MUT: Human-in-the-Loop Unit Test Migration

Yi Gao
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China
gaoyi01@zju.edu.cn

Xing Hu*
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China
xinghu@zju.edu.cn

Tongtong Xu
Huawei
Hangzhou, China
xutongtong9@huawei.com

Xin Xia
Huawei
Hangzhou, China
xin.xia@acm.org

David Lo
School of Computing and Information Systems, Singapore Management University
Hangzhou, China
davidlo@smu.edu.sg

Xiaohu Yang
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Hangzhou, China
yangxh@zju.edu.cn

ABSTRACT

Test migration, which enables the reuse of test cases crafted with knowledge and creativity by testers across various platforms and programming languages, has exhibited effectiveness in mobile app testing. However, unit test migration at the source code level has not garnered adequate attention and exploration. In this paper, we propose a novel cross-language and cross-platform test migration methodology, named MUT, which consists of four modules: code mapping, test case filtering, test case translation, and test case adaptation. MUT initially calculates code mappings to establish associations between source and target projects, and identifies suitable unit tests for migration from the source project. Then, MUT's code translation component generates a syntax tree by parsing the code to be migrated and progressively converts each node in the tree, ultima tely generating the target tests, which are compiled and executed in the target project. Moreover, we develop a web tool to assist developers in test migration. The effectiveness of our approach has been validated on five prevalent functional domain projects within the open-source community. We migrate a total of 550 unit tests and submitted pull requests to augment test code in the target projects on GitHub. By the time of this paper submission, 253 of these tests have already been merged into the projects (including 197 unit tests in the Luliyucoordinate-LeetCode project and 56 unit tests in the Rangerlee-HtmlParser project). Through running these tests, we identify 5 bugs, and 2 functional defects, and submitted corresponding issues to the project. The evaluation substantiates that MUT's test migration is both viable and beneficial across programming languages and different projects.

*Corresponding Author

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639124

ACM Reference Format:
Yi Gao, Xing Hu, Tongtong Xu, Xin Xia, David Lo, and Xiaohu Yang. 2024. MUT: Human-in-the-Loop Unit Test Migration. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639124

1 INTRODUCTION

During software development, developers often spend considerable time and effort manually writing unit test cases. To facilitate the testing process, many approaches are proposed to generate unit test cases automatically, such as EvoSuite [33] and Randoop [42]. These tools exploit random generation, genetic algorithms, and dynamic symbolic execution to generate test cases.

Existing studies mainly focus on generating unit test cases for Java and Python programming language [31, 33, 42, 44, 52]. However, generating test cases for C++ programs has not received much attention. Although Fuzz testing tools [34, 38, 57] are proposed to generate test inputs for C++ programs, generating unit test cases for them is challenging due to i) difficulty in tool development. Due to the high complexity and low-level nature of the C++ language, which involves handling numerous low-level concepts and technologies such as dynamic memory allocation and pointer arithmetic, developing unit test case generation tools for C++ is challenging, and ii) difficulty in generating effective test cases. As conventional reliance on random testing methods can produce randomized test data and logic that may lack real-world significance, thus limiting their effectiveness.

In modern software development, many software products with the same functionality are written in different programming languages for different platforms, such as Android phones have UI, WI-FI, and Bluetooth functions written in Java language, while OpenHarmony phones have the same functions, but they are implemented in C/C++ language. Inspired by the success in code migration [48], we argue that unit test cases can be migrated from an existing similar donor software with high-quality unit test cases.

To achieve this objective, we propose a migration technique aimed at generating new test cases from another programming language. By defining mapping rules between Java and C++ code and syntax, we can translate existing test cases written in Java

into C++ test cases, eliminating the need to create test cases from scratch. Furthermore, by leveraging the active open-source ecosystem and abundant test case resources available in Java, we migrate previously developed and validated test cases that were manually designed. These test cases possess explicit meanings in their test data and logic. Specifically, for the target migration project, we utilize the search function of the GitHub open-source community to identify projects with the same topic as the target migration project, implemented in Java and ranked high in terms of stars. These identified projects serve as candidate projects from which we extract the test cases eligible for migration. Moreover, the migrated C++ unit test code retains functional consistency with the original Java unit test code, ensuring that it continues to examine the same logic and behavior. As a result, this approach enables more effective identification of potential issues. Compared to generating unit test cases from scratch, migrating techniques can automatically translate original test cases into equivalent test cases of target code, so as to reduce the cost of testing and ensure the correctness and maintainability of migrated code.

To facilitate the generation of high-quality C++ test cases, we propose a test migration tool MUT, which targets the scope where two applications are different but functionally similar. The high-level intuition is that test cases, namely test bodies, and assertions, are simpler compared with general source codes, thus it is more feasible to conduct code migration. Existing studies [41] have made the first attempt to migrate UI tests from a source Android app to automatically generate equivalent tests for a target Android app. However, the task of migrating unit test cases between different programming languages has not gained much attention in previous research. Making such a tool for migrating unit test cases is a non-trivial task, considering the following challenges:

- *Establishing code mapping relationships.* To facilitate the migration of unit test cases between two projects implemented in different languages, we need to find similar production code snippets and establish a set of mappings between their code snippets, including classes, methods, and fields.
- *Migrating unit tests.* Once the code mapping relationships are established, we next need to identify the unit tests in Java projects that can be migrated. These migratable unit tests are then translated into the equivalent C++ unit tests in the target project.

To address the first challenge, MUT designs a code mapping method to identify functionally similar code snippet pairs between source and target projects. Subsequently, it establishes mapping rules for classes, methods, and fields. To address the second challenge, MUT identifies migratable unit tests in the source project by analyzing the invocation relationship between test methods and focal methods. Subsequently, based on the code mapping relationships obtained in the previous step, MUT translates the migratable Java unit tests to C++ unit tests by employing a set of code translation and replacement components based on Backus-Naur Form (BNF) rules [30]. BNF parsing guarantees the feasibility and high quality of code translation, as it captures the syntactic structure and semantic information of the source project comprehensively. We have devised a complete rule translation and replacement engine to handle each rule type in BNF flexibly and achieve source-to-target language translation of source code.

MUT can be easily integrated into the human test process to assist developers in understanding and writing their final-version test cases. To boost the efficient use of MUT, we develop a web interface that displays pertinent pieces of information during test migration including API matching relations, pre-migration test cases, and post-migration test cases. Developers can easily understand the reason that MUT generates the post-migration codes, and correspondingly confirm, revise, or reject the migration result at this time. We carefully design a comparison study to evaluate the usefulness of MUT in helping developers and find our users are satisfied with MUT in helping them efficiently write test cases. We successfully migrated 550 test cases across 10 projects in five categories. By compiling and running these tests, we detected a total of 7 issues in the projects.

