GlueTest: Testing Code Translation via Language Interoperability

Muhammad Salman Abid[1], Mrigank Pawagi[2], Sugam Adhikari[3], Xuyan Cheng[4], Ryed Badr[5], Md Wahiduzzaman[6], Vedant Rathi[7], Ronghui Qi[8], Choiyin Li[9], Lu Liu[10], Rohit Sai Naidu[11], Licheng Lin[12], Que Liu[13], Asif Zubayer Palak[6], Mehzabin Haque[14], Xinyu Chen[5], Darko Marinov[5] and Saikat Dutta[1]

[1]Cornell University, [2]Indian Institute of Science, [3]Islington College, [4]Dickinson College, [5]University of Illinois, [6]BRAC University, [7]Adlai E Stevenson High School, [8]Wuhan University, [9]Po Leung Kuk Ngan Po Ling College, [10]University of Washington, [11]University of California, Berkeley, [12]Zhejiang University, [13]University of Shanghai for Science and Technology, [14]University of Dhaka

ma2422@cornell.edu, mrigankp@iisc.ac.in, . . ., marinov@illinois.edu, saikatd@cornell.edu

***Abstract*—Code translation from one programming language to another has been a topic of interest for academia and industry for a long time, and has recently re-emerged with the advent of Large Language Models (LLMs). While progress has been made in translating small code snippets, tackling larger projects with intricate dependencies remains a challenging task. A significant challenge in automating such translations is *validating* the resulting code. Translating existing tests to the target language can introduce errors, yielding potentially misleading quality assurance even when all the translated tests pass.**

We propose the idea of testing the translated code using the existing, *untranslated* tests written in the original programming language. The key to our idea is to leverage *language interoperability* to run code written in two different languages together. This *partial translation* approach offers two main benefits: (1) the ability to leverage original tests for validating translated code, not only from the project being translated but also from the clients using this project, and (2) the continuous maintainability and testability of the project during translation.

We evaluate our approach by translating from Java to Python two popular Java libraries, Apache Commons CLI and Apache Commons CSV, with 1209 lines of code (in 22 Java files) and 860 lines of code (in 10 Java files), respectively. Our implementation uses Oracle's GraalVM framework for language interoperability. We successfully validate the translation using the original Java tests, not just from the CLI and CSV libraries themselves but also from client projects of these libraries (30 for CLI and 6 for CSV). Our approach is the first to systematically and semi-automatically validate translations for such non-trivial libraries.

***Index Terms*—Code Translation, Software Testing, GlueTest**

I. Introduction

Code translation from one programming language to another has been studied for decades, at least since 1962 [1]. This topic has gained renewed interest with the rise of machine learning (ML) and large language models (LLMs). While ML and LLMs have made substantial advances in translating smaller code components [2]–[8], the translation of larger, more complex projects remains challenging.

One key question is how to validate the translated code. Two broad approaches have emerged. One approach is limited to *tests with inputs of primitive type*, e.g., integers or strings [9]–[12], or types that can be easily translated from one language to another, e.g., Go structs to Rust structs [13]. This approach requires only tiny test driver code that can be easily translated across languages, but it cannot handle more involved tests, e.g., unit tests that create objects of the classes under test.

The other approach is to *translate the test code* itself [5], [7], [14]. However, translating the test code can introduce inconsistencies, leading to false positives (where a translated test passes although the code under test is not translated correctly) or false negatives (where a translated test fails although the code under test is translated correctly).

We propose *GlueTest—testing the translated code using the existing, untranslated tests* written in the original programming language. The key to GlueTest is to leverage *language interoperability* to run code written in two different programming languages together. Language interoperability allows us to use the *unmodified*, existing tests written in the original language to test the translated code in the new target language. We point out that GlueTest introduces a new risk that the interoperability layer itself may introduce inconsistencies, but the interoperability code is usually much simpler than the test code, and so the risk is much lower. For example, while Apache Commons CSV has 4588 lines of test code, it only requires 502 lines of interoperability code. The interoperability code is usually boilerplate and, hence, easier to debug and potentially automate, in comparison to the translated tests.

