Migrating Unit Tests Across Java Applications

Ajay Kumar Jha
North Dakota State University
Fargo, USA
ajay.jha.1@ndsu.edu

Sarah Nadi
New York University Abu Dhabi and University of Alberta
Abu Dhabi, United Arab Emirates
sarah.nadi@nyu.edu

***Abstract*—Writing effective unit tests is often tedious, difficult, and time-consuming. Test recommendation techniques facilitate this process by recommending existing manually written tests from other similar systems for developers to reuse. However, developers still have to put non-trivial effort into modifying the recommended tests. For example, they have to understand various code elements in the recommended tests to accurately replace them with semantically similar code elements from the target system. In this paper, we propose JTESTMIGRATOR, a technique to automatically migrate unit tests between semantically similar methods across applications. Given a source and a target method with similar functionality across applications, where the source method has some unit tests, JTESTMIGRATOR migrates unit tests by transforming the test code. JTESTMIGRATOR uses semantic similarity and type compatibility of code elements in the source and target systems to transform test code. We implement JTESTMIGRATOR to migrate JUnit tests and evaluate it on 104 tests for 42 methods across 5 popular libraries. JTESTMIGRATOR successfully migrates 76 (73%) of the tests for 32 (76%) of the methods across the 5 libraries. 10 (13%) of the successfully migrated tests increase the code coverage of 4 target methods.**

***Index Terms*—unit testing, test reuse, test migration, code transformation**

I. INTRODUCTION

Unit testing is one of the popular forms of testing [1], [2]. Unit tests have several advantages: assuring that the system under test (SUT) works as intended [3], improving developers' confidence in changing the code of the SUT [4], [5], reducing bug-fixing costs by detecting bugs earlier [5]–[8], and acting as documentation [5], [9]. Despite these advantages, some developers still do not write unit tests due to, for example, time pressure and a lack of recognition from management [10], [11]. Additionally, writing effective unit tests can take a significant amount of time [10], which can reduce developers' productivity [4], [10] and increase development costs [8].

Test-generation tools, such as Randoop [12], JCrasher [13], and EvoSuite [14], help developers create tests. These tools use a random or search-based approach to generate tests. However, these techniques still struggle to generate effective test inputs and oracles [15]–[17], which are key components of unit tests. Another major shortcoming of generated tests is their poor readability [15], [18], [19], which can create maintenance issues down the line [10], [20]. These shortcomings result in the limited adoption of test generation techniques in practice [21]. On the other hand, studies show that manually written tests are more readable and have effective inputs and oracles [15], [18]. Thus, reusing manually written tests can potentially mitigate some of the problems of automated test generation techniques.

One way to reuse existing tests is to find semantically similar functionality in a different software system and recommend the associated tests to developers for reuse [22]–[24]. The developers can then manually adapt these recommended tests and integrate them into their applications. Sondhi et al. [22] found that test inputs and oracles in unit tests can be reused with some adaptations such as converting data types. This implies that developers might still have to spend a non-trivial amount of time in manually reusing the recommended tests.

Manual test reuse might actually take more time than writing new tests because developers have to understand different code elements in the tests that need to be replaced with semantically similar code elements from a target application. For example, a test might contain various class constructors and method invocations; developers need to understand the functionality of such invocations in order to replace them correctly. Makady and Walker [25] performed an experiment of manual test reuse and found all the participants either gave up before their test reuse task was complete or hit the time limit. The experiment was conducted on the modified version of the same application. Test reuse in different applications might be even more challenging in terms of the effort required to find and replace similar code elements. Therefore, a tool that can automatically transform test code elements might help developers reduce the effort needed to reuse tests [26], [27].

Reusing tests by automatically transforming test code is, however, a non-trivial task due to various challenges involved in test code transformation [26], [28], such as finding similar software components, adjusting data types, and adjusting inputs according to different number and order of parameters. Researchers have proposed semi-automated tools to transform test code for test reuse [25], [26]. However, the proposed approaches still require prerequisite manual effort in creating test templates [26] or making a test reuse plan [25]. The applicability of the approaches is also limited in terms of the considered types of assertions, code transformations, and target systems. For example, the `Metallicus` tool proposed by Sondhi et al. [26] populates test templates provided by users with only `assertEquals` assertions. Moreover, the tools and datasets of the approach proposed by Makady and Walker [25] are not publicly available.

In this paper, we propose JTESTMIGRATOR, a technique and tool that automatically transforms various test code elements for test reuse. Given a source and a target method with similar functionality across applications, JTESTMIGRATOR uses a traceability technique to identify tests for the source method, migrates the identified tests to the target applica-

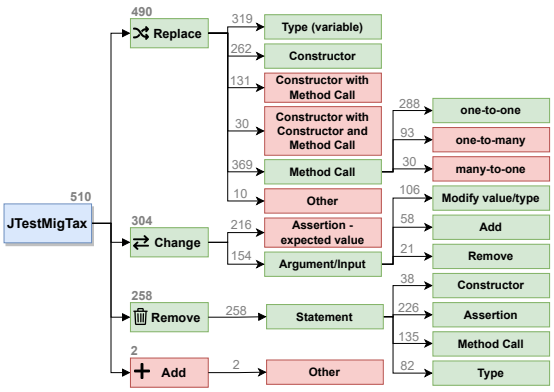

Fig. 1: JTESTMIGTAX, a taxonomy of migration-related code changes [28]. JTESTMIGRATOR supports the green-colored code changes.

tion, and systematically replaces the code in the migrated tests with semantic similar code from the target application. JTESTMIGRATOR automatically reuses manually written tests across similar applications by eliminating the manual effort required in code transformation. We evaluate the effectiveness of JTESTMIGRATOR by migrating 104 JUnit tests for 42 methods across 5 popular JSON and common utility libraries from JTESTMIGBENCH, a dataset of manually reused JUnit tests [28]. JTESTMIGRATOR successfully migrates 76 (73%) of the tests for 32 of the methods across the 5 libraries. The 76 migrated tests are passing tests and they can be directly used in the target libraries to test target methods. 13% of the successfully migrated tests increase the code coverage of 4 target methods, showing some practical values of the test migration concept. Moreover, 87% of the successfully migrated tests are exactly the same as the corresponding human migrated tests in JTESTMIGBENCH, which shows JTESTMIGRATOR could have saved human effort.

Our implementation, data, and results are available on our artifact page [29].

II. BACKGROUND

JTESTMIGRATOR is inspired by observations of our previous study that identify code transformation types necessary to migrate JUnit tests across applications [28]. We next describe the details of this previous study and a motivating example.

A. Types of Code Transformations in Unit Test Migration

To identify the types of code transformations required to successfully migrate JUnit tests across applications, we first created JTESTMIGBENCH, a benchmark of 510 manually migrated JUnit tests for 186 methods from 5 popular libraries. We then analyzed the code changes in the migrated tests to create JTESTMIGTAX, a taxonomy of test code transformation patterns, shown in Figure 1.

The taxonomy shows that potential unit test migration techniques mainly need capabilities to replace, change, and remove various types of program elements in the test code. Specifically, they need to replace types, constructors, and method calls, change method arguments and expected outputs, and remove types, constructors, method calls, and assertions that are not needed. The taxonomy also shows the frequency of these actions in JTESTMIGBENCH. For example, 319 tests out of 510 migrated tests required type replacement.