In summary, the main contributions of this paper include:

- We propose MUT, to facilitate cross-language, cross-platform test case migration and enhance test coverage, and minimize manual labor. The tool is available on our website [1].
- We develop a web interface to assist testers in examining pre- and post-migration code details, improving the ease of adapting migrated test cases.
- Experiments with 15 open-source projects (including Jsoup, Joda-Time, Commons-lang, etc), show the effectiveness and practicality of the MUT tool. In total, we detect 7 vulnerabilities and merge 253 unit tests.

2 PRELIMINARIES

In this section, we illustrate how a developer can potentially boost his test process by referring to a similar project, then we depict how a developer actually boosts his test process by integrating with MUT. Finally, we provide the definition of our test migration.

2.1 Motivation

During software development, developers often write different software products with different programming languages. For example, Jsoup [14] is a top HTML parsing project written in Java with 10.1k stars on GitHub, and Mylogin-Htmlparser [20] is a lightweight HTML parsing project developed in C++, which is similar to Jsoup. We find that Jsoup has high-quality unit test cases while Mylogin-Htmlparser does not have sufficient tests.

As Figure 1 illustrates, the test method `findsCharsetInMalformedMeta` is a unit test within the Jsoup, it is utilized to validate the accurate parsing of `meta` elements and the retrieval of the `charset` attribute value from an HTML document. This unit test employs the `parse` method to parse HTML elements, the `select` method to select HTML tags from the parsing result, then the `attr` method to retrieve specific attribute values from the tags. All of these APIs, which implement fundamental HTML manipulation functionalities, are available in the Mylogin-Htmlparser. However, Mylogin-Htmlparser is not provided with any test cases to check the correctness of its functionality.

With our tool. Our tool MUT aims to facilitate the reuse of unit tests from source projects and generate tests for target projects. It works by taking two projects with similar functionalities as input. The usage scenario of MUT is as follows:

[1] https://github.com/testmigrator/mut

(a) Source test from Jsoup

```
@Test public void findsCharsetInMalformedMeta() {
    String h = "<meta http-equiv=Content-Type content=
        text/html; charset=gb2312>";
    // example cited for reason of html5's <meta
        charset> element
    Document doc = Jsoup.parse(h);
    assertEquals("gb2312", doc.select("meta").attr("
        charset"));
}
```

(b) Migrated test by MUT

```
void findsCharsetInMalformedMeta() {
    string h = "<meta http-equiv=Content-Type content=
        text/html; charset=gb2312>";
    node doc = parser.parse(h);
    ASSERT_EQ("gb2312", doc-> select("meta")->attr("
        charset"));
}
```

(c) Migrated test in Mylogin-Htmlparser

```
TEST(test, findsCharsetInMalformedMeta) {
    string h = "<meta http-equiv=Content-Type content=
        text/html; charset=gb2312>";
    parser parser;
    node doc = parser.parse(h);
    ASSERT_EQ("gb2312", doc->select("meta")->get_attr("charset"));
}
```

Figure 1: An example of unit test migration from Jsoup to Mylogin-Htmlparser.

MUT facilitates developers in generating unit tests and supports the C++ language. For instance, when developing a lightweight HTML parser module, developers can find well-known projects with similar functionalities in the open-source community. MUT migrates function-related tests from these projects to the current development project, thereby generating unit test code. Although the migrated unit tests generated by MUT may only be partially correct, they still reduce the effort required for developers to write unit tests from scratch. Therefore, MUT helps developers improve the efficiency of writing unit tests.

2.2 Task Definition

This work primarily focuses on the migration of unit tests. The task can be formalized as follows: (1) Given the source code of two projects, s and s', the goal is to find a function f such that $f(s, s') = r$. We refer to s, s', and r as the production code snippets of the original project, the production code snippets of the target project, and the code mapping rules, respectively. We calculate the code similarity between the two projects to obtain f. (2) Given r and the unit tests of the source project, x, the objective is to find a function g such that $g(r, x) = y$. We refer to x and y as the original unit test and the target unit test, respectively. We utilize a code translation approach to obtain g.

3 PROPOSED APPROACH

Figure 2 illustrates the overall framework of our approach. The migration process mainly has four steps:

Step ❶ Code mapping. This step aims to find the corresponding production code snippet C_s with the same functionalities in the source projects given code C_t in the target projects.

Step ❷ Test case filtering. This step aims to find the unit test case T_s of code snippets C_s in the source project. These unit test cases are candidates to be migrated.

Step ❸ Test case translation. This step translates the test cases in the source projects into those in the target projects. Specifically, the test cases in the source projects are written in Java programming language with JUnit [15] test framework. These test cases will be translated into the C++ programming language and executed using the most popular test framework in C++, GTest [9].

Step ❹ Migrated test case adaptation. After the translation process, the generated test cases may not be used directly. Thus, we should adapt them to facilitate compilation and execution within the target project.

3.1 Code Mapping

As illustrated in Figure 2, during the Code Mapping stage, MUT identifies potential code mapping relationships between the two projects and concludes them into code mapping rules.

3.1.1 Code Data Extraction. During the code data extraction stage, MUT extracts basic code information (e.g., class names, method names, method parameters, return types, and documentation) from both the source project and the target project. We use ANTLR [1] to facilitate the extraction process of code written in different programming languages. ANTLR provides a straightforward syntax rule language, commonly utilized for parsing tasks in various programming languages. The extracted information is used to find code mapping relationships between two projects, which serve as the basis for migrating test cases from the source project to the target project.

3.1.2 Preprocessing. Due to the significant differences in code styles between different programming languages, we then preprocess the extracted code information to mitigate the impact of code style variations. Specifically, naming styles for class names and method names often follow composite forms, such as camel case or snake case. We split them into word sequences, for example, `startObject` and `start_object` are split into *start*, *object*. Then, we convert them into lowercase and eliminate irrelevant stop words (such as *a*, *an*, and *the*). Furthermore, each word is transformed into its stem form, for instance, *equals* is transformed to *equal*. For example, the method name `selectElement` is preprocessed into *select* and *element*, class name `HtmlParse` is preprocessed into *html* and *parse*, respectively.