Moreover, interoperability also allows us to test the translated code using not only tests from the project P being translated but also tests from the *clients* of P, i.e., other projects that depend on P. Such client tests can achieve a higher test coverage than just the project tests. The client code, including automated tests for the clients, can remain in the original language yet exercise the translated code. Without interoperability, benefiting from the clients and their tests to validate the translated code would be extremely challenging because one would also need to translate all the client code and its tests, increasing the cost of translation and running the risk of more inconsistencies. Beyond the ability to leverage the original tests, from the project or its clients, for validating the translated code, GlueTest also offers continuous maintainability and testability of the project during translation. We build on the idea of *partial translation* where one does not need to translate the entire project all at once but rather translates parts of a project one at a time [15]–[17].

The key challenge is to provide appropriate interoperability between the parts of the code written in two different languages. With successful interoperability, we can incrementally translate only some parts of the code while keeping the project maintainable and testable at all times. In contrast, aiming to translate the entire project can be an "all or nothing" approach where either the entire code gets translated, or it cannot be run at all. Terekhov and Verhoef report how this "all or nothing" approach led many software companies to bankruptcy [18].

To evaluate our idea in a concrete setting, we consider code translation from Java to Python, two of the most popular programming languages. Specifically, we translate Apache Commons CLI [19] and Apache Commons CSV [20], two widely-used libraries, from Java to Python. To provide interoperability between Java and Python code, we use Oracle's GraalVM [21], [22] polyglot runtime environment. We present the challenges that we encountered in this process, some solutions that we developed, and the potential for future work.

This paper makes the following contributions:

- **Idea:** We propose the first approach for testing code translation of a large project by leveraging *unmodified*, existing tests through language interoperability, which is particularly valuable in the context of translations performed by LLMs due to the unreliability of translated test code.
- **Technique:** We develop a systematic, partially automated technique for generating glue code to allow existing tests in the original language to run with the new language.
- **Evaluation:** We evaluate our approach for Java-to-Python translation on the widely-used Apache Commons CLI and CSV libraries. We successfully validate our translations of 1209 LOCs (22 Java files) in CLI and 860 LOCs (10 Java files) in CSV using our approach. These libraries are much larger codebases than those used in recent research on code translation using LLMs, which translate one method at a time rather than an entire class or multiple classes. We further successfully validate our translations using tests from the client projects, 30 for CLI and 6 for CSV. These promising results can motivate future work to use our approach for even larger codebases.

II. Background and Approach

Background. GraalVM [21], [22] is a JVM and JDK developed by Oracle that provides a high-performance runtime for applications that are written in multiple programming languages. GraalVM provides the *Polyglot* API [23] for developers to easily integrate programs written in multiple languages. Polyglot relies on the Truffle framework [21], [24] in GraalVM to allow the execution of multiple languages within the same runtime environment.

A. Our GlueTest Approach

Given the original tests and the translated code, our goal is to automatically validate the translated code by leveraging the original tests written by developers. To enable this approach, which we call *GlueTest*, we use GraalVM to run the original tests on the translated code. The main challenge of GlueTest is generating code, which we call the *glue code*, that allows the interoperability of the two languages. This approach allows one to run the original tests, *without any changes*. As a result, we avoid translating tests themselves for translation validation, which can be error-prone and, hence, unreliable.

GlueTest is particularly advantageous when developers translate a large project containing several interdependent classes and methods. GlueTest can help developers simplify the cumbersome process of translating large projects by splitting translation into multiple stages whilst ensuring that the translated code is tested at each stage. For instance, we translate classes in the reverse topological order of their dependency graph, which allows us to validate all dependencies of a class before translating the class itself, making debugging faster because bugs can be localized easier. Similarly, methods may be translated in the reverse topological order of their call graph. For maintenance of the translated project, developers would also eventually want to have the tests translated to the new programming language. However, GlueTest is still useful to enable testing of the translated code with client projects (written in the source language).

