AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation

ALI REZA IBRAHIMZADA, University of Illinois Urbana-Champaign, USA
KAIYAO KE, University of Illinois Urbana-Champaign, USA
MRIGANK PAWAGI, Indian Institute of Science, India
MUHAMMAD SALMAN ABID, Cornell University, USA
RANGEET PAN, IBM Research, USA
SAURABH SINHA, IBM Research, USA
REYHANEH JABBARVAND, University of Illinois Urbana-Champaign, USA

Code translation transforms programs from one programming language (PL) to another. One prominent use case is application modernization to enhance maintainability and reliability. Several rule-based transpilers have been designed to automate code translation between different pairs of PLs. However, the rules can become obsolete as the PLs evolve and cannot generalize to other PLs. Recent studies have explored the automation of code translation using Large Language Models (LLMs). One key observation is that such techniques may work well for crafted benchmarks but fail to generalize to the scale and complexity of real-world projects with inter- and intra-class dependencies, custom types, PL-specific features, etc. We propose AlphaTrans, a neuro-symbolic approach to automate *repository-level* code translation. AlphaTrans translates both source and test code, and employs multiple levels of validation to ensure the translation *preserves* the functionality of the source program. To break down the problem for LLMs, AlphaTrans leverages program analysis to decompose the program into fragments and translates them in the *reverse call order*.

We leveraged AlphaTrans to translate *ten* real-world open-source projects consisting of ⟨836, 8575, 2719⟩ (application and test) classes, (application and test) methods, and unit tests. AlphaTrans breaks down these projects into 17874 fragments and translates the entire repository. 96.40% of the translated fragments are syntactically correct, and AlphaTrans validates the translations' runtime behavior and functional correctness for 27.03% and 25.14% of the application method fragments. On average, the integrated translation and validation take 34 hours (min=3, max=121) to translate a project, showing its scalability in practice. For the syntactically or semantically incorrect translations, AlphaTrans generates a report including existing translation, stack trace, test errors, or assertion failures. We provided these artifacts to two developers to fix the translation bugs in four projects. They fixed the issues in 20.1 hours on average (5.5 hours for the smallest and 34 hours for the largest project) and achieved all passing tests. Without AlphaTrans, translating and validating such big projects could take weeks, if not months.

Additional Key Words and Phrases: Neuro-symbolic Code Translation and Validation

1 Introduction

Application modernization offers numerous benefits to developers, including better performance, maintainability, productivity, reliability, and security [27, 28, 30, 31]. Manual migration or modernization of real-world projects can be time-consuming and error-prone. Code translation can help automatically convert programs from one programming language (PL) to another.

Transpilers solely rely on program analysis and perform rule-based translation, failing to translate code between languages that greatly differ in syntax or semantics [4]. This also makes them very PL-specific; they cannot generalize to newer features of the same PL pairs easily, let alone other PLs.

Authors' Contact Information: Ali Reza Ibrahimzada, alirezai@illinois.edu, University of Illinois Urbana-Champaign, Urbana, Illinois, USA; Kaiyao Ke, kaiyaok2@illinois.edu, University of Illinois Urbana-Champaign, Urbana, Illinois, USA; Mrigank Pawagi, mrigankp@iisc.ac.in, Indian Institute of Science, Bengaluru, Karnataka, India; Muhammad Salman Abid, ma2422@cornell.edu, Cornell University, Ithaca, NY, USA; Rangeet Pan, rangeet.pan@ibm.com, IBM Research, Yorktown Heights, NY, USA; Saurabh Sinha, sinhas@us.ibm.com, IBM Research, Yorktown Heights, NY, USA; Reyhaneh Jabbarvand, reyhaneh@illinois.edu, University of Illinois Urbana-Champaign, Urbana, Illinois, USA.

Finally, the translations lack readability, requiring much effort to understand and validate them, and naturalness, failing to create idiomatic code in the target PL [42]. State-of-the-art code translation techniques attempt to harvest the emerging abilities of Large Language Models (LLMs) in code synthesis to overcome the limitations of transpilers [40, 42, 60]. However, these techniques are still limited to translating simple programs in crafted benchmarks or selected slices of real-world projects due to the following challenges:

(1) *Problem complexity.* The source and target PLs can be fundamentally different in programming paradigms, typing, and memory management. Some PLs have specific properties that may not exist in others, e.g., constructor overloading in Java. Such complexities are beyond the abilities of existing LLMs to handle, causing them to hallucinate when translating types, code constructs, or even method names [42], making translations non-compilable or useless.
(2) *Validation.* The translation should preserve the functionality of the source project. Most existing techniques follow a "translation first and validation next" approach, which can postpone the validation and not benefit from the potential use of validation as feedback to correct the translation [42]. A few techniques use formal methods [40, 60] to verify translations on the go. However, these techniques cannot scale to real-world projects. One possible solution for validation is reusing the tests in the source language. However, due to (1) multiple invocations of different methods in unit tests and (2) inherent long call chains in real-world projects, testing a translated method *in isolation* is impossible.
(3) *Limited context window.* Concerning repository-level translation, the entire project and, in many cases, even the entire class cannot fit into the context window of current LLMs [24]. Assuming an unlimited context window, LLMs still suffer from short attention span [36], preventing them from properly capturing the intra- and inter-procedural dependencies in real-world projects.

This paper presents AlphaTrans, a neuro-symbolic[1] approach for automated repository-level code translation and validation. AlphaTrans leverages static analysis to resolve PL-specific features of the source language (§4.1), decompose the source project into smaller fragments (§4.2), and create a compilable project skeleton in the target language (§5). It then starts translating fragments in the reverse call order and validates them using existing tests when possible (§6). After translating each fragment, AlphaTrans updates the project skeleton and ensures the whole project compiles, gradually translating and validating the source project into the target PL. To improve translation quality, static analysis again comes to the rescue: AlphaTrans collects relevant context for each fragment, including translated callee methods and surrounding contexts, e.g., class declaration, global variables/fields, etc. It also uses relevant in-context examples based on the specific properties of the fragment to be translated. The current version of AlphaTrans implements two levels of dynamic validation: (1) running the source tests on the translated fragment in isolation using language interoperability (§6.1) and (2) decomposing, translating, and executing the unit tests on the translated project (§6.3). Finally, AlphaTrans recomposes the translated fragments to create the program in the target PL (§6.2).

The idea of compositional translation and validation proposed by us is PL-agnostic; however, implementing the program transformation component is PL-specific. For the first version of AlphaTrans, the implementation supports translating from Java to Python. Our motivations for choosing this PL pair are: (1) Java offers many features that are not supported or common in other PLs by default (e.g., method/constructor overloading, complex types, circular dependencies, local or anonymous inner classes, interfaces, etc.); (2) Python programs are not compiled but interpreted,

[1]The keyword symbolic here refers to a general term of symbolic learning in contrast to machine learning and should not be confused with symbolic execution. We refer to combining LLMs and program analysis as a neuro-symbolic approach.