3.1.3 API Mapping. An API [2] represents the fundamental unit for implementing functionality in a system module, and each public method within a class is defined as an API. Different software projects might include code modules with similar or identical functionality, but the code implementation logic, style, and syntax of each API can vary significantly, particularly in cross-language and cross-platform contexts. We design a series of strategies to reveal

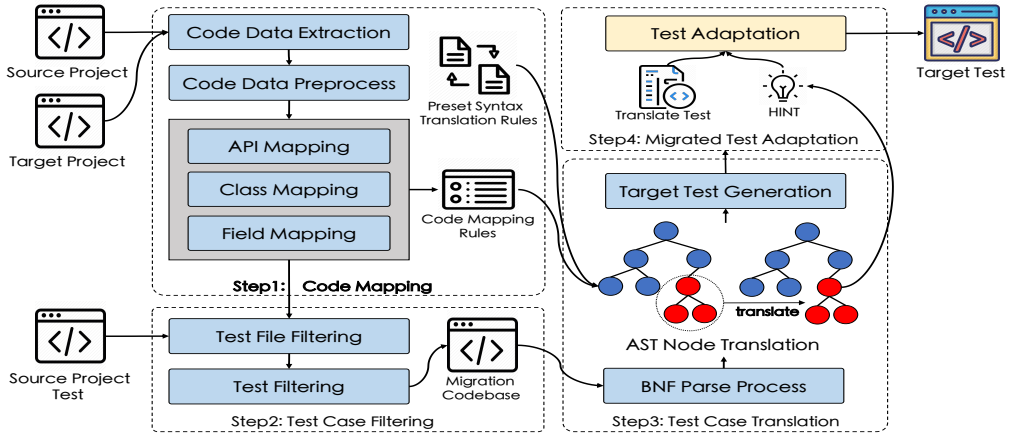

Figure 2: Overview of our Approach.

potential API mapping relationships, including token similarity, semantic similarity, and documentation similarity. These strategies explore API mappings between different production code snippets in terms of token and code semantic similarity, with a specific API represented as a combination of class name sequences and method name sequences, i.e., `api = ClassNameSequence $ MethodNameSequence`. The following section elaborates on the API mapping in detail.

a. Token Similarity: The token similarity mapping identifies candidate API mapping pairs when two APIs have identical or similar method names. To achieve this, we first input the sequences and utilize the *Word2Vec* [22] technique to transform them into vectors. Next, we calculate the sum and mean of the vectors within each set, deriving the corresponding API vectors. Subsequently, the cosine similarity between two API vectors is determined, if the similarity exceeds a predefined threshold, the APIs are considered a candidate mapping pair. In this paper, the threshold is set as 0.95 since we empirically find it achieves the best performance.

b. Semantic Similarity: This step aims to recognize API mapping pairs that possess semantic associations. Due to variations in naming styles across systems and platforms, APIs with similar functionalities might exhibit significant token differences, limiting the effectiveness of obtaining mappings based on token similarity.

For JSON processing libraries, an API named `startObject` is typically used to initiate the creation of a JSON object. It provides an entry point that facilitates the addition of key-value pairs or fields to the object. In another project, an API with a similar functionality might be called `beginObject`. Although the API names differ in the token, they are semantic-related. Therefore, we design a semantic-related mapping method to obtain the similarity between APIs.

Figure 3 illustrates the process of obtaining API mapping pairs by calculating semantic-related similarity. Specifically, we divide this process into two steps:

Step ❶: Calculate the semantic similarity between two tokens. We utilize *WordNet* [23] to identify synsets for each token in the API sequence. Then, we identify the two most semantically similar words from the synsets and calculate their similarity as the similarity between the two tokens.

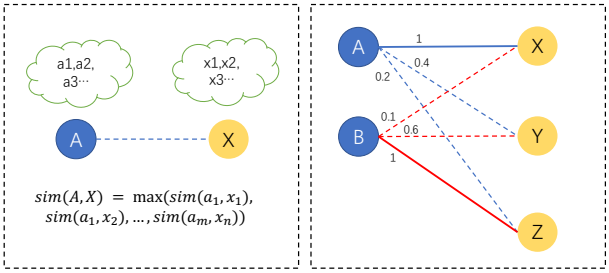

Figure 3: An example of semantic-related similarity.

Step ❷: Calculate the semantic similarity between two token sequences. Since an API name is composed of token sequences, we construct a bipartite graph with the tokens as nodes. The weights of the edges between the nodes are the token similarities calculated in Step 1. We then apply the *KM* [16] algorithm to find the maximum weighted matching in the bipartite graph. The weight of this matching in the graph serves as the similarity between the two token sequences. Finally, we set a threshold to determine whether the two token sequences form an API mapping pair.

c. Documentation Similarity: We can utilize API documentation to describe the functionality and usage of an API, which enables us to discover API mappings between two projects that use different programming languages. Generally, API documentation with similar or related functionalities exhibits semantic correlations. We remove semantically irrelevant identifiers in API documentation, such as `@param` and `@return`. Subsequently, similar to the semantic similarity, we employ *Word2Vec* to convert documents into vector representations and calculate the cosine similarity between the vectors. Then, we obtain candidate API mapping pairs based on this similarity.

3.1.4 Class Mapping. We construct class mappings based on API mapping relationships. As illustrated in Figure 1 (refer to Sec 2), the pair ⟨ `Jsoup.parse`, `parser.parse` ⟩ denotes an API mapping. We then establish a class mapping among the associated classes, namely

```
@Test
public void testIteratorUpdateable() {
    Attributes a = new Attributes();
    Iterator<Attribute> iterator = a.iterator();
    Attribute attr = iterator.next();
    attr.setKey("Foo");
    attr = iterator.next();
    attr.setKey("Bar");
    attr.setValue("Qux");

    assertEquals("a&p", a.get("Foo"));
    assertEquals("Qux", a.get("Bar"));
}
```

Figure 4: A test case from Jsoup.

⟨ `Jsoup`, `parser` ⟩. During the translation of the test case, we replaced the `Jsoup` class with `parser` based on the class mapping. Similarly, the pair ⟨ `Document.text`, `node.to_text` ⟩ denotes another API mapping from which the class mapping pair ⟨ `Document`, `node` ⟩ is derived. As a result, the `Document` class is ultimately replaced by the `node` class in this example. In addition, class mappings can exhibit a one-to-many relationship. In the translation process, the target class is determined based on the pre-existing API mapping relationships within the specific test case.

3.1.5 Field Mapping. Test cases often employ fields in a specific class as preconditions prior to conducting functional testing. The test case depicted in Figure 4 aims to evaluate the update functionality of HTML tag attributes in the Jsoup [14]. The `Attribute` class includes two fields: `key` and `value`. This test case validates whether the `attr` object accurately retrieves the key and value after field configuration. If we need to migrate this test case to Mylogin-Htmlparser [20], it is necessary that the two fields should also exist correspondingly in Mylogin-Htmlparser. Thus, it is essential to establish a mapping relationship between the fields in both projects. Since both Java and C++ programming languages exhibit object-oriented characteristics and their encapsulation feature requires accessing fields through getter and setter methods, in this step, we establish a set of field mapping pairs between the two projects by examining the consistency of getter and setter methods with the same names. For example, in the case of the `setKey` and `setValue` methods in the Jsoup `Attribute` class, we will locate the corresponding method with the same name in the target project and establish field mapping pairs.