We describe the steps involved in glue code generation for the Java-to-Python code translation for an entire project with several interdependent Java classes. We identify and describe the inherent challenges and limitations of this approach and present our solutions. We also share some preliminary efforts in automatically generating such glue code. Finally, we show how GlueTest enables the validation of translated code using tests in clients, i.e., projects that depend on the library being translated, to improve the quality assurance of the translation.

B. Glue Code Generation

Common Glue Code. Listing 1 presents the common glue code shared by the entire project. This class is responsible for loading the Python classes into Java code so that the tests written in Java can access and execute these classes. Lines 1-4 import the necessary GraalVM polyglot classes. Line 10 initializes GraalVM's runtime engine (`Engine`), which is used for loading Python classes. Lines 16-18 build a new `Context` object, which represents GraalVM's global runtime that can evaluate code from other languages (Python in this case).

The `getPythonClass` method loads the Python source file containing the Python class (Lines 24-26), from the given filepath relative to the package directory containing the translated Python code (`packageDirectory`). Then it uses the context object to evaluate the source file and load the Python class into memory as an object of `Value` class. The `Value` class represents a value from the host (Java) or guest language (Python) and defines a set of language-agnostic operations on them. Finally, it returns the loaded class to the caller (Line 27).

Class-Specific Glue Code. Listing 2 presents an example class-specific glue code for the `GnuParser` class from Apache Commons CLI [19]. We add this glue code to allow interoperability of the translated Python code with the Java tests. For each original Java class, we create a file that provides the same Java API to the Python code such that the Java tests can

```
1  import org.graalvm.polyglot.Engine;
2  import org.graalvm.polyglot.Context;
3  import org.graalvm.polyglot.Source;
4  import org.graalvm.polyglot.Value;
5
6  public class ContextInitializer {
7    // Directory containing the Python package
8    private static String packageDirectory = "...";
9    // The shared GraalVM engine
10   private static Engine sharedEngine = Engine.create();
11   // Polyglot context for evaluating Python
12   private static Context context;
13   static {
14     try {
15       // Initializes Polyglot Context (global)
16       context = Context.newBuilder("python").allowAllAccess(true)
17           .option("python.PythonPath", packageDirectory)
18           .engine(sharedEngine).build();
19     } catch (Exception e) { ... }
20   }
21   // Fetches Python class from the corresponding Python file
22   public static Value getPythonClass(String path, String className) {
23     try {
24       File file = new File(path);
25       Source source = Source.newBuilder("python", file).build();
26       context.eval(source);
27       return context.getBindings("python").getMember(className);
28     } catch (Exception e) { ... }
29   }
```

Listing 1. Common glue code

```
1  package org.apache.commons.cli;
2  import org.graalvm.polyglot.Value;
3
4  public class GnuParser ... {
5    // Java representation of the Python class
6    private static Value clz = ContextInitializer.getPythonClass("
         gnu_parser.py", "GnuParser");
7    public static Value parserObj; // Java rep. of the Python object
8    public GnuParser() { parserObj = clz.execute(); }
9    ...
10   // Delegating method invocation from Java to Python
11   protected String[] flatten(final Options options, final String[]
         arguments, final boolean stopAtNonOption) {
12     return parserObj.invokeMember("flatten", options, arguments,
         stopAtNonOption).as(String[].class);
13   }
14 }
```

Listing 2. Example class-specific glue code (for `GnuParser`)

use the Python code. This technique allows us to execute the original Java tests without modifications.

The glue code for each class is generated as follows. Starting from the original Java source file, we first add constructors to initialize Python objects. In this example, we load the Python class `GnuParser` from the Python file `gnu_parser.py` using the `ContextInitializer` class on Line 6. Line 8 defines the constructor that invokes the Python constructor using polyglot's `execute` method, creating a Python object in runtime memory that the rest of this class uses.