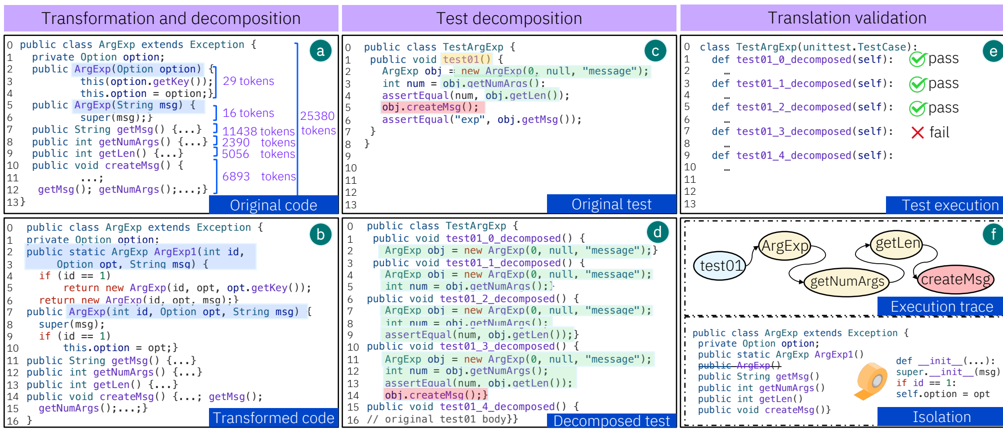

Fig. 1. Illustration of key challenges in repository-level code translation and AlphaTrans addressing them.

which makes many translation issues that can be caught at the compile time stay there until test execution and challenge the validation; and (3) both PLs are popular (top-5 on the TIOBE index [54]).

Using AlphaTrans to translate ten real-world Java projects to Python corroborates its *effectiveness*: It can translate 17874 field/method/test fragments, with 96.40% syntactical correctness. For the 4654 application method fragments that can be further evaluated through test execution, AlphaTrans achieves 27.03% runtime validation and 25.14% functional equivalence. AlphaTrans is *scalable*, completing translations in 34 hours, on average. Human subjects improved partial translation of AlphaTrans and achieved green test suites within 20.1 hours, on average, showing *practicality* of AlphaTrans. These results were achieved using a moderate-size open-access LLM (DeepSeek-Coder-33b-Instruct [23]). A stronger model, i.e., GPT-4o, improves the performance of AlphaTrans to 99.2% syntactic correctness for all fragments and 27.95% functional equivalence for application method fragments, with an overhead of $14.39 per project, on average. The affordable cost is due to the novel features of the pipeline, namely, decomposition into fragments, prompt crafting, in-isolation validation of translations, and efficient feedback loop.

To the best of our knowledge, AlphaTrans is the *first technique to translate an entire repository*, including tests, and generates validated translations (considering existing tests). The only prior repository-level translation attempt using GPT-4 [42] (translating Apache Commons CLI from Java to Python) resulted in non-compilable code, let alone the translation being validated. AlphaTrans is also *the first technique leveraging language interoperability for in-isolation validation of translated fragments.* The effort of human subjects to fix translation bugs by AlphaTrans and to achieve green tests creates pragmatic bug data sets for testing, bug localization, and program repair research. Our code and artifacts are publicly available for reproducing the results or translating new projects [32].

2 Challenges in Repository-Level Code Translation

To illustrate the most notable challenges in repository-level code translation and validation, we use the hypothetical example in Figure 1, inspired by the complexities in real-world Java projects.

Challenge 1: Class Size. The class consists of 25,380 tokens (a). Instructions for translating the code, in-context examples, and the model's response can also significantly increase the number of input tokens. While some commercial LLMs support tens of thousands of tokens, many open-access LLMs do not. For example, DeepSeek-Coder-33b-Instruct [23] used in this paper has a context window of 16,384 tokens, of which only 4,096 tokens can be used for generation. To address this challenge, AlphaTrans decomposes Java application classes into smaller *field* or *method fragments* and translates each separately in a reverse call order (§4.2.1, §4.2.2).

Challenge 2: PL-specific Properties. Java programs frequently use method/constructor overloading, which is not supported by default in Python (a). This example shows instances of constructor overloading (lines 2 and 5). In Python, declaring two constructors is allowed; however, at runtime, the last declaration overrides all previous constructors, resulting in unexpected behavior. To address this issue, AlphaTrans employs program analysis to refactor the original code while preserving the functionality (through test execution). The transformation includes changing the constructor's name, updating the references, and changing the constructor's implementation. The transformed code (b) makes the source program amenable to translation to Python.

Challenge 3: Validation. To illustrate the challenges with validation, consider `test01` (c) that invokes *four* methods in its body (`ArgExp`, `getNumArgs`, `getLen`, and `createMsg`) to test the functionality of method `getMsg` in the assert statement. Suppose we can successfully translate all methods except `createMsg`. If we choose test translation, the most natural way of validating code translation, the execution of the translated test results in a runtime error when invoking `createMsg`. As a result, a translation issue in one method casts a shadow in validating the translation of the other methods. We refer to this issue as the *test translation coupling effect*. To overcome this challenge, AlphaTrans executes source language tests as-is (i.e., without translation) by leveraging a language-interoperability framework called GraalVM [41] (f). In this setting, a test in the source language is executed every time one of its invoked application methods (method fragments) is translated. This approach validates *functional equivalence* of each method *in isolation* since other methods invoked in the test or the body of the translated method can remain in the source language.

Challenge 4: Test Translation. GraalVM has certain limitations (§6.1), which prevents AlphaTrans from validating all the code fragments in isolation. Furthermore, we need to translate tests regardless of whether they are used for validation to maintain the translated projects in the target language. Test errors as a result of *test translation coupling effect* under-approximates the quality of translation: failing to validate the translation of four methods because of one incorrect translation. Root causing the translation bugs also requires additional efforts from developers, i.e., looking at the stack trace and coverage. To overcome this challenge, AlphaTrans decomposes the original test suite into *test fragments* (d). Executing the translated decomposed test suite results in three test passes (e), validating the *runtime behavior* of three methods that the original test suite could not promptly provide.

An alternative approach is parsing the stack trace and code coverage results for each runtime error during translation. However, test decomposition is a cleaner way to see the results per test execution promptly. It is also done once before translation. In translation to interpreted languages such as Python, specifically, the execution of test fragments can validate the runtime behavior of methods before waiting for functional validation. For fragments that GraalVM cannot validate, if AlphaTrans can successfully translate all the methods invoked during test execution and test passes, such test will also be used for validating *functional correctness*.

3 Overview of Approach

AlphaTrans consists of three main phases, shown in Figure 2: program transformation and decomposition (§4), type translation and skeleton construction (§5), and compositional translation and validation (§6). The first two phases aim to decompose and simplify the repository-level code translation problem for LLMs, helping the third phase yield high-quality validated translations.

The program transformation and decomposition phase first refactors the PL-specific properties of the source program into programming paradigms common among many PLs (§4.1). Next, it decomposes the source project into smaller units, i.e., *fragments*, and stores fragment dependencies in a data structure called *schema* (§4.2).

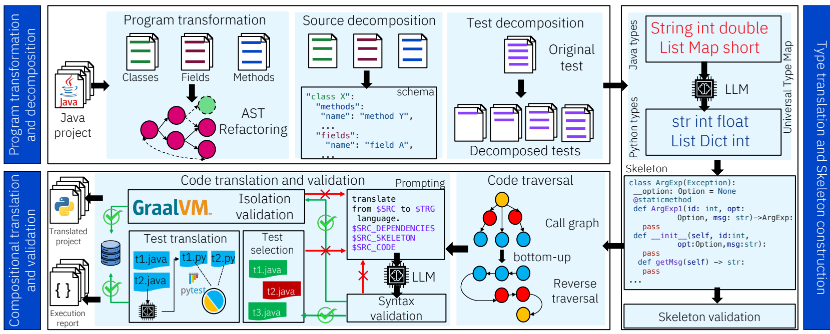

Fig. 2. Overview of AlphaTrans.

The type translation and skeleton construction phase takes the schema as input and produces *target project skeleton*, i.e., a compilable project in the target language with method signatures but no method implementation (§5.2). The first translation step happens here, where it translates the source PL types to the target PL to ensure that class skeletons are compilable (§5.1). The outcome of type translation is a type mapping from the source to the target PL, which AlphaTrans can reuse in translating other projects.