Now, we obtain the mapping pairs between the production code of the two projects. These mapping pairs serve as guidelines for code translation and are stored as configuration files, allowing for convenient modification and adaptation. Moreover, to adapt the substitution of test statements across diverse testing frameworks, we establish a predefined collection of syntax replacement rules. Table 1 shows some mapping rules between the Java unit testing framework JUnit [15] and the C++ unit testing framework GTest [9]. For instance, automatically substituting JUnit's `assertTrue` with GTest's `ASSERT_TRUE`. These rules are utilized to substitute the source code with the target code during the subsequent migration of test cases.

Table 1: Test Mapping Rules: JUnit and GTest Framework.

JUnit	GTest
assertTrue	ASSERT_TRUE
assertFalse	ASSERT_FALSE
assertEquals	ASSERT_EQ
assertNotEquals	ASSERT_NE
assertNull	ASSERT_TULL

3.2 Test Case Filtering

In the second stage, MUT identifies the test cases from the source project suitable for migration. We first identify the appropriate test files, and then we identify the code elements (classes, methods, fields) associated with the test cases.

Test File Filtering: For each class file in the source project, we first identify its focal test files. It is worth noting that the naming style for test files corresponding to class files is not always strictly standardized, thus requiring us to widen the scope of regular expression matching. A heuristic algorithm is employed to match test files using the following regular Expression: `Test[a-zA-Z] ClassName[a-zA-Z] (Test|Tests)?`, where `ClassName` represents the specific class name. A class may match multiple test files, for example, the `Date` class may match test files such as `TestDate`, `TestLocalDate`, and `TestDateTime`. These test files are further filtered based on API call relationships in the subsequent step.

Test Filtering: In the previous step, we filtered out the test files required for migration. It is important to note that not all test cases in the test file need to be migrated to the target project. In this step, based on the existing API mapping relationships, we further filter out the test cases in the test file that need to be migrated. To ensure that the migrated test cases can evaluate and verify the API functions in the target project, we set two filtering rules. First, the test cases in the test file need to be tested to the API defined in the focal class, which is judged according to the analysis of the call relationship between the test cases and the API. Second, the API tested in the test case is required to have a mapping relationship in the target project. At this stage, all the structures to be migrated, including classes, fields, and test methods, are identified.

3.3 Test Case Translation

In this stage, MUT employs the translation component and code mapping rules to progressively translate source code elements into the target code elements.

BNF Parse Process: In this step, MUT conducts syntax analysis and gathers the parsing results for the test cases designated for migration, which is utilized in the subsequent translation process. A test case can encompass diverse syntax structures, including class and field declarations, conditions, loops, method invocations, etc. Syntax analysis is essential for translating these structures from the source language to the target language, and we use ANTLR [1] to recognize and parse these syntax structures. The Java syntax rules defined in ANTLR establish mappings between each syntax structure and a corresponding BNF node object. Throughout this parsing process, we collect all the different types of BNF nodes present in the test cases, which serve as the translation targets.

```
methodDeclaration
    :methodModifier* methodHeader methodBody ;
methodBody
    :block
    |';' ;
block
    :'{' blockStatements? '}' ;
blockStatements
    :blockStatement+ ;
blockStatement
    :localVariableDeclarationStatement
    |classDeclaration
    |statement ;
```

Figure 5: The BNF Rules Generated by ANTLR for Java Method Declarations.

As shown in Figure 5, when parsing the test case, the declaration of the test method corresponds to the BNF node *MethodDeclaration*, and the corresponding node object *MethodDeclarationContext* can be obtained by parsing the test method. From this BNF node object, we can extract all the information related to the method declaration, such as the method name, return value type, parameter list, and each block statement in the method body. In the subsequent step, we systematically translate this information into the target language and structure it into new test cases.

BNF Node Translation: In the prior step, we obtain BNF node objects corresponding to each test case for migration. This stage involves parsing these node objects and executing the source-to-target code translation. To illustrate this translation process, let's consider the translation of a method. The specific node object corresponding to a method is called *MethodDeclarationContext*, essentially a syntax tree structure, the BNF syntax rule depicted in Figure 5 specifies that its child nodes comprise *MethodModifier*, *MethodHeader*, and *MethodBody*, signifying method modifiers (e.g., `public`, `private`), method headers, and method bodies, respectively.

The *Translate* component within MUT is tasked with parsing and converting these BNF node objects. It accepts the BNF nodes of the source code as input and produces target code elements as output. The fundamental function of *Translate* involves adjusting a syntax tree by replacing, adding, or eliminating child nodes, and subsequently converting it into the target structure. Given the 243 distinct BNF node types in Java syntax [1], to recognize, parse, and translate all possible syntax structure types that may be encountered in the source project, we establish a corresponding *Translate* for each BNF node type to facilitate the source-to-target translation. To ensure the translated test is executable within the target project, the *Replace* component replaces code according to the code mapping rules and pre-set syntax replacement rules (see Figure 2). By extending these two components, we further enhance the MUT tool's capability to facilitate translations between additional programming languages.

3.4 Migrated Test Adaptation

Finally, the migrated test is compiled and executed within the target project. However, ensuring the executability of the migrated test (e.g., from Java to C++) poses a significant challenge [25, 30, 32]. Consequently, addressing specific issues, such as adding missing dependencies and modifying unmapped code segments, is crucial for the successful compilation and execution of the migrated test within the target project.

To facilitate this process and enable manual adaptation on the migrated test, we developed a web tool that offers various beneficial features. As shown in Figure 6, developers can visualize the code before and after migration via the web interface, allowing for quick comparison and assessment of the translation results. Furthermore, when mapping relationships are absent, MUT provides hints to emphasize specific code that failed to convert, empowering developers to address these concerns more effectively.

Figure 6: An example of the web interface of MUT.

By utilizing the MUT web framework, developers can streamline the test adaptation process, identify and resolve potential issues more efficiently, and ultimately improve the overall quality and reliability of the migrated test.

4 EXPERIMENTS

In this section, we discuss the experimental details of our tool and address the following three research questions:

- RQ1: How is the quality of code mapping in MUT for the test migration task?
- RQ2: How useful is the MUT tool in practice?
- RQ3: How effective are the migrated tests in the target project?

4.1 Experimental Setup

As MUT aims to migrate unit test cases from Java to C++ open-source projects, we select our test migration dataset from widely used open-source projects in both Java and C++ languages. We choose Java projects that are either popular or highly ranked based on GitHub stars. Such projects exhibit more comprehensive unit tests, higher code writing standards, and enhanced quality, making them suitable for source code selection in test migration. When selecting C++ projects for the test migration experiment, we consider categories that are similar to Java projects, and we randomly select some newly developed, lightweight C++ projects. These projects typically lack sufficient test cases, making them suitable candidates for our test migration objectives.

In total, 15 open-source projects are selected as experimental subjects, which are shown in Table 2. Five of these projects are in Java and 10 are in C++. To ensure functional diversity, our selected projects cover five categories, including string processing, data structures and algorithms, HTML manipulation, date processing, and JSON processing. These categories host popular projects with active developer communities, facilitating the acquisition of suitable tests for migration.