Second, we replace each method body (e.g., `flatten`) with a call to the Python implementation of the method, while passing the same arguments, and return the values returned by the Python method with appropriate casting. For instance, Line 12 calls the `flatten` method on the Python object (`parserObj`) using the `invokeMember` method. We cast the returned Python list of strings into a Java string array using polyglot's `as` method. Finally, we remove all private members and unnecessary imports from the file.

```
1  # common utility for mapping types
2  def java_handler(cls):
3    for attr_name, attr_value in cls.__dict__.items():
4      if callable(attr_value):
5        # ignoring inner classes, which are also 'callable'
6        if not isinstance(attr_value, type):
7          setattr(cls, attr_name, type_mapper(attr_value))
8
9  # when the func is called, wrapper is called instead
10 def type_mapper(func):
11   def wrapper(self, *args, **kwargs):
12     args_ = tuple([type_map(arg) for arg in args])
13     kwargs_ = {key: type_map(val) for key, val in kwargs.items()}
14     return func.__get__(self, type(self))(*args_, **kwargs_)
15  return wrapper
16
17 # option.py
18 @java_handler
19 class Option:
```

Listing 3. Mapping types from Java to Python

```
1  public static <T, C extends Collection<T>> C valueArrayToCollection(
       Value source, Class<T> clazz, Class<C> collectionType) {
2    C result;
3    try {
4      result = collectionType.newInstance();
5      for (Value value : source.as(Value[].class)) {
6        T object = clazz.cast(clazz.getMethod("create", Value.class).
         invoke(null, value));
7        result.add(object);
8      }
9    } catch (Exception e) { result = null; }
10   return result;
11 }
```

Listing 4. Converting collection types

1) Challenges in Glue Code Generation: While GraalVM provides many utilities for integrating different languages, it does not automatically handle advanced features such as passing values of non-primitive types across languages, handling language-specific exceptions, and supporting third-party APIs. We identify and describe such challenges in the context of Java to Python translation and propose solutions to address them.

- **Type Conversion:** The Polyglot API provides inbuilt support for automatic conversion of simple types, such as integers and strings, as well as collections of such simple types between host and guest languages. However, it cannot handle passing objects of non-primitive and custom types. To address this problem, we map types during runtime from Java to Python. Listing 3 shows our utility decorator `java_handler` (Lines 2-7) that maps non-primitive function arguments passed from Java to Python, such as `File`, `Number`, and `Object`. Lines 3-7 iterate over all methods in the class (identified by the `callable` attribute on Line 4) and replace them with a `wrapper` method. The wrapper method (Lines 11-14) replaces each function argument with our custom `type_map` from Java to Python type. We annotate each translated Python class with this method, such as the `Option` class shown below (Line 18). This annotation method drastically reduces the amount of required glue code and developer effort.