Our approach in Section III is inspired by the above observations. JTESTMIGRATOR supports the code changes from JTESTMIGTAX that are highlighted in green in Figure 1.

B. Motivating Example

Figure 2a shows class `JsonWriter` from gson [30]. The class declares method `value` to write a value to a JSON array. Figure 2c shows the corresponding JUnit test class, `JsonWriterTest`. Figure 2b shows class `JSONWriter`, from a different library fastjson [31], which declares method `writeValue` to write a value to a JSON array.

Writing effective unit tests, similar to the one shown in Figure 2c, is often a tedious, difficult, and resource-consuming task for developers [12]. The test in Figure 2c uses different inputs to exercise the `value` method and checks actual output against a structured expected output. Developers find that determining oracles and finding relevant inputs are some of the most difficult tasks [10]. Given the common functionality between methods `value` and `writeValue`, a tool that can automatically migrate the test shown in Figure 2c to fastjson in order to test `writeValue` can save developers valuable time. The migrated test can also improve developers' confidence in asserting the intended behavior of the method [5], [32].

JTESTMIGRATOR can automatically migrate unit tests between applications with similar functionality. Figure 2d shows the test class that JTESTMIGRATOR produces for method `writeValue` from Figure 2b, by migrating the original test `testDoubles` from gson, shown in Figure 2c. In the process, JTESTMIGRATOR finds and replaces semantically similar methods (e.g., `beginArray` → `startArray`) and types (e.g., `JsonWriter` → `JSONWriter`).

III. APPROACH

Figure 3 shows a high-level overview of our approach. JTESTMIGRATOR takes two inputs: (1) a *source application*, which is an application that already contains tests and (2) a *target application*, which is an application that we want to add tests for. Specifically, for the target application, JTESTMIGRATOR takes the name of the method that needs to be tested (*target method*) and the name of the class that declares the method (*target class*). The target method and target class for our motivating example are `writeValue` and `JSONWriter`, respectively. For the source application, JTESTMIGRATOR takes the name of the method that performs the same function as the target method (*source method*), the name of the class that declares the method (*source class*), and the name of the test class that tests the method (*source test class*). The source method, source class, and source test class for our motivating example are `value`, `JsonWriter`, and `JsonWriterTest`, respectively. JTESTMIGRATOR optionally takes the name of the test(s) from the source test class that needs to be migrated, which is *testDoubles* for our motivating

```
1  public class JsonWriter implements Closeable, Flushable {
2    ---
3    public JsonWriter value(double value) throws IOException {
4      writeDeferredName();
5      if(!lenient && (Double.isNan(value) || Double.isInfinite(value))) {
6          throw new IllegalArgumentException("---" + value);
7      }
8      beforeValue();
9      out.append(Double.toString(value));
10     return this;
11   }
12 }
```

(a) *Source class* from gson, with a method to write a value to a JSON array

```
1  public class JSONWriter implements Closeable, Flushable {
2    private JSONSerializer serializer;
3    ---
4    public void writeValue(Object object) {
5      writeObject(object);
6    }
7    public void writeObject(Object object) {
8      beforeWrite();
9      serializer.write(object);
10     afterWrite();
11   }
12 }
```

(b) *Target class* from fastjson, with a method to write a value to a JSON array

```
1  public final class JsonWriterTest extends TestCase {
2    public void testDoubles() throws IOException {
3      StringWriter stringWriter = new StringWriter();
4      JsonWriter jsonWriter = new JsonWriter(stringWriter);
5      jsonWriter.beginArray();
6      jsonWriter.value(-0.0);
7      jsonWriter.value(1.0);
8      jsonWriter.value(Double.MAX_VALUE);
9      jsonWriter.value(Double.MIN_VALUE);
10     jsonWriter.value(0.0);
11     jsonWriter.value(-0.5);
12     jsonWriter.value(2.2250738585072014E-308);
13     jsonWriter.value(Math.PI);
14     jsonWriter.value(Math.E);
15     jsonWriter.endArray();
16     jsonWriter.close();
17     assertEquals("---", stringWriter.toString());
18   }
19 }
```

(c) *Source test class* for testing gson's `value`

```
1  public final class JSONWriterWriteValueTest extends TestCase {
2    public void testDoubles() throws IOException {
3      StringWriter stringWriter = new StringWriter();
4      JSONWriter jsonWriter = new JSONWriter(stringWriter);
5      jsonWriter.startArray();
6      jsonWriter.writeValue(-0.0);
7      jsonWriter. writeValue(1.0);
8      jsonWriter. writeValue(Double.MAX_VALUE);
9      jsonWriter. writeValue(Double.MIN_VALUE);
10     jsonWriter. writeValue(0.0);
11     jsonWriter. writeValue(-0.5);
12     jsonWriter. writeValue(2.2250738585072014E-308);
13     jsonWriter. writeValue(Math.PI);
14     jsonWriter. writeValue(Math.E);
15     jsonWriter.endArray();
16     jsonWriter.close();
17     assertEquals("---", stringWriter.toString());
18   }
19 }
```

(d) Migrated test class to test fastjson's `writeValue`

Fig. 2: An example of JTESTMIGRATOR migrating tests from gson to fastjson. Given the `value` method in Figure 2a and its semantically similar method `writeValue` in Figure 2b, JTESTMIGRATOR migrates the original tests in Figure 2c to fastjson as shown in Figure 2d.

example. JTESTMIGRATOR can take the above inputs directly from unit test recommendation techniques without any manual intervention. We envision test recommendation techniques using JTESTMIGRATOR for test code transformation.

Given the above input, JTESTMIGRATOR performs test migration. We divide the test migration task into the following four subtasks. (1) *Test traceability* - identifying unit tests for a method such as `testDoubles` for `value` in our example. We use multiple heuristics to establish traceability. (2) *Similar code identification* - identifying semantically similar code elements in two different applications such as `beginArray` and `startArray`. We use an approach that combines word embeddings, text similarity, and type similarity for this task. (3) *Code transformation* - replacing code elements from a source application used in tests under migration (TUM) with semantically similar code elements from a target application while preserving the semantics of the test code. We use a systematic code replacement approach that considers semantic similarity and type compatibility of the code elements used in TUM. (4) *Test input and oracle reuse* - reusing test inputs and oracles in TUM to test a target method. We mainly use a search-based approach guided by types to solve this problem. We now explain each of the above subtasks/steps in detail.

A. Setup and Copy Tests

JTESTMIGRATOR currently supports two popular build systems, Gradle [33] and Maven [34], both of which support a similar project structure that JTESTMIGRATOR leverages. If the target application does not contain a test directory, JTESTMIGRATOR first creates this test directory (`src/test/java`). It then creates a package directory structure that corresponds to the target class' package structure and copies the source test class to this new directory. We refer to this copied source test class as *target test class*. Once JTESTMIGRATOR creates this initial version of the target test class, it then resolves library dependencies in the target application. JTESTMIGRATOR parses the dependency file (*build.gradle* or *pom.xml*) of both source and target applications and import statements of the target test class. It adds all the dependencies used by the target test class to the target application if the target application does not already contain these dependencies.