Table 2: Subject projects for MUT's evaluation.

Category	Open Source Projects	
	Source Project	**Target Project**
String	Commons-lang [5]	Libstrop [17] Boost algorithm [3]
DataStructure	Fishercoder [8]	Luliyucoordinate-Leet-Code [18] Data-Structure-and-Algorithms [7]
HTML	Jsoup [14]	Rangerlee-Html-Parser [21] Mylogin-Html-Parser [20]
Date	Joda-Time [11]	CPP-DateTime-library [6] Boost Date [4]
JSON	Minimal-json [19]	JsonParser [12] Jsontest [13]

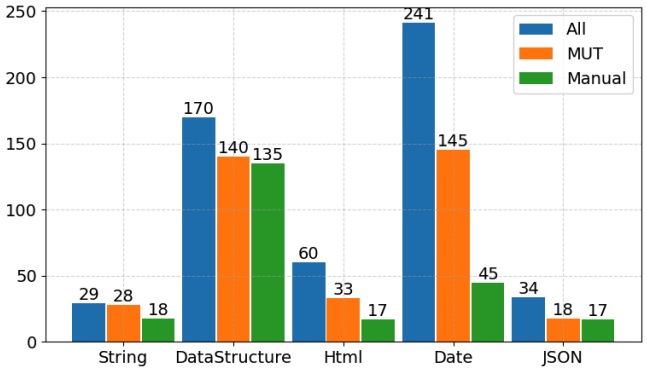

Figure 7: Results for API Mapping. all - total number of mappings obtained, mut - number of mappings after filtering with MUT, manual - number of mappings after manual selection.

4.2 RQ1: The Effectiveness of Code Mappings

In the first part of our experimental evaluation, we concentrate on the efficacy of code mapping generated by the MUT tool. During this stage, we identify all potential similarities and correlations between the source and target projects, which facilitate the creation of code mapping rules. API mapping is critical during the code mapping process. As elaborated in Section 3, API mapping assists in establishing class mapping relationships and field mapping relationships. Furthermore, the quality of API mapping directly influences the effectiveness of later stages, such as code translation. A test case is considered eligible for migration only if it utilizes the mapped API, thereby validating the functionality of the API.

As depicted in Figure 7, MUT automatically obtains 534 API mappings from five test migration projects, of which 364 API mappings are related to tests that invoked the mapped APIs in the source project. We manually verify all API mapping results for each dataset by inspecting the source code or documentation of the APIs. The experimental results reveal that the performance of API mapping varied considerably among different experimental groups. The proportion of finally selected API mappings is 63.7%. The highest performance is observed in the DataStructure dataset at 96.4%, while the Date dataset exhibits the lowest performance at a mere 31%.

We present examples illustrating the effectiveness of MUT's code mapping and explain the reasons behind the manual adaptation of specific mappings. For instance, in the JSON dataset, MUT successfully identifies API mapping pairs that are functionally identical, such as <JsonObject-isObject, JsonObject-IsObject>. Moreover, MUT recognizes semantically related API mappings with analogous functionality, such as <JsonObject-startObject, JsonObject-beginObject> and <JsonObject-stopObject, JsonObject-endObject>, which are employed to mark the starting and ending positions of a JSON object during its creation.

These findings suggest that MUT is capable of identifying API mapping relationships that are related in terms of tokens or semantics across diverse programming languages and projects. However, MUT also produces inaccurate mapping pairs. As MUT's code mapping relies on token or semantic correlations between APIs, APIs with some form of association will be chosen as candidate mapping pairs. For example, in the Date dataset, Joda-time [11] and date-boost [4] both feature the API `year`, used to obtain the year of a time object. MUT accurately maps them as a candidate API mapping pair, which is the expected correct mapping relationship. Nevertheless, the `Years` class in Joda-time also possesses a `years` method that is similar to `year`. MUT maps them as a candidate API mapping pair, but they exhibit different functionalities. Consequently, this API mapping pair is invalidated. Similar situations are prevalent in the Date dataset, resulting in the retention of only 45 out of 145 accurate API mapping pairs.

In summary, after manually verifying the mapping outcomes, we select 232 API mappings from the 364 API mapping results to serve as the basis for subsequent steps.

4.3 RQ2: The Effectiveness of Tool Usage

To address RQ2, we initially introduce the parts of MUT that require manual intervention during its usage. Subsequently, we evaluate the tool's effectiveness through a user study. Currently, the utilization of MUT necessitates manual intervention during three parts: code mapping selection, test case selection, and adaptation of migrated test cases to the target project. All of these adjustments can be executed directly on the web interface provided by MUT, eliminating the need to access source code from code files, thereby rendering the tool highly convenient to use.

Table 3 illustrates the time distribution for these three stages of the migration process across the five experiments. As evidenced in the Table 3, the manual time required for code mapping and test method selection is relatively brief, with total durations of 17 and 26 minutes, respectively. Experimenters merely need to remove obviously incorrect mappings and unsuitable test methods for migration to the target project via the web interface. The most time-consuming aspect involved adapting the migrated tests to the target system environment, with a total manual duration of 119 minutes. Throughout the experiment, a total of 550 test cases

Table 3: Measurements of manual effort spent on different stages of the migration process, including mapping selection (#MS), test selection (#TS), and test adaptation (#TA).

Category	#MS[min]	#TS[min]	#TA[min]
String	2	3	15
DataStructure	5	3	25
HTML	3	3	20
Date	3	7	33
JSON	4	10	26
ALL	17	26	119

are migrated, averaging a manual time of merely 18 seconds per migrated test case.

```
@Test
public void cssEscapedAmp() {
    Document doc = Jsoup.parse("<p class='\\&'>One</p>"
        );
    Element one = doc.expectFirst(".\\\\\\&"); //
        tested matches js querySelector
    assertEquals("One", one.text());

    String q = one.cssSelector();
    assertEquals("html > body > p.\\\\\\&", q);
    assertEquals(one, doc.expectFirst(q));
}
```

Figure 8: An example of Unit Test Case Migration from Jsoup to Mylogin-HtmlParser.

During MUT's second stage, the tool identifies candidate test cases from all test files in the source project that cover APIs with mapping relationships, providing functional test coverage and validation for the corresponding APIs. As shown in Figure 9, MUT selects a total of 1,103 candidate migration tests across five category projects. However, not every candidate can be successfully migrated to the target project and pass compilation and execution. There is a scenario in which candidate tests must be filtered out: although the test cases utilize APIs that have mapping relationships, the source test cases may contain functional features that are not present in the target project. We utilize the HTML project as an example to illustrate this, as shown in Figure 8, the `cssEscapedAmp` test case is for the Jsoup [14], and the test case verifies whether the `expectFirst()` method can accurately handle escape characters in class names and if the `cssSelector()` method can convert an element's CSS selector to a string. Although the target project also has similar functionality for Jsoup parsing and simple selector, the `cssSelector()` method is not present in the target project to translate an element's CSS selector to a string. Consequently, this test case cannot be migrated and used in the target project.