- **Handling collections of non-primitive types:** Polyglot cannot handle passing collections of non-primitive types between languages. Listing 4 presents our generic method to convert collections of custom types from Python to Java. This method takes a `Value` object (`source`) representing a Python list, the target Java class (`clazz`), and the Java collection type (`collectionType`), and creates a collection of Java objects of the specified type `T` to wrap these Python objects.
  In Listing 4, Line 4 first initializes a collection of the Java type `C`: `result`. Lines 5-8 then cast each Python object into a Java type by invoking the `create` method (Line 6). The `create` method is a constructor defined for each Java class that wraps a Python object of the same class to a Java object.
- **Value Comparison:** Values and objects passed through interop from Java that have no mappings defined for Python types get the type "foreign". For instance, passing a Java `null` to Python casts it to a "None-like" object of the "foreign" type, so Python statements such as `arg is None` do not work as expected, though statements such as `arg == None` do. We explicitly map Java `null` objects to Python `None` objects during runtime to avoid making any changes to either Java tests or the translated Python source code. To allow comparison of Python objects wrapped in Java objects, we add an attribute setting the class name for each Python class during runtime. We then redefine Python's `isinstance` method to explicitly check this attribute instead of the default behavior (which returns "foreign" due to GraalVM).
- **Handling exceptions:** When an exception occurs in the guest language (Python), GraalVM raises a generic exception of the `PolyglotException` class in the host language (Java). Because developers often write special code to handle specific kinds of exception classes in Java, we introduce custom glue code to handle different kinds of exceptions.
- **Handling third-party classes:** We write custom code to handle the conversion of objects of third-party classes when passed from Java to Python. For instance, the `Properties` class in Java is used to store and pass configurations. However, since Python does not have a similar class, we create a Python dictionary from the object. This limitation might pose a bigger challenge for more complex projects. In such cases, we need to either map members of the third-party class to a custom class or a class of a similar kind in the guest language.
- **Incompatibility of Standard APIs:** The behavior of some standard APIs that ship with the language compiler or interpreter may differ across languages. For instance, Java's default hash function returns integers, whereas Python's default hash function returns long integers. Thus we need to extract the Python result as a long integer and cast it as an integer.

2) Automating Glue Code Generation: We implement several glue code generation steps into a prototype tool. We use JavaParser [25] for parsing Java files and implement each generation step as a transformation rule. Of our 8 transformation rules, we fully implemented 5: (1) common glue code, (2) imports from polyglot, (3) adding helper fields/methods, (4) removing private fields/methods, and (5) replacing method bodies with calls to Python. We did not fully implement (6) general exception handling, (7) type conversion for collection types, and (8) type conversion for some library types.

Our prototype can fully glue (i.e., generate fully correct glue code) 4 classes in each of the two libraries and also partially glue 13 classes in CLI and 3 classes in CSV. Counting methods, it can fully glue 114 (56.7% of 201 methods) in CLI and 53 (27.9% of 129) in CSV. It can also partially glue 53 (41.1%) in CLI and 68 (52.7%) in CSV, where the automation creates a template to be filled out manually by a developer. We expect future work to improve automation of our approach.

C. Validation Using Client Tests

The clients of the library being translated, i.e., projects that depend on that library, may contain tests that allow us to test the translation. Testing translated code using the clients' tests can help improve the coverage of the translated code over the original tests by exercising more diverse code behavior with varied inputs present in the clients' tests. We automatically scrape GitHub's dependency tracking pages for CLI [26] and CSV [27] to collect such client projects. To limit the number of projects for evaluation, we only select projects that have at least 100 forks and stars. We further filter out projects that do not support specific library versions that we translate (i.e., CLI 1.5.0 and CSV 1.6.0). For the remaining projects, we check if they use the APIs of the translated library. Before running the tests of each client, we edit its `pom.xml` file to use our translated version of the libraries instead of the original.

III. Preliminary Results

We instantiate our approach on Java and Python, two of the most popular languages. We evaluate GlueTest using the Apache Commons CLI [19] and Apache Commons CSV [20] projects. Both are popular Java libraries that provide APIs for parsing command line options and reading/writing CSV files, respectively. CLI contains 22 Java files, 21 top-level classes, 19 test classes, and 438 tests, while CSV contains 10 Java files, 8 top-level classes, 18 test classes, and 306 tests.

Code Translation. We manually translate CLI and CSV from Java to Python. We follow a systematic process with lots of reviewing to minimize the chances of introducing bugs in the translated code. For each translated class, a member of our research team raises a pull request that is reviewed by at least two other members. The original Java code has 1209 lines of code for CLI and 860 lines for CSV, whereas the translated Python code has 1626 lines of code for CLI and 1851 lines for CSV. A key reason for increase is handling overloaded methods; Python does not support overloading, so we needed to simulate it by checking argument types and dispatching to appropriate methods. The translation was carried out by a group of undergraduate students with varying levels of programming experience, who anecdotally estimate spending $\sim$4.5 hours on average translating each main (non-test) class.