B. Trace Tests

The target test class may contain several tests to test various methods from the source class/application. However, we need only those tests that are testing the source method, `value` in our example. JTESTMIGRATOR uses two different criteria to check whether a test tests the source method: (1) whether the name of the test contains the name of the input source method and the test invokes the input source method and (2) whether the test invokes the input source method. JTESTMIGRATOR uses the second criteria only if the first criteria does not return any test. It then removes all other tests from the target test class. JTESTMIGRATOR skips this step if its input includes the tests that need to be migrated (e.g., tests already recommended by test recommendation techniques).

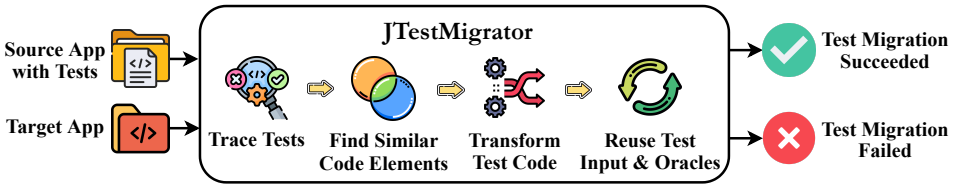

Fig. 3: High-level overview of JTESTMIGRATOR.

C. Find Similar Code Elements

At this point, we know which tests we need to migrate. Before we attempt to transform the code elements used in these tests, we first need to find semantically similar code elements in the target application. A test is basically a sequence of method invocations and constructor calls [12]. JTESTMIGTAX shows that a potential test migration technique needs to replace methods, constructors, and types [28]. Therefore, JTESTMIGRATOR finds a semantically similar target method/type for each source method/type used in the tests.

To determine the semantic similarity of methods, we consider both method name and parameter similarity as follows. We use SMFINDER [28] to determine method name similarity *nameSim*. SMFINDER leverages a Word2Vec [35] model generated from the method signatures present in the Java classes of 29,271 Android applications from AndroZooOpen [36]. Given a method pair, e.g., *writeValue* and *writeNumber*, SMFINDER first splits the methods by camel case: {write value}, {write, number}. For each word in the source method, it then uses cosine similarity to identify the best matching word in the target method: {write, write} and {value, number}. It then computes *nameSim* by taking the average of the cosine similarity scores of the best matching words: (cosine(write, write)+cosine(value, number))/2). The name similarity score *nameSim* ranges from 0 to 1.

SMFINDER only considers name similarity. However, a source or target application could have multiple similar or same-named methods (e.g., overloaded methods). Therefore, we also consider parameter similarity. We calculate parameter similarity *paramSim*, ranging from 0 to 1, as follows. Given two methods, *paramSim* is 0 if both methods do not have the same number of parameters and 1 if both methods have no parameters. Otherwise, we first calculate a type similarity value, *typeSim*, for each type pair formed by taking parameter types positioned at the same place in both methods. The *typeSim* value is 0 if types do not exactly match, 1 if they exactly match, and $W_c < 1$ if they are compatible (i.e., can be transformed to another type in the pair). To determine compatible types, we create a predefined database that contains a list of compatible source/target types for commonly used types. The database currently has type information for commonly used file handling types, primitive types, generics, and some other Java types, shown in Table I. We calculate *paramSim* as the average of *typeSim* values.

The final similarity score M_{sim} is a weighted sum of *nameSim* and *paramSim*. Specifically, $M_{sim} = nameSim * W_n + paramSim * W_p$, where W_n and W_p are weights for name and parameter similarity, respectively.

TABLE I: Database of convertible data types

Given Type	Compatible Types
File	FileInputStream, FileOutputStream, BufferedInputStream, BufferedOutputStream, ReadableByteChannel, WritableByteChannel, String
Java primitive	Object, Corresponding Java wrapper classes
Java wrapper	Corresponding Java primitive
Reference types	Object, Java generics
Number	Direct subclasses of Number
Writer	Direct subclasses of Writer

For each method used in the target test class, JTESTMIGRATOR selects a semantically similar method from the target application based on the highest M_{sim}. However, it is possible that more than one method has the same highest similarity score. In such cases, JTESTMIGRATOR leverages Levenshtein distance [37] to resolve the conflict and select a semantically similar method. For each semantically similar method pair JTESTMIGRATOR detects, it considers the enclosing classes as semantically similar types. For the remaining types used in the tests, JTESTMIGRATOR finds their semantic equivalent by using only name similarity *nameSim* between the class names.

D. Transform Test Code

Now that we have determined the method/type similarity between the source and target application, JTESTMIGRATOR replaces/modifies code elements of the target test class in order to prepare it for being able to invoke the target method.

1) Code Transformation: JTESTMIGRATOR first finds and replaces all instances of the input source-method invocation in the target test class with an invocation of the target method. For example, JTESTMIGRATOR replaces the method invocations `jsonWriter.value(...)` from Figure 2c with the method invocations `jsonWriter.writeValue(...)` in Figure 2d while keeping the same arguments. We discuss replacing the arguments in Section III-E. JTESTMIGRATOR checks exceptions thrown by the target method and adds the thrown exceptions to the test method if they are not already present in the test method.

At this point, the target test class still has constructor calls of the source class, which need to be replaced with target-class constructor calls. Assume that T_s and T_t are reference data types representing the source and target class, respectively. For our motivating example, T_s and T_t correspond to `JsonWriter` and `JSONWriter`, respectively. For each variable or field declaration of type T_s in the target test class, JTESTMIGRATOR replaces the variable or field's type with T_t. For example, JTESTMIGRATOR transforms the variable

declaration `JsonWriter jsonWriter` (Figure 2c, Line 4) to `JSONWriter jsonWriter` (Figure 2d, Line 4). For each object of type T_s, JTESTMIGRATOR replaces the object's type with the type T_t without modifying the constructor arguments. For example, JTESTMIGRATOR transforms the object `new JsonWriter(...)` (Figure 2c, Line 4) to `new JSONWriter(...)` (Figure 2d, Line 4). We discuss replacing constructor arguments in Section III-E. JTESTMIGRATOR also takes subclass information into account by replacing any instances of types T_{s-sub} with T_t. In cases where the constructor of type T_t is `private` and therefore cannot be used, JTESTMIGRATOR searches for a `static` method in the target class that returns an object of type T_t and replaces the constructor calls of type T_s in the target test class with calls to this static method. If both the source and target methods are `static`, JTESTMIGRATOR directly replaces T_s with T_t as the target-method call receiver. If only the source method is *static*, JTESTMIGRATOR creates an object of type T_t and uses the object as the receiver of the target-method call.

JTESTMIGRATOR then finds and replaces any other methods from the source application that are being invoked in the target test class. For our example in Figure 2c, such methods are *beginArray* (Line 5), *endArray* (Line 15), and *close* (Line 16). We refer to these other invoked methods as ***helper source methods***. In addition to the input source method, the source class, and helper source methods, JTESTMIGRATOR also replaces all reference data types from the source application used in the target test class. JTESTMIGRATOR replaces methods and types using the similarity mapping dataset created in Section III-C. It also adjusts the types of the variables that store the values returned by the replaced method calls if the return types of the replaced and original methods are different.