Therefore, a manual inspection of these test cases is required, and this process can be accomplished through the filtration functionality offered by the web interface of the MUT tool. If a functional API in the test is not present in the target project, these test cases must be removed, and subsequent migrations will not include them. In total, after manual inspection and filtering, we select a total of 550 test cases for all target projects.

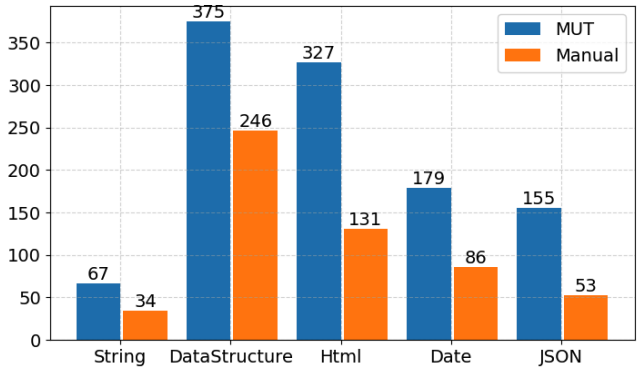

Figure 9: Results for Test Case Filtering. mut - number of test cases filtered by MUT, manual - number of tests confirmed by a manual check.

Due to the unavailability of an open-source and suitable C++ test generation tool as a baseline, we conduct a user study to evaluate the effectiveness and efficiency of the MUT tool in test migration. The study involves three doctoral students and three undergraduate students, all of whom had a basic year's experience in C++ and Java at least 5. To conduct the user study, we randomly select 20 tests from five Java projects. To ensure that the participants understood how to construct unit tests in the target project, we provide them with one day to familiarize themselves with the usage of each target project in advance. The participants are randomly divided into three groups: Group A, Group B, and Group C, with two participants in each group. Within each group, the two participants share the task of writing tests equally, and each group is required to complete the task of writing 20 test cases. Group A is given basic tips about which APIs required testing, and they are tasked with writing tests from scratch. Group B is provided with the source code for the tests from the source project but without any prompts. They are instructed to write the tests for the target project based on the original tests. Group C uses the MUT tool to write the tests.

Table 4: Result for the User Study of MUT.

Group	Total[sec]	Avg[sec]	Test Accuracy
Group A	1967	98	75
Group B	1496	75	85
Group C	**520**	**26**	**100**

By referring to Table 4, we observe that Group A takes the longest time to write tests, averaging 98 seconds to construct a single test. Additionally, compared to the other groups, Group A has the highest error rate, with 5 tests that could not be compiled and executed successfully. Group B, benefiting from the availability of source tests as references, requires relatively less time, with an average of 75 seconds per test. Additionally, Group B demonstrates an improved accuracy rate compared to Group A, with an accuracy rate of 85%. However, the most efficient is Group C, which, with the assistance of the MUT tool, averaged only 26 seconds of manual effort per test. The time efficiency for each group can be attributed

to the varying degrees of effort required. Group A needed to design test cases and then write the code from scratch. Group B, having a reference, saved some time by not needing to design test cases. However, as the source language is Java, they needed to consider the code mapping relationships between the source and target and manually translate the source code into a test case. Group C, on the other hand, benefited from the automatic API replacement in the MUT tool and the translation to the target testing framework, which saved considerable time. Their task is simply to rewrite any missing parts based on the provided hints.

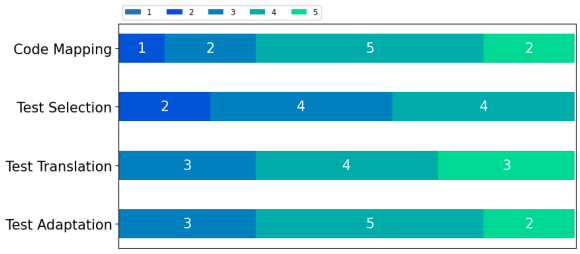

Figure 10: Participant Ratings of MUT Tool's Effectiveness Across Four Stages.

We carry out an assessment to gain insight into the effectiveness of the MUT tool across its four stages using Likert-scale diagrams (Figure 10). Participants are instructed to rank each stage on a scale ranging from one (poor) to five (excellent). In the code mapping stage, three participants are assigned an excellent rating, while four are designated a good rating, suggesting the proficiency of our tool in identifying code mappings—a vital aspect in the preliminary stage of the migration process. In the test selection stage, seven participants grant a good rating, and three adjudges it as average, indicating satisfactory performance in selecting an appropriate test case from the source project for migration.

During the test translation stage, four participants attribute favorable ratings, while four categorize them as average, and two as poor. Despite the inherent complexity of this stage, the tool's capability to translate test cases into the target language is generally appreciated, although with potential for further refinement.

In the test adaptation stage, seven participants award a good rating, and three deem it as average. The results of the user study confirm the efficacy of the MUT tool across its principal operational stages. The tool exhibits commendable performance in the code mapping and test case filtering stages, with areas for improvement identified in the test transformation stages. The findings of this study inform future enhancements to the MUT tool, particularly concerning its capacity to manage test translation and adaptation.

4.4 RQ3: Usefulness of Migrated Tests

To assess the efficacy of the test case migrated by MUT, we first execute all migrated tests. This process entails compiling the migrated code within the target projects to verify its correctness and applicability. Subsequently, we run these tests to perform functional testing of the target project, analyze the test outcomes, identify issues (bugs, functional defects, etc.) encountered during testing, and report them to the target project maintainers via GitHub issues. Lastly, we submit the migrated tests as separate GitHub pull requests to the maintainers of the target projects.

1) Pull Request Table 5 presents detailed information regarding the test case migration for each group of projects. The Source Project and Target Project columns indicate the names of the source and target projects for the migrated test case, respectively. The *#Tests* column displays the number of tests that are migrated and compiled for each group of projects, while the *#PRac* column denotes the current acceptance status of our migrated tests.

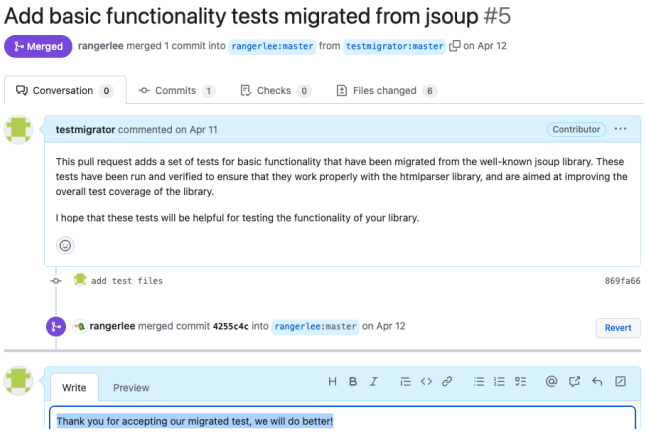

Figure 11: One of the Accepted Pull Requests for migrating test cases.