Results. We manually write the glue code classes by implementing the steps outlined in Section II-B and II-B1. These classes replace all original Java classes. We followed the same systematic process as above for raising PRs and reviewing code. We also partially automated the process of generating the glue code (Section II-B2). The glue code was written by a subset of the students who report spending $\sim$1 hour on average for each main class. We were able to run and pass all original 438 Java tests in CLI and 306 Java tests in CSV with the translated Python code, indicating that our translation was correct for these tests.

Testing using client projects. Using the process described in Section II-C, we find 30 client projects for CLI and 6 for CSV. Our evaluation involves a small subset of client projects, as we only select clients that (1) are compatible with the specific versions of the libraries we translate, and (2) include tests that invoke the libraries under translation. We build all client projects and run their tests with our translated Python code for CLI or CSV via the Graal glue code integration. All their tests pass when run with our translated Python code and glue code, further increasing our confidence in our translated code.

Data Availability. We share all our translations and glue code with documentation in our artifact [28].

IV. Related Work

Code Translation. Many techniques have been proposed for automated code translation since at least 1962 [1]. Earlier literature introduced various rule-based methods [1], [29], [30], leading to development of numerous tools [31]–[38]. More recent research has employed novel methods such as statistical machine translation [2], [39]–[43]. The latest research in the field utilizes approaches such as tree-to-tree neural networks [3] and other deep-learning techniques [44], including large language models (LLMs) [6], [13], [45]–[52].

For some specific projects, Sneed [53] explored the translation of COBOL code to Java in an airport management system; Lee et al. [11] employ a neurosymbolic approach for transpilation of assembly code, involving low-level programming languages; Roziere et al. [4] developed a fully unsupervised technique, called TransCoder, using a transformer architecture for source-to-source translation tasks. As common in prior work, Roziere et al. evaluated TransCoder for translating one function at a time, and these translated functions had arguments of primitive types. In contrast, the focus of our work is to develop a novel, systematic way to validate translated code by *reusing the original, unmodified unit tests* which invoke methods multiple times with arguments of non-primitive types. Further, in our work, we focus on the translation of an entire project with several interdependent classes.

Recently, Pan et al. [7] evaluated the code translation abilities of LLMs. They validate the translated code by checking whether it "compiles, passes runtime checks, and existing tests pass on the translated code." Like us, they also report that language-specific features, especially method overloading, can cause translation challenges for real-world projects.

Validating translated code. Prior work has used many techniques to validate translated code. For instance, Roziere et al. [5] used EvoSuite to automatically generate tests to validate translated methods. However, automated unit-test generation techniques often generate low-quality assertions and struggle with non-primitive input types. Another study [4] used manually created unit tests for validation, which requires high domain expertise and time to prepare quality tests. Other works rely on existing tests in target languages to validate translations [2], [3], [7], but such tests are not always available. Guizzo et al. [54] leverage mutation analysis for code translation assessment. Eniser et al. [55] focus on testing the translation model itself rather than the translated programs.

In the compiler testing domain, researchers have proposed many verification-based techniques [56]–[59] to validate compiler optimizations. However, such techniques require specifying the entire semantics of a language and have not been widely adopted. In contrast, our approach can validate translations across different source languages. Because we leverage existing tests, our approach can easily scale to large projects.

Transforming Legacy Systems. Prior studies also considered code translation, testing, or re-engineering concerning legacy systems [15], [60]–[63]. For instance, Sneed [15] identified key objectives for re-engineering, like improving maintainability and achieving greater reliability, and discussed encapsulation of legacy software [16], [64] in the reuse of legacy systems. Sneed explored the use of wrappers for the original code in the new language, enabling partial language translation, similar to our glue code. Baxter [60] used rule-based transformations for legacy systems. Bruneliere et al. [63] explored a comprehensive framework for model-reverse engineering by transforming the existing code of legacy systems into higher-level Eclipse Modeling Framework models. Unlike these approaches, our work focuses on maintaining the testability of translated code.