The compositional translation and validation takes schema and project skeleton as inputs and translates fragments, in the reverse call order, by prompting an LLM. After translating a fragment, it updates the class skeleton with a new translation and checks whether the skeleton is compilable. For method fragments, AlphaTrans looks for corresponding tests and, if any exist, validates them. The first level of validation is performed through GraalVM's language interoperability to isolate the validation of the method using tests in the source language. Next, AlphaTrans translates and executes the corresponding tests. In case of compilation errors or test failures, AlphaTrans re-prompts the LLM with feedback (from the compilation and runtime errors) to improve the translation. If no improvement is achieved within a certain budget, AlphaTrans continues to the next fragment until all are translated. For methods whose translations are not compilable or result in test errors/failures, AlphaTrans generates reports consisting of existing translations and relevant artifacts, such as stack traces, test errors/failures, and test coverage information.

4 Program Transformation and Decomposition

4.1 Program Transformation

This component performs semantics-preserving refactoring of method and constructor overloading in Java code to make it amenable to translation to Python. Other Java-specific features, namely, circular dependencies, inner classes, interfaces, and abstract classes, will be handled when constructing the project skeleton in Python (§5.2). The reason for resolving method and constructor overloading in the source language is that we have to change the implementation, i.e., call sites to methods and constructors. Therefore, such changes should be validated using source tests.

For overloaded methods, AlphaTrans makes each method name unique by adding an integer suffix (starting at 0) to the name and updates all call sites based on the new method names. Resolving overloaded constructors is not as straightforward, as they should have the same name as the enclosed declaring class. Furthermore, the invocation of constructors inside each other and the Java inheritance mechanism makes constructor overloading complex. Our algorithm (Algorithm 1) for resolving the constructor overloading handles three prominent[2] use cases shown in Figure 3.

[2]Following the best practices for constructor overloading from *Stack Overflow* and analyzing the use of constructor overloading in open-source projects, we categorized the use cases into the mentioned rules.

Original code

```java
public ArgExp(Option opt) {
  this.option = opt;
  this.threshold = 1;
}
public ArgExp(String msg) {
  this.msg = msg;
}
```

Transformed code

```java
public ArgExp(int id, Option opt, String msg){
  if (id == 0)
{
   this.option = opt;
   this.threshold = 1;
}
  else {
   this.msg = msg
}
```

(a)

Original code

```java
public ArgExp(Option opt) {
  this(opt.getKey());
}

public ArgExp(String msg) {
  super(msg);
}
```

Transformed code

```java
public static ArgExp ArgExp1(Option opt) {
  return new ArgExp(opt.getKey());
}


public ArgExp(String msg) {
  super(msg);
}
```

(b)

Original code

```java
public ArgExp(Option opt) {
  this(opt.getKey());
  this.option = opt;
}
public ArgExp(String msg) {
  super(msg);
}
```

Transformed code

```java
public static ArgExp ArgExp1(
                  int id, Option opt, String msg){
  if (id == 1)
    return new ArgExp(id, opt, opt.getKey());
  return new ArgExp(id, opt, msg);}
public ArgExp(int id, Option opt, String msg){
  super(msg);
  if (id == 1)
    this.option = opt;}
```

(c)

Fig. 3. Constructor overloading patterns and their corresponding transformations.

The first pattern (Figure 3-a) shows multiple independent constructors. AlphaTrans merges these constructors into one and uses an `id` parameter to differentiate between them. All the call sites of the constructors will be updated accordingly, based on the given `id`. The second use case (Figure 3-b) is more challenging to resolve, as one constructor calls the other using `this()`. AlphaTrans transforms the first constructor into a factory method and invokes the second constructor inside it. Factory methods are static, and AlphaTrans updates the call sites by directly invoking them on the class instance, e.g., `ArgExp.ArgExp1(id,opt,msg)`. The last use case (Figure 3-c) is similar to the second, except that both constructors implement some code. AlphaTrans refactors the first constructor into a factory method and adds an extra `id` parameter to differentiate between behaviors implemented by different constructors. The call sites of the constructors will be updated like in previous cases. Real-world projects often combine these patterns, which AlphaTrans handles using Algorithm 1.

Algorithm 1: Constructor Overloading

Inputs: Overloaded Constructors *OCs*
Output: Code without Overloaded Constructors *NOCs*

if $!hasThisCall(OCs)$ **then**
    $ids \leftarrow createConstructorIDS(OCs)$;
    $NOCs \leftarrow merge(OCs, ids)$;
else
    if $hasOnlyThisCall(OCs)$ **then**
        $refactor \leftarrow createRefactorMethod(OCs)$;
        $NOCs \leftarrow merge(OCs, refactor)$;
    else
        $refactor \leftarrow createRefactorMethod(OCs)$;
        $ids \leftarrow createConstructorIDS(OCs)$;
        $NOCs \leftarrow merge(OCs, refactor, ids)$;
$refactorCallSites()$;
return *NOCs*;

4.2 Program Decomposition

Translating the entire repository of real-world projects is a very complex problem. As a result, AlphaTrans breaks down projects into fragments, performs the translation and validation at the fragment level, and re-composes the translation as a repository in the target language.

4.2.1 Source Decomposition. Real-world projects can include hundreds of files with thousands of lines of code, which exceed the context window of state-of-the-art LLMs. AlphaTrans employs static analysis to decompose code into smaller fragments, i.e., *field fragments* and *method fragments*. A field fragment includes modifiers, type, name, and field value. A field fragment can belong to an application or test class. A method fragment includes the full signature of an individual method in the source language. A method fragment can be an application or test method (e.g., helper methods or unit tests). During decomposition, AlphaTrans extracts meta-information related to the fragments, such as their location in code (e.g., start and end line), code (e.g., implementation between start and end line), dependencies (e.g., callers and callees), types (of inputs and output), and other necessary information such as file paths, class inheritance, imports, and method annotations. AlphaTrans stores fragments and their corresponding collected meta-data in a data structure called *schema*, which components in other phases will use. AlphaTrans also extracts the call graph to guide the translation, i.e., translate fragments in reverse call order.

4.2.2 Test Decomposition. The burden of validating *functional equivalence* in AlphaTrans is on GraalVM. Yet, we still need to translate and execute tests to validate the fragments that GraalVM cannot be used to validate (§6.1). Unit tests in real-world projects can invoke multiple methods and include multiple assert statements. Furthermore, long call chains are inevitable in real-world projects due to the high degree of intra- and iter-procedural dependencies. As we show later (§7.4), the average number of direct method invocations and method executions in tests for our studies subjects are 3 and 27, respectively. This can result in *test translation coupling effect* discussed in §2.

To enable test translation for validating runtime validation or functional equivalence, AlphaTrans decomposes each unit test into a series of *test fragments*, as shown in Figure 1-d. It uses each statement with a call to an application method as a cutting point. For statements enclosed by branches, loops, of exception-handling blocks, AlphaTrans includes the entire block. This process generates an ordering of executable test fragments for each unit test. Each test fragment includes all the statements of the lower-order fragments, along with additional statements that invoke *only one* method that was not invoked by previous fragments. AlphaTrans executes test fragments in increasing order until a test fails and skips running following fragments, as they will also fail.

5 Type Translation and Skeleton Construction

5.1 Type Translation

Automatically resolving types is a challenging problem [22, 53], and a large body of work has attempted to address this, mostly using symbolic rule-based approaches [6, 9, 10, 33, 46, 47]. AlphaTrans employs a Retrieval-Augment Generation (RAG) [35] technique for finding the equivalent types in target language. To that end, it first extracts all the types in the source language of a given project. Custom, application-level types will be resolved during the translation as AlphaTrans translates fragments and classes in the target language. For the remaining types, it crawls the API documentation of the source language and retrieves the relevant description of each type.

```python
class ArgExp(Exception):
  __option: Option = None
  @staticmethod
  def ArgExp1(id: int, opt: Option, \
              msg: str) -> ArgExp:
    pass
  def __init__(self, id: int, opt: \
               Option, msg: str):
    pass
  def getMsg(self) -> str:
    pass
  def getNumArgs(self) -> int:
    pass
  def getLen(self) -> int:
    pass
  def createMsg(self) -> None:
    pass
```

Fig. 4. Target PL skeleton for the example in Figure 1-b.

