final int radius;

    public Circle(int radius) {
        if (radius ⩽ 0) throw new IllegalArgumentException("Radius must be bigger than 0.");
        this.radius = radius;
    }

}
```

(b)

```kotlin
class Circle(private val radius: Int) {

    init {
        require(radius > 0) { "Radius must be bigger than 0." }
    }
}
```

Figure 2: (a) Java code with explicit IllegalArgumentExceptoin (b) Kotlin syntactic sugar for the same exception

functionalities of the code tested. There may be some discrepancies with the scoping of members as the philosophy behind Kotlin is definitely different to that of Java, but this will be further discussed later.

3.3 Unit testing of string payloads

When testing for functionalities of programs with unit tests, we often also check the formats of string payloads, if they matter. However, with our method we discovered that it is not possible to generate test cases that can check for formats of string payloads. The User series proved that our method fails to detect the native function ‘toString()‘ produces differently formatted string payloads in Java and Kotlin, marking the tests successful when they are not semantically the same. The second case of the IMAPClient series also showed the same. A mutation in the format of the string payload was not killed by the generated test cases of our project. Therefore, our approach cannot cover the semantics of programs and must be supplemented by unit tests for such purposes. However, as we have verified that it is possible to run Java test cases with JUnit on Kotlin code, the existing Java test cases can be used to test the semantics without having them converted into Kotlin.

3.4 Kotlin-specific Language Features

Although both Java and Kotlin run on JVM and are interoperable, there are still some syntactic differences between the two languages that hinders the effectiveness of our methodology. Here are some of the examples that we have identified during the process.

Syntactic Sugar

Kotlin provides a vast amount of syntactic sugar thanks to its degree of freedom. We have analyzed if our test suite works for various syntactic sugars. This would be an evaluation for cases where the code does not look the same, but is intended to work the same way. For instance, as shown in Figure 2, Kotlin’s “require” syntax throws IllegalArgumentException if the value is false. Therefore, following two code snippets work the same. We have also confirmed that all of our test suites work the same on the syntactic sugars.

Data Class

Kotlin data class is used for cases when classes are mainly used to hold data. This frequently happens when we have to handle entities in our application. The compiler automatically derives

(a)

```java
public class User01 {

    private String email;

    private String name;

    private String password;

    private Integer age;

    private Double height;

    // getter, setter

    public User01(String email, String name, String password, Integer age, Double height) {
        this.email = email;
        this.name = name;
        this.password = password;
        this.age = age;
        this.height = height;
    }
}
```

(b)

```kotlin
data class User01(
    var email: String? = null,
    var name: String? = null,
    var password: String? = null,
    var age: Int? = null,
    var height: Double? = null
)
```

Figure 3: (a) Java code for User entity class (b) Same User entity class in Kotlin data class

equals(), hashCode(), toString(), and so on from all properties declared in the primary constructor. For example, Figure 3 shows how entity class in Java can be written in Kotlin.

They are basically the same class, so our test cases work the same for both classes. However, there are some differences due to how Kotlin treats data classes. equals() method in Kotlin data class actually compares the content of the data class, while the original Java class compares whether they have the same reference. Also, Java toString() by default prints out the reference address of the class, while Kotlin prettifies the content of the data class and prints out the actual content of the class. Therefore, although not shown explicitly, the result of calling equals() or toString() may be different when they are called from Java or Kotlin class. Such differences, also shown on the String payloads section of our evaluation, are not properly caught by our test suites.

Object declaration

Object declaration is a Kotlin feature that makes it easy to declare singletons. It is very common to use singleton pattern or utility classes in Java, so Kotlin object could be an alternative to such patterns. As shown on Figure 4, ArraySorter util class could be migrated into object class.

However, since it is still possible to instantiate ArraySorter class, the test suite for constructor would not work in Kotlin object, where instantiation is forbidden. This could be solved by declaring a companion object inside Kotlin class, but this would be unnecessary. A better solution would be making the constructor of Java class private, so that instantiation outside the class is not possible.

Nullable Syntax

In Java, accessing a member of a null reference will result in a null reference exception. Although Kotlin does have a null pointer exception(NPE), the language aims to eliminate such exceptions through its language features. The only cases where NPE happens in Kotlin code is when NPE is explicitly thrown, usage of !! operator, some data inconsistency during initialization, and due to Java interoperation. Kotlin defends the reference of null variables by means of ?(elvis operator) syntax. It

(a)