V. Conclusion and Future Work

In this work, we proposed the idea of validating translated code using original tests by employing language interoperability. We demonstrated the feasibility of our approach, GlueTest, on the Apache Commons CLI and CSV libraries, for Java-to-Python translations. We successfully validated the translation of all classes in both libraries using the original, *unmodified* Java tests via GraalVM glue code. Because the effectiveness of our approach is sensitive to the quality of the test suite in the original project (and its clients), future work can explore techniques like mutation testing to expand the fault-detection ability of the tests and perform a more thorough validation of the translated code. We also presented our initial attempts at automating the glue code generation. Future work can investigate automating the entire glue code generation process, generalizing our insights to translation between other pairs of languages, and evaluating our approach on larger codebases.

Acknowledgments

We thank the reviewers for their valuable comments and Faustino Aguilar for help in some translations. This work was supported in part by NSF grants CCF-1763788 and CCF-1956374, and by IBM-Illinois Discovery Accelerator Institute.

References

[1] R. S. Ledley and J. B. Wilson, "Automatic-programming-language translation through syntactical analysis," *Commun. ACM*, 1962.

[2] K. Aggarwal, M. Salameh, and A. Hindle, "Using machine translation for converting Python 2 to Python 3 code," *PeerJ Prepr.*, 2015.

[3] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *NeurIPS*, 2018.

[4] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *NeurIPS*, 2020.

[5] B. Roziere, J. M. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "Leveraging automated unit tests for unsupervised code translation," *arXiv*, 2021.

[6] Q. Sun, N. Chen, J. Wang, X. Li, and M. Gao, "TransCoder: Towards unified transferable code representation learning inspired by human skills," *arXiv*, 2023.

[7] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *ICSE*, 2024.

[8] Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang, "Unifying the perspectives of NLP and software engineering: A survey on language models for code," *arXiv*, 2024.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv*, 2021.

[10] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks," *arXiv*, 2021.

[11] C. Lee, A. Mahmoud, M. Kurek, S. Campanoni, D. Brooks, S. Chong, G.-Y. Wei, and A. M. Rush, "Guess & sketch: Language model guided transpilation," *arXiv*, 2023.

[12] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation," in *NeurIPS*, 2023.

[13] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening, "Towards translating real-world code with LLMs: A study of translating to Rust," *arXiv*, 2024.

[14] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda, "Multipl-e: A scalable and polyglot approach to benchmarking neural code generation," *TSE*, 2023.

[15] H. M. Sneed, "Planning the reengineering of legacy systems," *IEEE Software*, 1995.

[16] ——, "Encapsulation of legacy software: A technique for reusing legacy software components," *Annals of Software Engineering*, 2000.

[17] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBLEU: a method for automatic evaluation of code synthesis," *arXiv*, 2020.

[18] A. Terekhov and C. Verhoef, "The realities of language conversions," *IEEE Software*, 2000.

[19] 2024, https://commons.apache.org/proper/commons-cli.

[20] 2024, https://commons.apache.org/proper/commons-csv.

[21] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One VM to rule them all," in *Onward!*, 2013.

[22] 2024, https://www.graalvm.org/.

[23] 2024, https://graalvm.org/latest/reference-manual/polyglot-programming/.

[24] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck, "High-performance cross-language interoperability in a multi-language runtime," in *Symposium on Dynamic Languages*, 2015.

[25] 2024, https://javaparser.org.

[26] 2024, https://github.com/apache/commons-cli/network/dependents.

[27] 2024, https://github.com/apache/commons-csv/network/dependents.

[28] 2024, https://figshare.com/s/8a729bac89aefb980f9f.

[29] P. van den Bosch, "The translation of programming languages through the use of a graph transformation language," *ACM SIGPLAN Notices*, 1982.