To form the prompt[3], AlphaTrans uses the retrieved description and instructs the model with an in-context example to return the most relevant type in the target language, given the use of types in the source language and the retrieved description. To account for potential hallucination in LLM's response, i.e., returning a type that does not exist in Python, AlphaTrans employs a simple Python script, uses the translated type as an annotation, executes the script, and keeps those without any syntactic or runtime issue. The types in the source language and their corresponding in the target language form a data structure called *Universal Type Mapping*. In practice, AlphaTrans reuses or augments the mapping when translating new projects.

5.2 Skeleton Construction

AlphaTrans builds the project's structure in the target language before translation. This step is necessary for compositional translation and validation, as AlphaTrans can insert the translated fragments into the project, compile it, or even execute the existing translated test suites, gradually completing the translation. At this step, AlphaTrans also resolves Java-specific features in Python before starting the translation. Specifically, it resolves circular imports and dependencies, inner

[3]Due to space limit, we could not include the prompt. Please refer to the artifacts to see the prompts used for type resolution.

classes, interfaces, and abstract classes. Figure 4 shows the class skeleton corresponding to the illustrative example of Figure 1-b.

At the first step, AlphaTrans creates classes corresponding to each application class in Java. The fields for Python classes are set to `None`, and AlphaTrans uses the information in schema (§4.2.1) to ensure the naming corresponds to the type of their access modifier in the source language. In the example of Figure 4, the translation of field `private Option option;` in Java is `__option: Option = None`. The classes also include method signatures, with their body set to `pass`. AlphaTrans uses the universal type mapping (§5.1) to use relevant types in the full method signature. Once the initial skeleton is created, AlphaTrans leverages the extracted call graph during program decomposition to detect circular dependencies. If there exist any, AlphaTrans resolves them with local imports. For inner classes, AlphaTrans unfolds them in Python and uses *dot notation* to access specific methods and fields (e.g., `Class.methodName`). Finally, AlphaTrans implements best practices in Python and sub-classes all abstract classes and interfaces from `abc.ABC` class. Here, `ABC` is a class from the `abc` module in the Python standard library, which is used for defining abstract base classes.

6 Compositional Translation and Validation

AlphaTrans translates method fragments in the reverse call order. It takes the project's call graph, removes the back edges to make it acyclic, computes topological order (i.e., linear ordering of its vertices), and translates method fragments (corresponding to vertices) in reverse topological order. Field fragments are independent; therefore, AlphaTrans translates them before method fragments. The algorithm takes a fragment F, LLM M, Skeleton S, and a series of parameters as inputs, translates the fragment, recomposes the skeleton with the successfully translated fragment, and returns translation outcomes: (1) syntax check (denoted by the "non-parseable" and "parseable" labels), (2) functional equivalence check (denoted by the "graal-fail", "graal-success", and "graal-error" labels), and (3) translated test execution check (denoted by "not-exercised", "test-fail", and "test-success" labels).

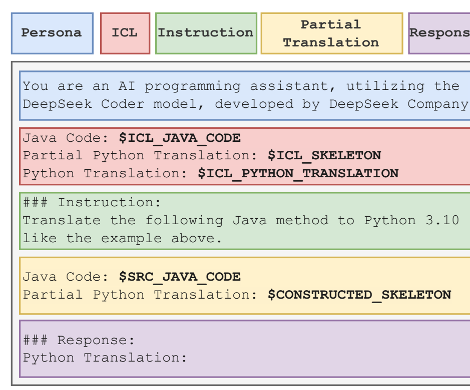

Fig. 5. Prompt structure in AlphaTrans.

AlphaTrans employs iterative and feedback-based prompting. That is, if one of the mentioned checks fails, e.g., the translated fragment is not syntactically correct, it prompts the model for another translation attempt. To control the number of iterations, AlphaTrans considers a reprompting budget, i.e., *repromptBudget*. The algorithm takes the minimum (min_{budget}) and maximum (max_{budget}) values for the budget and dynamically sets reprompting budget to a number between them based on the coverage information, e.g., the budget is close to max_{budget} for a fragment if it is exercised multiple times (high hit rate based on coverage information) by different unit tests. The rationale is to give more importance to fragments covered by more tests to eventually increase translation validation success. The main translation and validation loop (lines 3–31) runs until the assigned budget is exhausted.

Inside the loop, AlphaTrans first crafts a unique prompt based on the template shown in Figure 5 and then instructs the LLM to translate the fragment (lines 4–5). It then validates the generated translation in multiple steps. The first step checks for syntactical correctness and assigns proper labels to TVO (lines 6–10). Then, AlphaTrans leverages GraalVM for isolation-based validation of fragment F (lines 12–20), if there exists a test in the source language covering the fragment during its execution.

Finally, it translates and executes decomposed fragment tests: if there are no eligible tests (a test becomes eligible if all its dependencies are translated) for the fragment, AlphaTrans simply assigns the *"not-exercised"* label to the fragment and moves on to the next one (lines 21–23). Otherwise, it translates the tests, executes them to validate the fragment, and assigns test outcome labels in *TVO* (lines 24–31). In case of a test failure, AlphaTrans extracts all involved fragments and re-prompts them with feedback extracted from test execution.

Due to inherent intra- and inter-procedural dependencies in real-world projects, the number of fragments involved in re-prompting could be high, logarithmically increasing the translation time. AlphaTrans filters out those with GraalVM label *"graal-success"*, ranks the remaining based on suspiciousness score, and re-prompts *topK* suspicious fragments. The suspiciousness score for fragments is calculated such that a fragment with more failing tests will get a higher score and, therefore, ranked higher among other fragments.

Our prompt template consists of *five* distinct parts (Figure 5). The first part is the *persona* message used by DeepSeek-Coder-33b-Instruct during instruction fine-tuning and is required for producing the best outputs. The next part introduces the *In-Context Learning (ICL)* example, which reflects the complexities of code translation and instructs LLM on how to deal with them. The green part indicates the natural language instruction given to the model. After describing the objective, the prompt embeds the source Java code along with *partial translation* as a skeleton, which includes all dependencies and translations of the fragments invoked by the current one. The prompt concludes with `### Response:` special keyword to guide the model for code generation.

Algorithm 2: Main Translation Loop

```
   Inputs: Skeleton S, Fragment F, Model M, min_budget,
           max_budget, and topk suspicious methods
   Output: Translation and Validation Outcome TVO
1  feedback ← ∅; TVO ← {};
2  repromptBudget ←
     getAdaptiveBudget(F, min_budget, max_budget);
3  while repromptBudget > 0 do
4      prompt ← generatePrompt(F, feedback);
5      translation ← translateFragment(prompt, M);
6      if !syntacticCheck(translation) then
7          TVO["syntax_outcome"] ← "non − parseable";
8          feedback ← getFeedback(translation);
9          repromptBudget ← repromptBudget − 1;
10         continue;
11     TVO["syntax_outcome"] ← "parseable";
12     if !graalCheck(translation) then
13         TVO["graal_outcome"] ← "graal − fail";
14         feedback ← getFeedback(translation);
15         repromptBudget ← repromptBudget − 1;
16         continue;
17     else if graalLimitation(translation) then
18         TVO["graal_outcome"] ← "graal − error";
19     else
20         TVO["graal_outcome"] ← "graal − success";
21     if !hasEligibleTests(F) then
22         TVO["test_outcome"] ← "not − exercised";
23         break;
24     testTranslation ← getTestTranslation(F);
25     S ← S ∪ translation;
26     if !testCheck(S, testTranslation) then
27         TVO["test_outcome"] ← "test − fail";
28         repromptSuspiciousMethods(testTranslation, topK);

29         repromptBudget ← repromptBudget − 1;
30         continue;
31     TVO["test_outcome"] ← "test − success";
32     break;
33 return TVO;
```

6.1 Language Interoperability

GraalVM [41, 58] is a Java Development Kit by Oracle. It offers the *Polyglot* API [21], which allows the integration of programs written in different guest languages within a Java-based host application. In the context of this paper, GraalVM allows developers to execute Python code from Java and vice versa. AlphaTrans leverages the Polyglot API to perform *in-isolation* validation of the fragments by replacing the Java implementation of a method with its translated Python version while keeping the rest of the project in Java. It then executes the Java tests covering the fragment to validate the functional equivalence of the translation.