Although a test is mainly composed of method invocations and constructor calls, the compilation of these code elements also depends on various class-level code elements. Therefore, JTESTMIGRATOR also transforms the dependent class-level code elements. JTESTMIGRATOR replaces the package name declared in the target test class (i.e., *source package*) with the package name declared in the target class (i.e., *target package*). Since the target test class originates from the source test class, at this point, the target test class may have usages of *static* fields declared in the source class. JTESTMIGRATOR adds declarations for the used `static` fields in the target test class.

2) Test Resource Migration: The target test class might use resources, such as files, and test helper classes to test the input source method. JTESTMIGRATOR finds and locates any resources used in the target test class, and copies these resources from the source application directory to the target application directory. To migrate the test helper classes, JTESTMIGRATOR finds the receivers of all the methods called in the target test class. For each receiver, JTESTMIGRATOR then checks whether the receiver's class type matches with a class name from the source application. If it does and the file containing the class is in the source application test directory, JTESTMIGRATOR copies this file to the target application test directory. After copying the file, JTESTMIGRATOR also performs code transformation on the helper class.

3) Test Code Cleanup: At this stage, the target test class might have some code elements from the source application that JTESTMIGRATOR could not replace. The target test class might use some reference types or fields from the source application that are not available in the target application. JTESTMIGRATOR removes all the code elements that it could not replace or are no longer required.

E. Reuse Test Input and Oracles

Test inputs and oracles are key elements of a test. Our goal is to preserve manually written inputs and oracles while reusing tests in the target application. Therefore, JTESTMIGRATOR reuses the test inputs and oracles from the source application. However, although JTESTMIGRATOR does not change the test input values, it may not be possible to directly use the test inputs due to differences in the number and types of parameters between the source and target methods [22]. This also reflects in JTESTMIGTAX in Figure 1, which shows that input-related code changes in test migration involve modifying arguments' type or value, adding arguments, and removing arguments. JTESTMIGRATOR supports all the input-related code changes in JTESTMIGTAX except modifying arguments' value. We discuss how JTESTMIGRATOR converts and infers test inputs in Section III-E1 and generates test inputs that cannot be inferred from the test inputs available in the target test class in Section III-E2.

1) Test Input Transformation: JTESTMIGRATOR kept the arguments of any replaced source program elements intact during code transformation. Thus at this point, JTESTMIGRATOR tries to infer test inputs for the transformed tests. For each replaced constructor and method invocation, JTESTMIGRATOR checks whether the parameter types of the replaced program element are the same as the parameter types of its counterpart in the target application. If the parameter types are the same, JTESTMIGRATOR does not change the arguments. Otherwise, JTESTMIGRATOR changes the arguments of the target program elements. The process to infer test inputs for the target method and any helper target methods is the same. Therefore, we only discuss the process for the target method.

Since fields from the target test class or local variables from the current test method can be passed to the target-class constructor or the target-method arguments, JTESTMIGRATOR finds and stores all fields and local variables that are within the scope as *potential inputs* for the target-class constructor and target-method. However, some potential inputs may not have the required types. JTESTMIGRATOR tries to convert such potential inputs to the needed target types using the convertible type database in Table I. For each potential input value that does not have a direct match, JTESTMIGRATOR searches the database to find alternative compatible types. If one of the compatible types matches with a parameter type of the target-class constructor or the target method, JTESTMIGRATOR keeps this potential input as an acceptable value for the argument corresponding to the matched parameter type.

Otherwise, JTESTMIGRATOR removes this input from the list of potential inputs for this argument.

Given the filtered list of potential inputs, JTESTMIGRATOR attempts to convert their types to the expected argument type if the Java compiler cannot perform an implicit conversion. The database we created (Table I) also contains an initialization pattern for each source and target type pair, such as "`.getAbsolutePath()`" and "`.getParent()`" patterns for converting an input of type `File` to `String`. JTESTMIGRATOR uses these initialization patterns to convert the types of potential inputs. For our example, although the source and target methods have different parameter types (i.e., `double` and `Object`, Figure 2a and 2b), JTESTMIGRATOR does not convert the input values used in the source test class in Figure 2c because the Java compiler automatically does that. JTESTMIGRATOR generates potential inputs for each argument of the target-method invocation in a similar way.

2) Test Input Generation: JTESTMIGRATOR may not find potential input types that can be convertible to some parameter types of the target-class constructor or the target method. In such cases, JTESTMIGRATOR instead attempts to generate inputs for these parameters.

For each parameter for which JTESTMIGRATOR failed to convert the available input types, it declares the parameter as a field variable. It then uses two different approaches to initialize the field based on its type: (1) If the declared field is a Java primitive or `String` type, JTESTMIGRATOR generates a random value and initializes the field with the random value. (2) If the declared field is of a reference type from the target application, JTESTMIGRATOR instantiates an object of the reference type and initializes the field with this object. JTESTMIGRATOR then infers input for the constructor of the instantiated object by following the same approach as used to infer input for the target-class constructor.

3) Test Input Selection: After test input transformation and test input generation, JTESTMIGRATOR has a set of inputs for each target-class constructor and target-method invocation. Each input set is essentially a matrix of the parameter index and potential input values for this parameter. JTESTMIGRATOR then generates a cartesian product of the sets of inputs. Each entry in the resulting set acts as a distinct input set for the target-class constructors and target-method invocations.

JTESTMIGRATOR then uses a dynamic approach to select test inputs. Specifically, JTESTMIGRATOR executes the target test class with each distinct input set. For the first input set, if the execution fails with compilation errors, JTESTMIGRATOR stops executing the target test class with the other input sets and indicates that the test cannot be migrated. The compilation errors indicate that JTESTMIGRATOR could not successfully transform the test code. Therefore, JTESTMIGRATOR does not execute the target test class with the other input sets. However, if there are no compilation errors, JTESTMIGRATOR checks the results of the test execution. If the test passes, JTESTMIGRATOR does not execute the test with the other input sets and reports that the test migration is successful. However, if the test fails, JTESTMIGRATOR keeps executing the test with the other input sets, until the test passes or JTESTMIGRATOR exhausts all the input sets. If the test does not pass after exhausting all the input sets, JTESTMIGRATOR keeps the last input set it tries and reports that the migration failed.

IV. EVALUATION SETUP

To evaluate our approach, we implement JTESTMIGRATOR and investigate the following research questions:

- **RQ1:** *How many tests can* JTESTMIGRATOR *successfully migrate?* Successfully migrated tests are those that JTESTMIGRATOR correctly transforms into passing tests in the target application. Successfully migrated tests demonstrate JTESTMIGRATOR's capability to automatically reuse tests.
- **RQ2:** *How do* JTESTMIGRATOR*'s migrated tests compare to human migrated tests?* To provide additional qualitative insights into the generated tests, we compare JTESTMIGRATOR's successfully migrated tests to the human migrated tests in JTESTMIGBENCH.
- **RQ3:** *How many of the* JTESTMIGRATOR*'s successfully migrated tests are useful?* We check whether JTESTMIGRATOR's successfully migrated tests improve the code coverage of the target methods or applications.

To calculate a similarity score for methods, we use 0.5, 0.5, and 0.6 values for W_n, W_p, and W_c, respectively, based on our initial experiments.