[30] L. Qiu, "Programming language translation," Cornell, Tech. Rep., 1999.

[31] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to high-level language translation," in *ICSM*, 1998.

[32] L. De Rose and D. Padua, "Techniques for the translation of MATLAB programs into Fortran 90," *TOPLAS*, 1999.

[33] 2024, https://github.com/mono/sharpen.

[34] 2024, https://github.com/paulirwin/JavaToCSharp.

[35] 2024, https://github.com/immunant/c2rust.

[36] 2024, https://github.com/gotranspile/cxgo.

[37] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating C to safer Rust," *OOPSLA*, 2021.

[38] H. Zhang, C. David, Y. Yu, and M. Wang, "Ownership guided C to Rust translation," in *CAV*, 2023.

[39] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *ESEC/FSE*, 2013.

[40] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Onward!*, 2014.

[41] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Migrating code with statistical machine translation," in *ICSE Companion*, 2014.

[42] ——, "Divide-and-conquer approach for multi-phase statistical migration for source code," in *ASE*, 2015.

[43] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *ASE*, 2015.

[44] F. Liu, J. Li, and L. Zhang, "Syntax and domain aware model for unsupervised program translation," in *ICSE*, 2023.

[45] L. Gong, J. Wang, and A. Cheung, "ADELT: Transpilation between deep learning frameworks," *arXiv*, 2023.

[46] J. Pan, A. Sadé, J. Kim, E. Soriano, G. Sole, and S. Flamant, "SteloCoder: a decoder-only LLM for multi-language to Python code translation," *arXiv*, 2023.

[47] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, "Multilingual code co-evolution using large language models," in *FSE*, 2023.

[48] Z. Tang, M. Agarwal, A. Shypula, B. Wang, D. Wijaya, J. Chen, and Y. Kim, "Explain-then-translate: An analysis on improving program translation with self-generated explanations," in *EMNLP*, 2023.

[49] S. Gandhi, M. Patwardhan, J. Khatri, L. Vig, and R. K. Medicherla, "Translation of low-resource COBOL to logically correct and readable Java leveraging high-resource Java refinement," in *LLM4Code*, 2024.

[50] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and unleashing the power of large language models in automated code translation," *arXiv*, 2024.

[51] V. Nitin and B. Ray, "Spectra: Enhancing the code translation ability of language models by generating multi-modal specifications," *arXiv*, 2024.

[52] A. Z. H. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening, "VERT: Verified equivalent rust transpilation with large language models as few-shot learners," *arXiv*, 2024.

[53] H. M. Sneed, "Migrating from COBOL to Java," in *ICSM*, 2010.

[54] G. Guizzo, J. M. Zhang, F. Sarro, C. Treude, and M. Harman, "Mutation analysis for evaluating code translation," *Empirical Softw. Engg.*, 2023.

[55] H. F. Eniser, V. Wüstholz, and M. Christakis, "Automatically testing functional properties of code translation models," in *AAAI*, 2024.

[56] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *TACAS*, 1998.

[57] T. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *PLDI*, 2013.

[58] S. Gupta, A. Rose, and S. Bansal, "Counterexample-guided correlation algorithm for translation validation," in *OOPSLA*, 2020.

[59] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: Bounded translation validation for LLVM," in *PLDI*, 2021.

[60] I. D. Baxter, "Using transformation systems for software maintenance and reengineering," in *ICSE*, 2001.

[61] ——, "DMS: Program transformations for practical scalable software evolution," in *IWPSE*, 2002.

[62] R. Akers, I. Baxter, M. Mehlich, B. Ellis, and K. Luecke, "Re-engineering C++ component models via automatic program transformation," in *WCRE*, 2005.

[63] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot, "MoDisco: A model driven reverse engineering framework," *Inf. Softw. Technol.*, 2014.

[64] H. M. Sneed, "Encapsulating legacy software for use in client/server systems," in *WCRE*, 1996.