The Polyglot API allows access to Python data objects from Java and vice-versa, as these objects reside in a shared memory space. However, objects must be cast to appropriate types for passing parameters to and processing returned values from polyglot calls. The Polyglot API can perform this casting implicitly for only a few simple data types. AlphaTrans builds on top of the Polyglot API to provide a framework to create a Python program state that is isomorphic to the Java program state. The Python translation is restricted to this isomorphic state, and the states are synchronized after method calls to preserve the isomorphism. AlphaTrans allows for the casting of user-defined types as well as several built-in and library types. Using both the static and the dynamic type information of Java objects, AlphaTrans can disambiguate the target types when casting Python objects to Java types. It further preserves object identities and aliasing during such casting and propagates exceptions across language boundaries.

To validate the translation of a method m in isolation, AlphaTrans creates an instrumented version of the Java source code. We will refer to this instrumented Java project as the *primal project*, P_m. During instrumentation, AlphaTrans replaces the original Java implementation, m_J of m with a polyglot call to its Python implementation, m_P. m_P resides inside a Python project, which we will refer to as the *dual project*, D_m. The structure of D_m is similar to that of the original Java project. All other methods in D_m wrap a call to the corresponding methods in P_m. Doing so provides an interface for m_P to execute with access to all other methods and fields, although these are defined only in P_m. Using the call graph for the Java project, AlphaTrans determines all test methods that invoke m and executes them in P_m to validate the translation m_P. This *in-isolation* validation approach is limited in the sense that it can handle only a limited number of built-in and library types. In certain cases, like reference cycles involving maps or objects with impure methods for hashing, the isomorphism between Java and Python states may not be maintained. Furthermore, it may sometimes not be possible to disambiguate target types when casting Python objects to Java types, for example, if the target object has type `List<Object>`.

6.2 Target Program Recomposition

AlphaTrans recomposes the project skeleton with syntactically correct translated fragments at each iteration of Algorithm 1, gradually constructing the project in the target PL. The recomposed target program in Python is executed against eligible translated tests, i.e., those that cover only the translated fragments.

6.3 Test Translation

Similar to translating application fragments, AlphaTrans also translates test fragments. Using the dependency information captured during static analysis, it crafts prompts for unit tests along with their dependencies for the model to translate. The ICL examples used for prompting test fragments differ from prompts used for translating application method fragments. The focus of ICL examples here is to prevent LLM from hallucinating the usage of assert statements in the source to target PL. To construct ICL examples for test fragment translation, we created a pool of in-context examples, where each example shows the Python assert statements equivalent to Java assert statements in the context of a test. When prompting a test fragment, the AlphaTrans detects the assert statement in the fragment and retrieves the corresponding examples from the pool. For translated tests, only syntactic validation is performed as there is no other means of validating their translations.

7 Empirical Evaluation

To evaluate different aspects of AlphaTrans, we investigate the following research questions:

- **RQ1:** *Effectiveness.* To what extent AlphaTrans can automatically resolve types from source to target PL? Can AlphaTrans effectively translate real-world projects?
- **RQ2:** *Translation Bugs and Fixes.* How much effort do developers spend completing the partial translations by AlphaTrans? What is the nature of translation bugs?
- **RQ3:** *Impact of Test Decomposition.* What is the impact of test decomposition on validation results?

RQ4: *Impact of Test Coverage.* To what extent does a test suite with higher coverage impact the validation results?

RQ5: *Ablation Study.* To what extent do program transformation, choice of LLM, and program decomposition impact the performance of AlphaTrans?

7.1 Experiment Setup

We followed three steps for selecting subjects: **1- Mining:** We mined GitHub and retrieved a list of repositories that use Java as the primary language, are self-contained (include build files, etc.), and have more than 30 stars with at least one commit pushed within the last 12 months. **2- Filtering:** We filtered out projects based on the number of call edges in their call graphs: removed those with less than 2,000 call edges to ensure the subject projects are big enough to challenge AlphaTrans. We also removed those with more than 30,000 call edges to reduce the computation and manual effort for further steps. Per GraalVM requirements, we only selected projects we could successfully build (compile and achieve green tests) using Java at 21. **3- Reduction:** AlphaTrans currently supports the following Java APIs: core Java API (`java.util`, `java.text`, `java.lang`, `java.io`, `java.nio`, `java.net`, `java.time`, and `java.math`) and third-party libraries (`org.opentest4j`, `org.slf4j.Logger`, and `org.junit`). We automatically removed all other third-party library dependencies and their usage in the source code in the selected projects. We chose a project if at least 50% of its total methods were preserved after such process. Table 1 shows the list of ten projects used in the evaluation of AlphaTrans and details about their size (classes, methods, tests, and fragments).

AlphaTrans uses CodeQL [20] and tree-sitter [57] for static analysis. For running tests, validating translation, and computing coverage, AlphaTrans uses GraalVM 21.0.3 + 7.1 [41], JUnit 4 and 5 [52], Pytest 8.2.1 [45], JaCoCo [51], and Python's `coverage` [5]. AlphaTrans works with API- and open-access LLMs. We considered the following criteria for selecting the LLM: (1) for better reproducibility, we prioritized open-access models; (2) due to computing constraints, we wanted an LLM with moderate size ($< 10B$ but $< 70B$ parameters); (3) the model should perform reasonably well in code-related tasks; and (4) the model should have fast inference time due to the huge number of prompts. Per the mentioned criteria, we selected DeepSeek-Coder-33b-Instruct for main experiments (RQ1–RQ4). We also used GPT-4o, one of the best-performing commercial models, in RQ5 (§7.6) to demonstrate the impact of stronger models on improving the performance of AlphaTrans. We prompted models under the temperature 0 setting for reproducibility and used their default setting for other parameters. For the base prompting (Algorithm 2), we set the minimum and maximum values of the reprompting budget to 3 and 5. For the feedback prompting, we set the reprompting budget to 1, i.e., AlphaTrans attempts to fix issues with feedback only once.

7.2 RQ1: Effectiveness of AlphaTrans

In this RQ, we evaluate AlphaTrans in (1) type translation and skeleton construction (§7.2.1) and (2) compositional translation and validation (§7.2.2).

7.2.1 Effectiveness in type resolution and skeleton validation: AlphaTrans extracted 1,797 distinct types from the source projects and attempted to translate them to equivalent Python types. Of these, 915 are application-level types (e.g., classes defined within the Java projects) and were directly resolved during skeleton construction. AlphaTrans prompts DeepSeek-Coder-33b-Instruct to resolve the remaining 882, out of which it successfully translated 738 ($(915 + 738)/1,797 = 91.99\%$) of them: generated types passed the syntactic and runtime check. The column *ATR* in Table 1 shows the results of automated type resolution. Since type resolution is essential to project skeleton construction, we manually checked the type mappings generated by DeepSeek-Coder-33b-Instruct, and also attempted to translate the remaining unresolved 144 types.