A. Study Subjects

We evaluate JTESTMIGRATOR on JTESTMIGBENCH dataset [28], which contains 510 manually migrated JUnit tests for 186 methods from 5 popular libraries: JSON-java, gson, fastjson, commons-lang, and guava. We use JTESTMIGTAX to systematically select the data from JTESTMIGBENCH for evaluating JTESTMIGRATOR. Figure 1 shows what types of code changes JTESTMIGRATOR currently supports (in green) and does not support (in red). For example, JTESTMIGRATOR cannot support tests that involve replacing a source method with multiple target methods (i.e., *one-to-many*) and vice versa (i.e., *many-to-one*). Overall, we select only those unit tests and their corresponding methods from JTESTMIGBENCH that do not require the code changes highlighted in red in Figure 1, as we already know that JTESTMIGRATOR does not support these. This results in 104 unit tests for 45 method pairs from all five libraries as shown in Table II. The method pairs have 42 unique target methods. Overall, we evaluate JTESTMIGRATOR by attempting migration of 104 tests for 42 target methods.

V. EVALUATION RESULTS

A. RQ1: Successfully migrated tests

1) Method: If the inputs to JTESTMIGRATOR do not include which tests to migrate, it first identifies the tests to migrate for the given source method. Otherwise, it migrates the specified tests. For each method pair in Table II, we run JTESTMIGRATOR twice. We first run JTESTMIGRATOR without specifying tests and check how many of the corresponding

TABLE II: Study subjects

Source Library	Target Library	#Similar Method Pairs (selected/total)	#JUnit tests (selected/total)
JSON-java	gson	1/27	1/70
gson	JSON-java	3/10	4/15
JSON-java	fastjson	4/30	4/100
fastjson	JSON-java	6/36	9/59
gson	fastjson	19/33	66/185
fastjson	gson	1/30	1/47
commons-lang	guava	3/10	9/20
guava	commons-lang	8/10	10/14
Total	**-**	**45/186**	**104/510**

tests from the last column of Table II does JTESTMIGRATOR correctly identify and attempt to migrate. We next run JTESTMIGRATOR by providing specific tests. In both setups, we execute each migrated test that does not have compilation errors. If the migrated test passes, we determine that the test has been successfully reused in the target library. If the test fails or has compilation errors, we compare the test with the corresponding manually migrated test in JTESTMIGBENCH to identify the reasons for unsuccessful migration.

2) Results: Table III shows the results for the 42 methods. We first examine JTESTMIGRATOR's ability to detect the source tests that need to be migrated. Out of the 104 source tests, JTESTMIGRATOR was able to successfully find 86 tests. The tests JTESTMIGRATOR fails to identify are due to the criteria it uses to identify tests in Section III-B. Given a source method `add` and a source test class containing tests `testAdd` and `testNumbers` that have at least one statement that invokes the `add`, JTESTMIGRATOR will identify only `testAdd` as a test for the `add` method using the first criteria. Since the first criteria returns at least one test, JTESTMIGRATOR does not use the second criteria, which results in missing `testNumbers` as another test. However, we found that using both criteria to identify tests results in false positives, where the source method is only used as a helper method, which is why we use this current design.

To focus on JTESTMIGRATOR's migration abilities, we now look at the results of running it while specifying the tests to migrate (last two columns of Table III). JTESTMIGRATOR successfully migrates 76 (73%) tests for 32 (76%) methods across the 5 libraries. This means that JTESTMIGRATOR enabled full automated reuse of 73% of tests.

Among the 28 tests that JTESTMIGRATOR could not successfully migrate, 9 tests have compilation errors and 19 tests failed. Table IV shows the reasons for unsuccessful migrations. Note that there may be multiple reasons for unsuccessful migrations in a test. JTESTMIGRATOR could not successfully migrate 16 of the 28 tests because they require the types of code transformations that it does not support. Specifically, these tests require a method call addition, a change in an expected output value, or a many-to-one method call replacement. For example, 15 of the 16 tests migrated from gson to fastjson for two methods require the addition of a `close` method call, similar to line 16 shown in Figure 2d. However, unlike the motivational example, the original tests in these cases do not include a `close` method call. The 16 tests are not labeled with these code changes in the original dataset.

TABLE III: Successfully migrated tests

Source Library	Target Library	No Input Test	With Input Tests	
		#Tests Identified	#Passing Tests	#Methods
JSON-java	gson	1/1	1/1	1/1
gson	JSON-java	4/4	2/4	2/3
JSON-java	fastjson	3/4	1/4	1/4
fastjson	JSON-java	9/9	9/9	6/6
gson	fastjson	49/66	44/66	11/16
fastjson	gson	1/1	1/1	1/1
commons-lang	guava	9/9	9/9	3/3
guava	commons-lang	10/10	9/10	7/8
Total	**-**	**86/104**	**76/104**	**32/42**

TABLE IV: Reasons for unsuccessful migrations

Reasons	#Tests		
	Total	Compilation Errors	Test Failures
Unsupported code transformation	**16**		
Method call addition		4	11
Change in expected output values		-	1
Many-to-one method call replacement		1	-
Supported code transformation	**12**		
Incorrect similar method replacement		2	7
Incorrect type conversion		2	1
Incorrect input values		1	2
Sum	**28**	**10**	**22**

There are 12 tests JTESTMIGRATOR could not successfully migrate, because it performed certain code transformations incorrectly. In 9 tests, JTESTMIGRATOR finds and replaces similar methods that are not actually similar. For example, in Figure 4, JTESTMIGRATOR correctly finds and replaces `addProperty` (line 6) and `get` (line 8) methods from gson with `put` (line 28) and `get` (line 30) methods from fastjson, respectively. However, JTESTMIGRATOR incorrectly identifies `isValid` (line 29) and `getRawType` (line 32) in fastjson as similar methods to `has` (line 7) and `getAsString` (10) from gson, respectively. In this example, JTESTMIGRATOR also correctly adjusts type (`Object`, line 30) based on the return type of `get` but fails to remove the affected method `getRawType` (line 32), resulting in a compilation error.

JTESTMIGRATOR performs incorrect type conversion in 3 of the 12 unsuccessfully migrated tests. It also generates incorrect (random) input values in 3 tests. For example, `getAsBoolean` method in gson does not take a parameter, whereas a similar method `getBoolean` in JSON-java takes a string parameter. Therefore, while migrating a test containing `getAsBoolean`, JTESTMIGRATOR generates a random string input and passes it to `getBoolean`, resulting in a test failure.

RQ1: JTESTMIGRATOR successfully migrates 76 (73%) tests, allowing automatic test reuse for 32 (76%) methods across the 5 libraries. Reasons for unsuccessful migrations include unsupported code transformations or incorrectly performed transformations.