A checkmark signifies that the target project developers have approved the migrated test and merged it into their projects, whereas a dash indicates that the pull request is pending review by the target project developers and has not been rejected yet. As depicted in Table 5, developers of two target projects have responded, acknowledging the usefulness of these tests for maintaining their existing projects, and ultimately accepted our migrated tests. As shown in the Figure 11, the developer of Rangerlee-HtmlParser has accepted our migrated tests. As of the submission of this paper, out of the 550 test cases we submitted, 253 test cases have been accepted, accounting for 46% of the total. The remaining test cases are currently under evaluation by the developers.

2) Issue Report Besides augmenting the testing capabilities of the target project, another goal of migrating tests is to uncover potential issues within the target system, such as functional defects or code bugs. We identify seven issues in our experiments across the five categories of datasets, as illustrated in Table 6. The example in section 2 refers to a bug detected in the Mylogin-HtmlParser library by our migrated test, and Figure 12 is the feedback from the developer of the library, who confirmed the bug and fixed it accordingly. In addition to discovering bugs, we also observe that some API functions in the target project are not fully developed. For instance, in the HTML category experiment, the `parse` API is utilized to parse HTML elements. However, we notice that in some tests migrated from Jsoup to Mylogin-HtmlParser, for certain uncommon HTML elements like `frameset`, Mylogin-HtmlParser could not successfully parse them as HTML elements but instead

Table 5: Results for Migration testing: #Tests - the total number of migrated and executed tests; #PRac - the acceptance status of the pull requests submitted for the migrated tests on Github, where a checkmark indicates acceptance by the maintainer.

Source Project	Target Project	#Tests	#PRac
Commons-lang	Libstrop	11	-
	Boost algorithm	23	-
Fishercoder	Luliyucoordinate-Leet-Code	197	✓
	Data-Structure-and-Algorithms	49	-
Jsoup	Rangerlee-HtmlParser	56	✓
	Mylogin-HtmlParser	75	-
Joda-Time	CPP-DateTime-library	26	-
	Boost Date	60	-
Minimal-json	JsonParser	19	-
	Jsontest	34	-

Table 6: Issues discovered in the process of migration testing in the target projects.

Category	Issue Report		
	Bugs	Functional Defects	Total
HTML	2	2	4
Date	1	0	1
JSON	2	0	2
ALL	5	2	7

as regular strings. Consequently, we raised an issue concerning the incomplete functionality of Mylogin-HtmlParser and presented the comprehensive test. We furnished detailed descriptions of the identified issues and offered suggestions for rectifying them in all of our submitted issues.

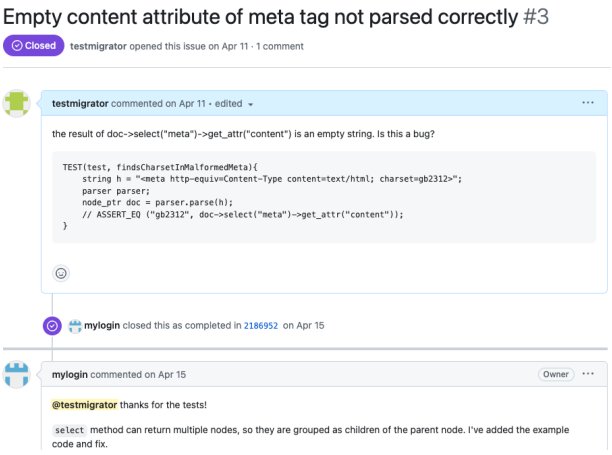

Figure 12: The select API bug detected by the migrated test was confirmed and fixed by the Mylogin-HtmlParser developer.

In summary, our experimental results demonstrate that the MUT tool can effectively migrate test cases, and the migrated test cases can effectively test the functional correctness of the target project, aiding not only in maintaining the existing code library of the target project but also in detecting potential issues. In our experiments, the migrated test cases are not only approved by the target project developers but also contributed to resolving bugs.

5 THREATS TO VALIDITY

The primary concern regarding the external validity of our tool is whether it can be generalized to other projects, tests, and categories. To address this concern, we selected real-world projects from five different categories using a random sampling approach. A potential threat to internal validity arises from the fact that some users who wrote the tests used in our evaluation were not familiar with the projects under test. However, developers are generally knowledgeable about the projects they develop themselves. Another threat to internal validity is the possibility of errors in the manual inspection and adaptation of the test migration results. Despite the differences in code design and implementation between two projects within the same category, they share common functionality, making the inspection and adaptation feasible. We aim to compare Fuzzer-based methods, including test generation tools like Randoop and Evosuite, with our tool MUT. However, we encounter a limitation as these existing tools lack support for C++ language testing generation. As a result, we are unable to conduct direct comparative experiments.

6 RELATED WORK

A considerable body of research has been devoted to test generation techniques [26, 36, 37, 39, 47, 51, 55, 58, 61], such as model-based methodologies [47, 55, 58], Fuzz testing approaches [37, 39, 61], and specification-based strategies [26, 53, 56, 59], primarily focusing on revealing data and logical errors. Besides automatic test generation, test migration has emerged as an effective technique for reusing tests [27, 28, 35, 41, 45, 50, 54, 60], leading to the development of various tools and methods. TestMig [50] is a migration tool designed to transfer tests from iOS to Android, implemented using UI events and requiring source code from both source and target applications. MAPIT [54] presented a cross-platform test migration approach based on user interactions with the UI, eliminating the need for source code and facilitating the migration of oracle events and system events among apps with similar functionalities. Mariani et al. [46] framed the test reuse problem as a search challenge, employing evolutionary testing to achieve test migration across different Android applications. The test transplantation technique facilitates the reuse of test cases. Abdi et al. [24] conducted program slicing on tests from the project's dependent libraries, extracting test inputs, and isolating them by creating mocks. Subsequently, they transplanted the test code into the target project. Lima et al. [40] applied the technique of test transplantation to various JavaScript engines, transplanting a test suite from one engine to others. This approach aided in the detection of potential issues and vulnerabilities in the different engines.

We systematically study the existing studies on code translation [25, 29, 32, 43, 49], and find the research scope is limited within translating cross-language projects for the same application. They

often struggle to handle complex semantic functions and ensure consistency and compatibility post-conversion, which hampers their widespread application. Java2C# [30], created using TXL, a language explicitly designed for program transformation that employs tree-based rewriting. TXL is a functional rule-based language that accepts arbitrary context-free grammars in Extended BNF (EBNF) notation as input and applies a collection of transformation rules to the input program, demonstrated through examples. Java2CSharp [10] utilizes manually defined mappings and rules to transform Java code into C#.