Table 1. Effectiveness of AlphaTrans in program transformation, automated type translation, and skeleton validation. **ATR:** Automated Types Resolution, **SV:** Skeleton Validation. The number of classes and methods include both application and test classes/methods.

Subjects	# Classes	# Methods	# JUnit Tests	Method Coverage (%)	ATR (%)	SV (%)	# Fragments			
							Fields		Application Methods	Test Methods
							Application	Test		
cli [11]	58	664	437	94.14	96.60	100	104	57	273	2180
codec [12]	156	1780	992	91.03	96.01	100	425	140	680	2849
csv [13]	41	694	309	90.64	92.34	100	146	35	235	1272
exec [14]	56	407	70	54.84	78.90	100	104	27	248	327
fast-pfor [34]	82	971	82	54.55	87.40	100	127	14	748	302
fileupload [15]	49	381	39	13.02	98.31	100	121	39	192	194
graph [16]	118	879	146	58.78	97.13	100	216	29	541	975
jansi [49]	48	474	107	23.47	84.83	100	378	0	409	123
pool [17]	98	1097	73	22.29	91.60	100	203	91	682	649
validator [18]	130	1228	464	63.31	95.51	100	421	209	646	1463
Total	836	8575	2719	56.57	91.99	100	2245	641	4654	10334

Through manual investigation of the automatically resolved types, we observed that DeepSeek-Coder-33b-Instruct's type resolution for 182 cases, while *correct*, can be *improved*. For example, AlphaTrans translated `java.io.File`, a class concerning file manipulation functionality to `str`. The resolved type can represent file paths in Python but lacks features for file manipulation. We suspect this translation is impacted by the Java use case provided in the prompt. While this translation is correct concerning the use case, we replaced it with `pathlib.Path` to have a more generic type mapping. We also augmented the type mapping with additional types in the target language for 38 types. For example, AlphaTrans translated `java.nio.Buffer` to `bytearray`, which is correct as they both provide a mutable sequence of bytes with efficient in-place modifications. However, `array.array` and `memoryview` also provide similar functionality with efficient and low-level data manipulation capabilities. Consequently, we augmented type mapping to `typing.Union[bytearray, array.array, memoryview]`. Given that type mapping can be reused, this one-time manual effort increases the chance of AlphaTrans's success on unseen projects.

Using the universal type mapping, AlphaTrans successfully creates and validates project skeletons in target PL, achieving 100% syntax and runtime validation (column *SV* in Table 1). The skeleton validation step ensures all module imports, class structures, method signatures, and type annotations are done properly, making the subsequent steps easier. Applying AlphaTrans to unseen projects, if a class skeleton cannot be validated, AlphaTrans removes it from the target project, updates the skeleton based on the class dependencies, and proceeds to the next phase.

Summary. AlphaTrans can successfully transform projects to remove method and constructor overloading. Moreover, it can automatically translate 91.99% of the source PL types and use that to create and validate project skeletons in the target PL.

7.2.2 Effectiveness in compositional translation and validation: Table 2 shows the compositional translation and validation results. The *AMF* column indicates the total number of application method fragments. The numbers in subsequent columns demonstrate the effectiveness of AlphaTrans in the translation and validation of *AMF*s only[4]. The *Syntax Check* column indicates the percentage of *AMF*s that pass syntactic validation. Column *SNEF* shows the percentage of *AMF*s not covered by JUnit tests. AlphaTrans successfully generates syntactically correct code (98.80% of AMFs and 96.40% of all fragments—field, application, and test fragments—across ten subjects). We also

[4]Due to space limit, please refer to our artifact [32] for details of all the translation and validation of other fragments shown in Table 1.

Table 2. Effectiveness of AlphaTrans in repository-level code translation. Abbreviations in the table stand for **AMF**: #Application Method Fragments, **SNEF**: Source Non-Exercised Fragments, **GS**: Graal Success, **GF**: Graal Fail, **GE**: Graal Error, **TNEF**: Target Non-Exercised Fragments, **ATP**: Fragments All Test Pass, **OTF**: Fragments One Test Fail, **MTF**: Fragments Many Test Fail, **ATF**: Fragments All Test Fail, **TPR**: Test Pass Rate, **O**: Overall, **RE**: Runtime Error, **AF**: Assertion Failure, and **M1**: Number of *AMFs* that GraalVM could not execute (*GE*) but translated test fragments exercised.

Subjects	AMF	Syntax Check (%)	SNEF (%)	GraalVM			Test Translation												M1	
				GS (%)	GF (%)	GE (%)	TNEF (%)	ATP (%)	OTF (%)			MTF (%)			ATF (%)			TPR (%)	All	Some
									O	RE	AF	O	RE	AF	O	RE	AF			
cli	273	100	5.86	70.70	11.72	11.72	35.90	8.42	10.62	51.72	48.28	32.23	91.07	8.93	6.96	100	0	10.08	0	16
codec	680	98.53	8.97	38.38	32.50	20.15	65.59	4.12	4.41	60.00	40.00	12.79	55.11	44.89	4.12	75.21	24.79	9.43	11	27
csv	235	98.72	9.36	38.72	26.81	25.11	74.47	0	13.62	96.88	3.13	0	0	0	2.55	100	0	0	0	3
exec	248	100	45.16	33.47	2.02	19.35	34.27	4.44	2.02	40.00	60.00	7.26	93.36	6.64	6.85	100	0	19.29	6	9
fast-pfor	748	95.32	45.45	12.03	24.20	18.32	41.71	4.28	1.74	84.62	15.38	3.74	79.23	20.77	3.07	85.88	14.12	20.08	6	25
fileupload	192	100	86.98	8.85	1.04	3.13	1.56	3.65	6.77	30.77	69.23	1.04	91.67	8.33	0	0	0	63.44	2	3
graph	541	99.63	41.22	24.77	22.92	11.09	57.12	0	0.92	100	0	0.18	100	0	0.55	100	0	11.04	0	1
jansi	409	99.76	76.53	8.07	11.49	3.91	22.25	0.24	0.98	100	0	0	0	0	0	0	0	1.07	0	1
pool	682	100	77.71	6.01	1.32	14.96	19.35	1.61	1.03	100	0	0.29	100	0	0	0	0	6.62	4	2
validator	646	99.23	36.69	30.50	11.15	21.67	46.44	3.25	5.42	71.43	28.57	4.18	84.50	15.50	4.02	99.12	0.88	11.70	1	31
Total	4654	98.80	43.43	24.50	16.23	15.84	41.92	2.88	3.72	70.52	29.48	5.44	82.77	17.23	2.62	92.86	7.14	9.76	30	118

observe that 43.43% of *AMF*s are not covered during the execution of any test, i.e., we cannot go beyond syntactic check and validate their runtime behavior or functional equivalence.

For 56.57% of *AMF*s that can be covered by source project tests, AlphaTrans attempts to validate the functional equivalence using GraalVM. The super column *GraalVM* shows the percentage of *AMFs* that GraalVM executes and successfully validates (*GS*), executes but there is a test assertion failure (*GF*), and cannot execute due to its limitation (*GE*) mentioned in §6.1. 24.50% (min=6.01% and max=70.70%), 16.23% (min=1.04% and max=32.50%), and 15.84% (min=3.13% and max=25.11%) of *AMFs* resulted in *Graal Success*, *Graal Fail*, and *Graal Error*, respectively. Note that these numbers add up to 56.57% of *AMFs* that were covered by source program tests. With respect to only covered methods, GraalVM Success will be 43.30%. Furthermore, our analysis shows that a high portion of methods that are not covered by tests are either abstract methods or getter/setter methods. If their translations are syntactically correct, they are also likely functionally equivalent, which can ramp up the success rate (§7.5). Spearman Rank Order Correlation [50] indicates a strong positive correlation between the method coverage (Table 1) and *GS* numbers (ρ = 0.92), confirming that with better method coverage, the validated *AMF*s are very likely to be higher.