B. RQ2: Qualitative analysis of the migrated tests

1) Method: While passing tests is a positive result, it does not tell us anything about what is in the migrated tests. For example, JTESTMIGRATOR could create empty tests (i.e., not

```java
//An original test from gson
public void testAddingStringProperties() throws Exception {
  String propertyName = "property";
  String value = "blah";
  JsonObject jsonObj = new JsonObject();
  jsonObj.addProperty(propertyName, value);
  assertTrue(jsonObj.has(propertyName));
  JsonElement jsonElement = jsonObj.get(propertyName);
  assertNotNull(jsonElement);
  assertEquals(value, jsonElement.getAsString());
}
//A manually migrated test for fastjson
public void testAddingStringProperties() throws Exception {
  String propertyName = "property";
  String value = "blah";
  JSONObject jsonObj = new JSONObject();
  jsonObj.put(propertyName, value);
  assertTrue(jsonObj.containsKey(propertyName));
  Object jsonElement = jsonObj.get(propertyName);
  assertNotNull(jsonElement);
  assertEquals(value, jsonElement);
}
//A JTestMigrator's migrated test for fastjson
public void testAddingStringProperties() throws Exception {
  String propertyName = "property";
  String value = "blah";
  JSONObject jsonObj = new JSONObject();
  jsonObj.put(propertyName, value);
  assertTrue(JSONObject.isValid(propertyName));
  Object jsonElement = jsonObj.get(propertyName);
  assertNotNull(jsonElement);
  assertEquals(value, jsonElement.getRawType());
}
```

Fig. 4: A test from gson and the migrated tests for fastjson

```java
//An original test from gson
public void testEqualsNonEmptyArray() {
  JsonArray a = new JsonArray();
  JsonArray b = new JsonArray();
  ...
  b.add(new JsonObject());
  MoreAsserts.assertEqualsAndHashCode(a, b);
  ...
  assertFalse(b.equals(a));
  b.add(JsonNull.INSTANCE);
  assertFalse(a.equals(b));
  assertFalse(b.equals(a));
}
//A manually migrated test for fastjson
public void testEqualsNonEmptyArray() {
  JSONArray a = new JSONArray();
  JSONArray b = new JSONArray();
  ...
  b.add(new JSONObject());
  assertEqualsAndHashCode(a, b);
  ...
  assertFalse(b.equals(a));
  b.add(JsonNull.INSTANCE);
  assertFalse(a.equals(b));
  assertFalse(b.equals(a));
}
//JTestMigrator's migrated test for fastjson
public void testEqualsNonEmptyArray() {
  JSONArray a = new JSONArray();
  JSONArray b = new JSONArray();
  ...
  b.add(new JSONObject());
  MoreAsserts.assertEqualsAndHashCode(a, b);
  ...
  assertFalse(b.equals(a));
  assertFalse(a.equals(b));
  assertFalse(b.equals(a));
}
```

Fig. 5: A test from gson and the migrated tests for fastjson

migrate anything) and still result in a passing test. Therefore, in this RQ, we are interested in diving deeper into the results of the migration. Specifically, we compare JTESTMIGRATOR's 76 successfully migrated tests (i.e., those that pass) to the corresponding human migrated tests in JTESTMIGBENCH. If they are different, we investigate the reasons for differences.

2) Results: We find that 66 (87%) of the JTESTMIGRATOR's successfully migrated tests are exactly the same as the corresponding human migrated tests. Table V shows the reasons for differences in the remaining 10 tests. Note that there are multiple reasons for the differences in a test.

In 7 of the tests, the differences is due to the human migrator performing some form of refactoring. In the example

TABLE V: Reasons for differences in JTESTMIGRATOR's migrated tests and human migrated tests

Reasons	#Tests
Refactoring performed in human migrated tests	7
JTESTMIGRATOR was not successful in transforming some code elements	3
Use of alternate semantically similar code elements	2
Total	**12**

```java
//An original test from gson
public void testSize() {
  JsonObject o = new JsonObject();
  assertEquals(0, o.size());
  o.add("Hello", new JsonPrimitive(1));
  assertEquals(1, o.size());
  ...
}
//A manually migrated test for JSON-java
public void testSize() {
  JSONObject o = new JSONObject();
  assertEquals(0, o.length());
  o.put("Hello", 1);
  assertEquals(1, o.length());
  ...
}
//A JTestmigrator's migrated test
public void testSize() {
  JSONObject o = new JSONObject();
  assertEquals(0, o.length());
  o.append("Hello", new JSONArray(1));
  assertEquals(1, o.length());
  ...
}
```

Fig. 6: A test from gson and the migrated tests for JSON-java

in Figure 5, JTESTMIGRATOR correctly used `MoreAsserts` (line 33), which is a helper class in gson that declares various assertion methods. Whereas, the human migrator extracted `assertEqualsAndHashCode` method declared in `MoreAsserts` and placed it in the migrated test class. In another example in Figure 6, JTESTMIGRATOR replaces `JsonPrimitive` with `JSONArray` (line 21) whereas the human migrator simply passes the integer 1 instead of using any type (line 13). These cases show how human intelligence plays a role during code transformation.

In 3 of the 10 different tests, JTESTMIGRATOR could not replace a field accessed in a statement, and therefore removed the entire statement to avoid compilation errors. In the example in Figure 5, JTESTMIGRATOR removed the statement `b.add(Jsonnull.INSTANCE)` on line 10, because it could not replace the accessed field `Jsonnull.INSTANCE`. Note that these tests still pass and have assertions, still making them useful to the target libraries. Finally, in 2 of the 10 tests, JTESTMIGRATOR uses a different semantically similar method. In the example in Figure 6, JTESTMIGRATOR replaces the `add` method from gson with a similar method `append` from JSON-java (line 21) without affecting the end result, whereas the human migrator replaces it with `put` (line 13). These cases as well as refactoring cases indicate that there are multiple ways to correctly transform code elements.

***RQ2**:* 87% of the JTESTMIGRATOR's successfully migrated tests are exactly the same as the corresponding human migrated tests, indicating that JTESTMIGRATOR can migrate tests that look like human migrated tests. The different migrated tests indicate that there are multiple ways to correctly transform a code element.

C. RQ3: Usefulness of successfully migrated tests

1) Method: In this RQ, we are interested in checking whether the JTESTMIGRATOR's 76 successfully migrated tests for the 32 methods augment existing tests by improving the

Fig. 7: Coverage improved by the successfully migrated tests (M=Method ID, I=Instruction Coverage, B=Branch Coverage)

methods' code coverage. To determine this, we measure the methods' coverage using JaCoCo [38]. We first measure the methods' instruction and branch coverage without migrating tests. For the methods that have less than 100% instruction or branch coverage, we again measure their instruction and branch coverage after including successfully migrated tests.

2) Results: 6 of the 32 methods have less than 100% instruction or branch coverage. Figure 7 shows the instruction and branch coverage for the 6 methods before and after test migration. We find that 10 (13%) of the successfully migrated tests improve coverage of 4 methods. `isEmpty` method (M1 in Figure 7) in gson is not covered by any existing tests. In this case, a JTESTMIGRATOR's migrated test increases the instruction coverage by 100%. For the remaining 3 methods (M4, M5, and M6) in fastjson, JTESTMIGRATOR's migrated tests increase their instruction or branch coverage although they had high existing coverage (>82%). The libraries used in the evaluation are highly popular with many unit tests. Despite that, the migrated tests increase coverage of the 4 methods.

***RQ3**:* 13% of the JTESTMIGRATOR's successfully migrated tests increase code coverage of 4 methods, showing some practical values of JTESTMIGRATOR's migrated tests.

D. Threats to Validity

Construct validity For evaluating JTESTMIGRATOR, we systematically select only those JUnit tests from JTESTMIGBENCH that require the types of code transformations JTESTMIGRATOR supports. The purpose is to evaluate the effectiveness of JTESTMIGRATOR in performing the types of code transformations it supports rather than finding what it cannot perform, which we already know from JTESTMIGTAX.