7 CONCLUSION AND FUTURE WORK

This paper proposes a unit test code migration method, which builds a test migration process based on the source code of projects with functional intersections. Our work demonstrates that it is feasible to migrate unit test source code across languages and platforms, by transferring high-quality test cases from well-known projects to the target project to discover functional defects and bugs, and to serve as test code for the target project. In future work, we intend to explore the effectiveness of MUT's test migration method in a broader range of domains within the open-source community, validating it with a more extensive selection of projects and migrating a larger number of test cases. Additionally, we aim to extend the applicability of our method to support migration between more programming languages, such as Java to Python, and beyond.

ACKNOWLEDGMENTS

This research is supported by the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), National Natural Science Foundation of China (No. 62141222), and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] 2023. ANTLR. http://www.antlr.org.
[2] 2023. API. https://en.wikipedia.org/wiki/API.
[3] 2023. boost-algorithm. https://github.com/boostorg/algorithm.
[4] 2023. Boost-DateTime. https://github.com/boostorg/date_time.
[5] 2023. commons-lang. https://github.com/apache/commons-lang.
[6] 2023. CPP-DateTime-library. https://github.com/jeremydumais/CPP-DateTime-library.
[7] 2023. Data-Structure-and-Algorithms. https://github.com/Keshav0907/Data-Structure-and-Algorithms.
[8] 2023. Fishercoder. https://github.com/fishercoder1534/Leetcode.
[9] 2023. Gtest. https://github.com/google/googletest.
[10] 2023. Java2CSharp. http://sourceforge.net/projects/j2cstranslator/.
[11] 2023. joda-time. https://github.com/JodaOrg/joda-time.
[12] 2023. JsonParser. https://github.com/mindflower/JsonParser.
[13] 2023. jsontest. https://github.com/josexy/jsontest.
[14] 2023. Jsoup. https://github.com/jhy/jsoup.
[15] 2023. JUnit. https://github.com/junit-team/junit4.
[16] 2023. KM. https://encyclopedia.thefreedictionary.com/Hungarian+algorithm.
[17] 2023. libstrop. https://github.com/nicmcd/libstrop.
[18] 2023. Luliyucoordinate-LeetCode. https://github.com/luliyucoordinate/Leetcode.
[19] 2023. minimal-json. https://github.com/ralfstx/minimal-json.
[20] 2023. Mylogin-HtmlParser. https://github.com/mylogin/htmlparser.
[21] 2023. Rangerlee-HtmlParser. https://github.com/rangerlee/htmlparser.
[22] 2023. Word2Vec. https://zh.wikipedia.org/wiki/Word2vec.
[23] 2023. WordNet. https://wordnet.princeton.edu/.
[24] Mehrdad Abdi and Serge Demeyer. 2022. Test Transplantation through Dynamic Test Slicing. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 35–39. https://doi.org/10.1109/SCAM55253.2022.00009
[25] Vicki H Allan and X Chen. 2001. Convert2Java: semi-automatic conversion of C to Java. *Future Generation Computer Systems* 18, 2 (2001), 201–211.
[26] Yusuke Aoyama, Takeru Kuroiwa, and Noriyuki Kushiro. 2020. Test case generation algorithms and tools for specifications in natural language. In *2020 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 1–6.
[27] Farnaz Behrang and Alessandro Orso. 2018. 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*. 164–175.
[28] Farnaz Behrang and Alessandro Orso. 2019. Test migration between mobile apps with similar functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 54–65.
[29] Santiago Bragagnolo, Stéphane Ducasse, Nicolas Anquetil, Abderrahmane Seriai, and Mustapha Derras. 2022. Alce: Predicting Software Migration. (2022).
[30] James R Cordy, Thomas R Dean, Andrew J Malton, and Kevin A Schneider. 2001. Software engineering by source transformation-experience with TXL. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 168–178.
[31] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. 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*. 57–67.
[32] Mohammad El-Ramly, Rihab Eltayeb, and Hisham A Alla. 2006. An experiment in automatic conversion of legacy Java programs to C. In *IEEE International Conference on Computer Systems and Applications, 2006*. IEEE, 1037–1045.
[33] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419.
[34] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.
[35] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. 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*. 269–282.
[36] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. 2019. A survey on adaptive random testing. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2052–2083.
[37] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
[38] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. 2022. POWER: Program option-aware fuzzer for high bug detection ability. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 220–231.
[39] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218. https://doi.org/10.1109/TR.2018.2834476
[40] Igor Lima, Jefferson Silva, Breno Miranda, Gustavo Pinto, and Marcelo D'Amorim. 2020. Exposing Bugs in JavaScript Engines through Test Transplantation and Differential Testing. (2020).
[41] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test transfer across mobile apps through semantic mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–53.
[42] Xiangjun Liu and Ping Yu. 2022. Randoop-TSR: Random-based Test Generator with Test Suite Reduction. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*. 221–230.
[43] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 727–739.
[44] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated unit test generation for python. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*. Springer, 9–24.
[45] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic matching of gui events for test reuse: are we there yet?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 177–190.
[46] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. 2018. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In *Proceedings of the 40th International Conference on Software Engineering*. 280–290.
[47] Muhammad Luqman Mohd-Shafie, Wan Mohd Nasir Wan Kadir, Horst Lichter, Muhammad Khatibsyarbini, and Mohd Adham Isa. 2021. Model-based test case generation and prioritization: a systematic literature review. *Software and Systems Modeling* (2021), 1–37.

[48] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 544–547.

[49] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–596.

[50] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. Testmig: Migrating gui test cases from ios to android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 284–295.

[51] S Rani and Amandeep Kaur. 2020. A literature survey on automatic generation of test cases using genetic algorithm. *Wesleyan J. Res.(UGC Care Listed)* 13, 2 (2020), 65–76.

[52] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2020. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 287–298.

[53] Yuji Sato. 2020. Specification-based test case generation with constrained genetic programming. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 98–103.

[54] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. 2021. UI test migration across mobile platforms. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 756–767.

[55] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).

[56] Rong Wang, Yuji Sato, and Shaoying Liu. 2019. Specification-based Test Case Generation with Genetic Algorithm. In *2019 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1382–1389.

[57] Cerdic Wei Kit Wong. 2022. American fuzzy lop (AFL) fuzzer. (2022).

[58] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of the 44th International Conference on Software Engineering*. 163–174.

[59] Arvin Zakeriyan, Ramtin Khosravi, Hadi Safari, and Ehsan Khamespanah. 2021. Towards automatic test case generation for industrial software systems based on functional specifications. In *Fundamentals of Software Engineering: 9th International Conference, FSEN 2021, Virtual Event, May 19–21, 2021, Revised Selected Papers 9*. Springer, 199–214.

[60] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. 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*. 1190–1201.

[61] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.