Regardless of GraalVM's validation for functional equivalence, AlphaTrans translates and executes the test fragments on the recomposed translated project. Super column *Test Translation* shows the results of test translation and execution. Column *TNEF* indicates the ratio of *AMF*s where execution of translated tests never reached them. Columns *ATP* through *ATF* show the number of *AMFs* that AlphaTrans executed using translated test fragments, categorized per the test execution results. For 2.88% of *AMFs*, all the test fragments that covered them were marked as pass (*ATP*). For 3.72% and 5.44%, at least one (*OTF*) or more than one test (*MTF*) failed. All the test fragments failed for 2.62% of them. Note that these numbers (TNEF, ATP, OTF, MTF, and ATF) add up to 56.57% of AMFs covered by source program tests.

For cases with test failure, columns *RE* and *AF* show the breakdown of whether test failure was due to assertion failure or runtime error. We can observe that most test executions terminated with a runtime error due to translation bugs and never reached the assert statement. *Our manual investigation confirms that a high rate of runtime errors is due to a relatively small number of fragments with translation bugs.* Although test decomposition helps with *test translation coupling effect* (§2), there is still a high degree of runtime errors due to the long call chains in these projects (the average number of methods executed per test in the original and decomposed test suites are 27.4 and 21.8).

As a result, the overall pass rate, i.e., the percentage of recomposed test fragments for the translated projects that pass (column *TPR*), is low. These results also confirm the necessity of in-isolation testing through language interoperability by AlphaTrans.

We also calculated the number of *AMFs* that GraalVM could not execute (numbers under *GE* column) but translated test fragments exercised (column *M1*). *All* indicates the number of *AMF*s with all passing tests, including test fragment with assert statement, indicating the validation of functional equivalence. *Some* corresponds to the number of *AMF*s with at least one passing test, which indicates runtime validation. Overall, test translation can validate the functional correctness and runtime behavior of 30 and 118 fragments that GraalVM could not exercise.

Finally, we analyzed translations using PyLint [2], which scores Python files on a scale of 0 to 10 based on how Pythonic the code is. All AlphaTrans translations achieved scores of 10, mainly because (1) LLMs inherently generate idiomatic code and (2) AlphaTrans uses Black [1] for formatting the translations extracted from the LLM response. These results confirm that the translations are all Pythonic, i.e., they adhere to coding standards and best practices.

Summary. AlphaTrans effectively performs compositional translation and validation of 17, 874 fragments, achieving overall 96.40% syntactical correctness (98.80% for AMFs), 27.03% runtime behavior validation (GS+M1 Some), and 25.14% functional equivalence (GS+M1 All).

7.3 RQ2: Translation Bugs and Fixes

We investigated the manual effort for fixing translation bugs in a subset of studied subjects, namely *Commons-FileUpload*, *Commons-CLI*, *Commons-CSV*, and *Commons-Validator*. We also discuss some of the translation bugs and fixes for them to better illustrate the challenges in code translation.

7.3.1 Human Study. Our two human subjects were selected due to their relative familiarity with the selected projects. Their effort indicates an upper bound for the amount of time required to fix translation bugs since developers of the source projects are likely to fix the bugs better and faster. We shared with them the source program in Java, the translations by AlphaTrans, and all the reports and artifacts generated by AlphaTrans during translation.

For *Commons-FileUpload*, achieving green tests took 5.5 hours and required 120 and 114 line additions and deletions from partial translations. For *Commons-CLI*, the manual fix took 11 hours, making 614 and 1, 253 line additions and deletions, respectively. The project was very dense for *Commons-CSV*, with many method calls, making it harder to manually fix bugs. Nevertheless, a developer achieved all green tests in 30 hours with 2, 676 and 999 line additions and deletions, respectively. Finally, for *Commons-Validator*, the developer spent 34 hours to fix translation bugs, with 3, 585 and 2, 416 line additions and deletions, respectively. One of the major feedback from developers was that test decomposition greatly helped locate and fix translation bugs: in case of a test failure, developers only need to investigate the last call statement in the failed test fragment instead of looking at the stack trace and other prior calls.

7.3.2 Translation Bugs. Our artifacts [32] contain partial translations and fixed versions as separate commits. These commits can serve as useful benchmarks for evaluating fault localization, program repair, and test generation techniques. This section shows *four* instances of such translation bugs.

The two most prevalent sources of translation bugs are mismatches between APIs and behavioral differences in the PLs. The code snippet below demonstrates a bug that happened due to a mismatch in the logic of `Calendar` (Java) and `datetime` (Python). Line 3 in Java sets the `MONTH` field to `0`, which corresponds to the first month of the year (*January*). Similarly, the Python translation sets the `month` attribute to `0`; however, in the Python library, *January* is the first month, i.e., the correct translation should use index `1`.

```
----------- JAVA SOURCE CODE -----------
Calendar calendar = Calendar.getInstance()
calendar.set(Calendar.MONTH, 0);
```

```
---------- PYTHON TRANSLATION ----------
  calendar = datetime.datetime.now()
- calendar = calendar.replace(month=0)
+ calendar = calendar.replace(month=1)
```

The next example shows the difference in implicit type casting between the two Pls. Line 5 in Java source code concatenates a `String` with `nullStr = null`. During execution, Java runtime silently casts `null` to a `String` and then performs the binary operation on it. In Python, concatenating an `str` with `None` results in a `TypeError` as the operands of the binary operation has different types. A correct Python translation requires explicit casting of `None` to `str` as shown in Line 5.

```
1          ----------- JAVA SOURCE CODE -----------
2 qChar = "'";
3 nullStr = null;
4
5 this.qNullStr = qChar + nullStr + qChar;
```

```
1         ---------- PYTHON TRANSLATION ----------
2   qChar = "'"
3   nullStr = None
4 - self.qNullStr = qChar + nullStr + qChar
5 + self.qNullStr = qChar + str(nullStr) + qChar
```

The third example shows an instance of `write(int b)` method from `ByteArrayOutputStream` class, where the least significant 8 bits of the integer `(b2 « 4) | (b3 » 2)` are directly written to the stream. The incorrect Python translation attempts to construct a `bytes` object using a singleton list with the input integer before writing it to an object of `io.BytesIO`. However, this neglects that the `bytes()` constructor requires the integers in the input iterable to be strictly in the range of `[0, 255]`. Thereby, a `ValueError` is thrown when `b2` is large. The correct translation requires `0xF`, a mask that maintains only the 4 lowest bits of `b2` before left-shifting by 4 as shown in Line 3 under Python translation. Given that `b3` and `b4` each contain no more than 8 bits, this change ensures the least significant 8 bits of `(b2 « 4) | (b3 » 2)` are correctly written to the `BytesIO` object.

```
1          ----------- JAVA SOURCE CODE -----------
2
3 out.write((b2 << 4) | (b3 >> 2));
```

```
1         ---------- PYTHON TRANSLATION ----------
2 - out.write(bytes([(b2 << 4) | (b3 >> 2)]))
3 + out.write(bytes([((b2 & 0xF) << 4) | (b3 >> 2)]))
```

The last code snippet demonstrates the Java behavior of an `iterator` that is unavailable in Python. The incorrect Python translation uses `next()` to implement both `.next()` and `.hasNext()` APIs. However, calling `next()` increments the iterator in Python. The correct translation should implement `PeekableIterator` interface in Python with a method `hasNext() -> bool`.

```
----------- JAVA SOURCE CODE -----------
Iterator<String> headers = ls.keySet().iterator();

assertEquals("content", headers.next());

assertFalse(headers.hasNext());
```

```
---------- PYTHON TRANSLATION ----------
- headers = iter(ls.keys())
+ headers = PeekableIterator(ls.keys())
  self.assertEqual("content", next(headers))
- self.assertFalse(next(headers, None) is not None)
+ self.assertFalse(headers.hasNext())
```

Implications. In order to obtain correct translation, especially for translating APIs, models need to generate test cases as well, which will validate the translated fragment in isolation. This could be an interesting direction for applying the agentic approach, where the orchestrating agent can decide when to generate test cases, and the test case generator agent gets all the information by running static analysis tools, gathering context from previous runs, collecting API documentation by crawling internet, and can finally generating the translation based on all the information.