Internal validity We consider a test successfully migrated only if it passes, but a migrated test may fail due to bugs in the target system. However, the results (RQ1) show that the unsuccessful migrations are due to incorrect code transformations rather than bugs. We manually compare the tests migrated by JTESTMIGRATOR with their corresponding manually migrated tests to find the reasons for unsuccessful migrations (RQ1) and qualitatively analyze the migrated tests (RQ2). Manual tasks are subject to mistakes, and they might cause biased and hard-to-reproduce results. We release all the artifacts used in this study for further validation [29].

External validity While we cannot claim that our results generalize beyond our subjects, we perform our evaluation by attempting migration of 104 unit tests for 42 methods. Previous test reuse work that reuses tests across different applications was evaluated on only 12 to 29 methods [23], [26].

VI. DISCUSSION

A. Implications of the Results

We proposed a technique and tool to migrate unit tests between applications with similar functionality. JTESTMIGRATOR was able to successfully migrate 73% of the tests, making them directly usable in the target libraries. Furthermore, 87% of those successfully migrated tests looked exactly the same as the human migrated tests in JTESTMIGBENCH, which means JTESTMIGRATOR could have saved human effort while producing the same results. Moreover, 13% of the successfully migrated tests incresed coverage of the target systems, showing some practical values of test migration.

Although, to the best of our knowledge, there are no tools that we can directly compare JTESTMIGRATOR against, SKIPPER [25] and METALLICUS [26] are the two closest tools that reuse unit tests by performing limited code transformation. SKIPPER reported migrating 245 (80%) passing tests out of 308 tests. However, the target applications used to evaluate SKIPPER are minimally modified versions of the corresponding source applications, which is why 23% of the tests migrated by SKIPPER do not even require code transformation. METALLICUS, on the other hand, inserts assertions in syntactically valid manually-created test templates instead of directly transforming original tests. Different from both these efforts, JTESTMIGRATOR migrates tests for target libraries that are completely different than the source libraries and directly transforms original tests. Overall, our results are promising and show that the concept of reusing unit tests across similar applications can be practical and useful.

B. Key Challenges in Unit Test Migration

1) Finding Semantically Similar Code for Code Transformation: A key challenge in migrating unit tests is finding semantically similar code (methods and classes). Most clone detection techniques still struggle to find semantically similar code [39]. Overall, our evaluation showed that our approach performed relatively well in finding semantically similar helper methods and types. However, there are shortcomings especially when a Java class in an application has multiple methods with similar names and the same parameter types. Large Language Models have shown some promising results in detecting code clones [40]. In general, advances in research related to detecting semantically similar methods can also further improve the results of test migration [26], [41], [42].

2) Implementation Differences: The challenge does not end at finding similar methods or types. Even similar methods may have slight implementation differences that may result in unsuccessful test migrations. We mainly observed two kinds of implementation differences. First, semantically similar methods return the same output, but with different data types, for

the same input as we saw in Figure 4 for `get` method (line 8 and 30). JTESTMIGRATOR currently maintains a list of some common data types for type conversion. However, future test reuse techniques might need to explore additional techniques for type conversion [43] such as human-in-the-loop [44]. Second, semantically similar methods may have some implementation differences in how they handle certain input or corner cases. In this case, we have to change the expected outputs in the assertions while migrating tests, popularly known as the oracle problem [45], which JTESTMIGRATOR does not handle. It is also an open problem that researchers have tried to solve through various techniques [17], [44], [46], [47]. Overall, techniques to adjust data types and expected outputs can improve the results of the migration.

C. Applicability of JTESTMIGRATOR

We envision various practical scenarios where JTESTMIGRATOR can be used. One main scenario is to use JTESTMIGRATOR along with code [48], [49] and test [22], [23], [50] recommendation techniques. For example, once developers use the recommended code, they can migrate the corresponding tests using JTESTMIGRATOR to maintain code quality. JTESTMIGRATOR can also be integrated into test recommendation techniques to eliminate manual test reuse effort.

VII. RELATED WORK

Unit Test Tracing, Recommendation, and Reuse: White et al. [24] used various criteria, such as the similarity of method and test name, to trace tests. JTESTMIGRATOR also uses similar criteria to find tests for the target methods. To help developers in writing tests manually, researchers have proposed techniques to recommend tests based on the source method similarity [23], [50], [51]. Similar to some of these techniques, JTESTMIGRATOR also uses word2vec to find similar methods. However, while these techniques stop at recommending tests, JTESTMIGRATOR transforms the tests to reuse them in target applications. Sondhi et al. [22] and White et al. [23] adapted some of the recommended tests manually to show that tests can be reused across applications. Sondhi et al. [26] later proposed METALLICUS to reuse oracles of the recommended tests. METALLICUS requires that users provide a manually written template with at least one valid test, whereas JTESTMIGRATOR directly copies the tests to be reused from the source application to the target application and performs code transformation. Makady and Walker [25] proposed SKIPPER that directly copies the tests to be reused from the source to the target application. However, SKIPPER is designed to reuse tests in the target application that is created by removing some code (fields, methods, or classes) from the source application. Therefore, SKIPPER does not need to perform various code transformations, such as finding and replacing similar types and method calls. SKIPPER requires a manually written test reuse plan, which defines the code to be reused in the target application. JTESTMIGRATOR does not require a test reuse plan and uses a semantic code mapping technique to transform tests. Zhang et al. [52] proposed a technique to reuse unit tests for testing reused code/clones. However, rather than transforming test code, they transform cloned code to make the tests executable for the clone.

Unit Test Generation: There has been a lot of research for generating test inputs [53]–[55], [55], test oracles [17], [45], [56], [56], and test cases [13], [57], [58]. We limit our discussion to JCrasher [13], Randoop [58], and EvoSuite [14] that generate JUnit tests. These tools generate sequences of constructor/method calls, where each sequence acts as a unit test. The tools then use different techniques to generate test inputs and oracles. The main purpose of JCrasher is to detect bugs by causing the SUT to throw a runtime exception. Therefore, JCrasher exercises public methods by passing random inputs. Whereas, to avoid generating redundant and illegal inputs, Randoop uses a feedback-directed approach to generate test inputs. EvoSuite uses an evolutionary search-based approach optimized for coverage criteria to generate unit tests. In contrast to these tools, JTESTMIGRATOR reuses unit tests across applications by transforming the tests.

GUI Test Reuse in Mobile Apps: Behrang and Orso have a recent line of work on migrating GUI tests between Android apps [59]–[61]. Their recent work [59] and the work proposed by Lin et al. [62] leverage a Word2Vec model to find similar GUI elements. The techniques use a statically extracted GUI model to transform the GUI event sequences present in the tests. To account for a huge search space of GUI tests, Mariani et al. [63] proposed an evolutionary approach to reuse GUI tests across Android apps. Instead of directly transforming tests, Hu et al. [64] proposed a technique to create modular reusable tests that can be then reused across similar apps. Qin et al. [65] proposed a technique that migrates GUI tests across different platforms (iOS and Android). To facilitate the evaluation of GUI test reuse efforts, Zhao et al. [66] proposed a framework, which includes different GUI test migration techniques and a benchmark of reused GUI tests, to evaluate GUI test reuse across mobile apps. Different from the above work which focuses on GUI test migration in mobile apps, our work migrates unit tests across Java applications, which comes with different challenges due to the level of abstraction at which the technique works.