```java
public class ArraySorter {

    public static byte[] sort(final byte[] array) {
        Arrays.sort(array);
        return array;
    }

    public static int[] sort(final int[] array) {
        Arrays.sort(array);
        return array;
    }

    // ...

    public static <T> T[] sort(final T[] array, final
Comparator<? super T> comparator) {
        Arrays.sort(array, comparator);
        return array;
    }

}
```

(b)

```kotlin
object ArraySorter {

    @JvmStatic
    fun sort(array: ByteArray?): ByteArray? {
        Arrays.sort(array)
        return array
    }

    @JvmStatic
    fun sort(array: IntArray?): IntArray? {
        Arrays.sort(array)
        return array
    }

    @JvmStatic
    fun <T> sort(array: Array<T>?, comparator:
Comparator<in T>?): Array<T>? {
        Arrays.sort(array, comparator)
        return array
    }

}
```

Figure 4: (a) Java code for util class (b) Kotlin code for singleton util class

indicates that such variables are nullable, and only these variables can store null value. If you attempt to store null value in such variables, the Kotlin runtime will throw an exception that tells you that there is null value in a non-null variable. This leads to differences in how the test cases work with both Java and Kotlin classes.

As shown in Figure 5, Java code by default does not check for null-safety of its input parameter. Therefore, if the code in the method accesses the variable without checking for null-safety, it is very likely that it throws NPE. Our test suite checks for such misbehaviors. However in Kotlin, although it looks very similar to Java code, the variable bar does not accept null values. Therefore, the exception will be thrown on the method call time, not when the variable is actually used. This leads to a difference in exception between Java and Kotlin code. To have the same exception as Java, Kotlin should use not-null assertion operator(!!) to explicitly throw NPE when null value is given.

4 Conclusion

We have built an application that tests the identity of two codes with differential testing by auto generated test code. We've tested especially for Java and Kotlin, which occur quite frequently these days.

This approach was very effective for the aspect of time-saving, and effort-saving. With our project, all users have to do is execute our code. It helps people who want to test the identity of migrated functions. Because of buggy Java-Kotlin converter, our application should be useful for people who want to convert between Java and Kotlin.

However, there were certain limitations on our work. First, we should keep our classes interface. And, this test doesn't know reasonable changes like the destination language's philosophical choices. And last, it is less persuasive because we just used evosuite which used SBST technique. If we implemented a test code generator for differential testing, we can more rigidly argue for the identity of two codes.

5 Future Work

Expand into other JVM Languages

We can expand our works to other JVM Languages such as Groovy, Scala, Clojure, etc. Because evosuite generate testing JAVA file with class file, so we can generate regardless of source language. So we can choose two languages in JVM Langauges and use our application as stated above.

(a)

```java
public String foo(String bar) {
    return bar.toUppercase() // if bar == null, NullPointerException!
}
```

(b)

```kotlin
fun foo(bar: String): String {  // if bar == null, exception here!
    return bar.toUppercase()
}
```

(c)

```kotlin
fun foo(bar: String?): String? {
    return bar?.toUppercase()  // if bar == null, returns null
}
```

Figure 5: (a) Java method that throws exception when null is given (b) Kotlin method that does not accept null parameter (c) Kotlin method that throws exception when null is given

Use other test code generator

In our project, we used evosuite as a test code generator. It is very powerful and intelligent tool. But evosuite is not perfect option for every code. As we seen before, there are some points that evosuite is not covering. To cover that points, we can expand by using other test code generator. There are several test code generators in industry. We can use other test code generator like randoop or squareTest etc. By replacing or adding test code generator, we can cover more cases and enhance our performances.

Fault Localization

If two codes are behaving differently at some point, and if we can detect them by our test case, we can try fault localization. By adding fault localization, our user can fix some differences between two codes easily.

6 Project Repository

https://github.com/min0623/kotlin-java-identifier

7 Code References

We used or referenced a number of existing open source codes available on the web, in the input directory of our repository, to evaluate the effectiveness of our test suite generator. We claim no rights to any of the source codes we coined for evaluating the effectiveness of the test suite generator. The following is a list of all the sources we referenced to construct the evaluating set.

- Packt Publishing, Java Programming for Beginners
- Apache Commons Net

- Apache Commons Lang
- TutorialsPoint
- w3schools
- geeksforgeeks