VIII. CONCLUSION

We proposed a technique and corresponding implementation, JTESTMIGRATOR, to migrate Java unit tests between similar applications. We evaluated JTESTMIGRATOR by attempting migrations of 104 unit tests across 5 popular libraries. JTESTMIGRATOR successfully migrated 73% of the tests, allowing their direct use in the target libraries. In 87% of these, JTESTMIGRATOR produced tests that are exactly the same as the corresponding human-migrated tests. Moreover, 13% of the successfully migrated tests increased code coverage of the target systems. Although this is the first attempt to migrate unit tests across applications by using a completely automated technique, our results are promising and motivate further research in this direction.

REFERENCES

[1] V. Garousi and M. V. Mäntylä, "A systematic literature review of literature reviews in software testing," *Information and Software Technology*, vol. 80, pp. 195–216, 2016.

[2] V. Garousi and J. Zhi, "A survey of software testing practices in canada," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354–1376, 2013.

[3] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[4] E. M. Maximilien and L. Williams, "Assessing test-driven development at ibm," in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 564–569.

[5] J. Langr, A. Hunt, and D. Thomas, *Pragmatic unit testing in java 8 with JUnit*. Pragmatic Bookshelf, 2015.

[6] M. Ellims, J. Bridges, and D. C. Ince, "The economics of unit testing," *Empirical Software Engineering*, vol. 11, no. 1, pp. 5–31, 2006.

[7] J. Shore, "Fail fast [software debugging]," *IEEE Software*, vol. 21, no. 5, pp. 21–25, 2004.

[8] E. Dustin, T. Garrett, and B. Gauf, *Implementing automated software testing: How to save time and lower costs while raising quality*. Pearson Education, 2009.

[9] P. S. Kochhar, X. Xia, and D. Lo, "Practitioners' views on good software testing practices," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 61–70.

[10] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 201–211.

[11] A. Deak, T. Stålhane, and G. Sindre, "Challenges and strategies for motivating software testing personnel," *Information and software Technology*, vol. 73, pp. 1–15, 2016.

[12] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[13] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[14] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[15] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 263–272.

[16] Z. Fan, "A systematic evaluation of problematic tests generated by evosuite," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 165–167.

[17] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 342–351.

[18] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 348–3483.

[19] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 57–67.

[20] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 23–32.

[21] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?" in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 291–301.

[22] D. Sondhi, D. Rani, and R. Purandare, "Similarities across libraries: Making a case for leveraging test suites," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 79–89.

[23] R. White, J. Krinke, E. T. Barr, F. Sarro, and C. Ragkhitwetsagul, "Artefact relation graphs for unit test reuse recommendation," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 137–147.

[24] R. White, J. Krinke, and R. Tan, "Establishing multilevel test-to-code traceability links," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 861–872.

[25] S. Makady and R. J. Walker, "Validating pragmatic reuse tasks by leveraging existing test suites," *Software: Practice and Experience*, vol. 43, no. 9, pp. 1039–1070, 2013.

[26] D. Sondhi, M. Jobanputra, D. Rani, S. Purandare, S. Sharma, and R. Purandare, "Mining similar methods for test adaptation," *IEEE Transactions on Software Engineering*, 2021.

[27] M. Landhäußer and W. F. Tichy, "Automated test-case generation by cloning," in *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE, 2012, pp. 83–88.

[28] A. K. Jha, M. Islam, and S. Nadi, "Jtestmigbench and jtestmigtax: A benchmark and taxonomy for unit test migration," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 713–717.

[29] Artifacts, "Migrating unit tests across applications," https://figshare.com/s/9172d87fbf0a188f9ccf, 2024.

[30] Gson, https://github.com/google/gson, 2024.

[31] Fastjson, https://github.com/alibaba/fastjson, 2024.

[32] G. J. Myers, T. Badgett, and C. Sandler, *The art of software testing*. Wiley Online Library, 2012, vol. 3.

[33] Gradle, https://gradle.org/, 2024.

[34] Maven, https://maven.apache.org/, 2024.

[35] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*, 2013.

[36] P. Liu, L. Li, Y. Zhao, X. Sun, and J. Grundy, "Androzooopen: Collecting large-scale open source android apps for the research community," in *2020 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 2020.

[37] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[38] JaCoCo, https://www.eclemma.org/jacoco/, 2024.

[39] A. Walker, T. Cerny, and E. Song, "Open-source tools and benchmarks for code-clone detection: past, present, and future trends," *ACM SIGAPP Applied Computing Review*, vol. 19, no. 4, pp. 28–39, 2020.

[40] S. Dou, J. Shan, H. Jia, W. Deng, Z. Xi, W. He, Y. Wu, T. Gui, Y. Liu, and X. Huang, "Towards understanding the capability of large language models on code clone detection: A survey," *arXiv preprint arXiv:2308.01191*, 2023.

[41] M. Kamp, P. Kreutzer, and M. Philippsen, "Sesame: a data set of semantically similar java methods," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 529–533.

[42] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, "Modeling functional similarity in source code with graph-based siamese networks," *IEEE Transactions on Software Engineering*, 2021.

[43] A. Ketkar, O. Smirnov, N. Tsantalis, D. Dig, and T. Bryksin, "Inferring and applying type changes," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1206–1218.

[44] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "An empirical validation of oracle improvement," *IEEE Transactions on Software Engineering*, 2019.

[45] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[46] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary improvement of assertion oracles," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1178–1189.

[47] N. Alshahwan, M. Harman, and A. Marginean, "Software testing research challenges: An industrial perspective," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 1–10.

[48] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[49] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "Facoy: a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 946–957.

[50] W. Janjic and C. Atkinson, "Utilizing software reuse experience for automated test recommendation," in *2013 8th International Workshop on Automation of Software Test (AST)*. IEEE, 2013, pp. 100–106.

[51] C. Zhu, W. Sun, Q. Liu, Y. Yuan, C. Fang, and Y. Huang, "Homotr: online test recommendation system based on homologous code matching," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1302–1306.

[52] T. Zhang and M. Kim, "Automated transplantation and differential testing for clones," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 665–676.

[53] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[54] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 249–260.

[55] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[56] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1398–1409.

[57] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2011.

[58] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 75–84.

[59] F. Behrang and A. Orso, "Test migration between mobile apps with similar functionality," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 54–65.

[60] ——, "Automated test migration for mobile apps," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 384–385.

[61] ——, "Test migration for efficient large-scale assessment of mobile app coding assignments," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 164–175.

[62] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 42–53.

[63] L. Mariani, M. Pezzè, V. Terragni, and D. Zuddas, "An evolutionary approach to adapt tests across mobile apps," *arXiv preprint arXiv:2104.05233*, 2021.

[64] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable ui tests," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 269–282.

[65] X. Qin, H. Zhong, and X. Wang, "Testmig: Migrating gui test cases from ios to android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 284–295.

[66] Y. Zhao, J. Chen, A. Sejfia, M. Schmitt Laser, J. Zhang, F. Sarro, M. Harman, and N. Medvidovic, "Fruiter: a framework for evaluating ui test reuse," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1190–1201.