Summary. Although AlphaTrans cannot validate all the translations, it provides partial translations and artifacts that developers can use to complete the translation and achieve green tests in a reasonable time (20.1 hours, on average).

7.4 RQ3: Impact of Test Decomposition

We previously showed the effectiveness of test translation in validating the runtime behavior or functional correctness of application method fragments (§7.2.2). To better understand how test decomposition helps *test translation coupling effect*, we collected translated unit tests with at least two corresponding decomposed test fragments. We further filtered out these unit tests and kept those that *all* their decomposed test fragments were executed, regardless of passing or failing. The

Table 3. Effectiveness of test augmentation in exercising and validating more application method fragments. Abbreviations in the table stand for **ATP**: Fragments All Test Pass and **TPR**: Test Pass Rate. ATP+ and TPR+ demonstrate ATP and TPR gain through test augmentation.

Subjects	Developer-Written Test					EvoSuite Test				
	Method Coverage (%)	# Decomposed Tests	Avg. Methods Executed / Test	TPR (%)	ATP (%)	Method Coverage (%)	# Tests	Avg. Methods Executed / Test	TPR+ (%)	ATP+ (%)
cli	94.14	3036	34.25	10.08	8.42	95.97	569	12.15	2.99	1.47
codec	91.03	3522	10.56	9.43	4.12	80.74	1141	8.02	3.51	0.88
csv	90.64	1219	52.62	0	0	74.04	220	39.16	0	0.00
exec	54.84	311	18.99	19.29	4.44	61.29	245	6.32	3.27	1.21
fast-pfor	54.55	249	41.62	20.08	4.28	39.17	1843	4.31	5.59	1.07
fileupload	13.02	93	3.54	63.44	3.65	70.31	231	5.29	11.26	2.60
graph	58.78	933	25.02	11.04	0.00	76.71	800	9.00	5.13	0.92
jansi	23.47	187	13.57	1.07	0.24	51.83	332	9.08	3.31	0.73
pool	22.29	287	6.52	6.62	1.61	37.24	394	7.36	10.41	2.20
validator	63.31	1479	11.68	11.70	3.25	81.42	1305	13.43	9.73	7.59
Total	56.57	11316	21.84	9.76	2.88	66.87	7080	11.41	5.85	2.11

yellow bars in Figure 6 show the percentage of our selected unit tests from the translated test suites. None of the tests met the criteria for *Commons-CSV*, so we excluded it from further investigation.

We categorized the selected unit tests into two groups: Those with all their test fragments pass (green bars in Figure 6), and those with at least one test fragment fail (red bars in Figure 6). For the *unit tests* in the latter group, we calculate the pass rate of the *decomposed test fragments*. The blue box chart in Figure 6 shows the distribution of the measured pass rate per unit test. These unit tests would have been marked as *fail* without test decomposition. However, we can observe that these tests can be decomposed into test fragments that 62.41% of them pass. These results confirm how decomposed test fragments were useful in helping developers localize translation bugs more easily and resolve the translation bugs faster.

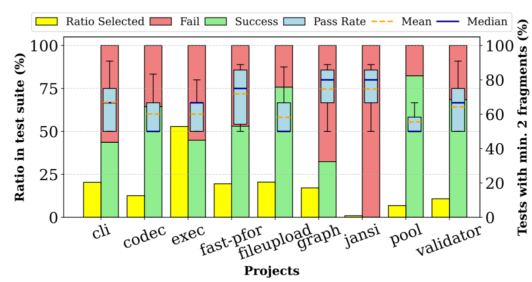

Fig. 6. Effectiveness of test decomposition in AlphaTrans for validating earlier fragments in failing tests.

Summary. Test decomposition unburdens validation of application method fragments from incorrect translations. 62.41% of test fragments for unit tests that would have been marked as failed achieved a test pass.

7.5 RQ4: Impact of Test Coverage

While existing developer-written tests are useful for validating functional equivalence, they can pose two major issues for automated code translation and validation. First, the coverage for these tests can be extremely low (e.g., 13.02% for Commons-FileUpload [15] reported in Table 1), preventing most of the code from being validated (as shown in RQ1, the translation validation rate strongly correlates with test suites' (method) coverage). Second, executing a developer-written test can have a long call sequence. To show the positive impact of better, more focused tests with higher coverage on translation validation, we automatically generated additional tests using EvoSuite [19]. We used DynaMOSA [43] as an optimization algorithm and set a timeout of 120 seconds as a stopping criterion for the generation of evolutionary tests.

Table 3 compares the developer-written and EvoSuite tests' properties. Evosuite tests cover more methods than the developer-written test suite (66.87% *Method Coverage* compared to 56.57%).

Table 4. Importance of program transformation. Abbreviations in the table stand for **AMF**: #Application Method Fragments, **SNEF**: Source Non-Exercised Fragments, **GS**: Graal Success, **GF**: Graal Fail, **GE**: Graal Error, **TNEF**: Target Non-Exercised Fragments, **ATP**: Fragments All Test Pass, **OTF**: Fragments One Test Fail, **MTF**: Fragments Many Test Fail, **ATF**: Fragments All Test Fail, and **TPR**: Test Pass Rate.

Subjects	AMF	Syntax Check (%)	SNEF (%)	GraalVM			Test Translation					
				GS (%)	GF (%)	GE (%)	TNEF (%)	ATP (%)	OTF (%)	MTF (%)	ATF (%)	TPR (%)
cli	276	100	5.80	0	0	94.20	85.51	0	2.17	1.09	5.43	0.46
codec	678	98.82	10.62	0	0	89.38	72.86	2.06	5.46	2.80	6.19	5.04
csv	235	100	8.51	0	0	91.49	88.09	0.43	0.43	0.85	1.70	0.32
exec	253	100	46.25	26.88	4.35	22.53	49.41	1.58	1.19	0	1.58	4.29
fast-pfor	754	98.14	46.02	0	0	53.98	38.20	0.93	6.63	0.80	7.43	4.88
fileupload	168	100	85.12	10.71	0.60	3.57	8.93	1.19	2.98	1.19	0.60	23.08
graph	556	99.82	40.65	0	0	59.35	56.12	0	1.98	0.36	0.90	5.48
jansi	314	99.68	69.75	0	0	30.25	29.94	0	0.32	0	0	0
pool	574	99.83	70.56	0	0	29.44	27.00	1.39	0	0.35	0.70	5.48
validator	623	99.20	33.39	18.78	20.06	27.77	66.29	0	0	0.32	0	0.43
Total	4431	99.32	40.01	4.58	3.09	52.31	52.79	0.81	2.57	0.86	2.96	3.05

Furthermore, the average number of methods executed per single test is almost half that of decomposed test suites (11.41 compared to 21.84 methods). To demonstrate the impact of test quality on AlphaTrans's overall performance, we translated and executed the EvoSuite-generated tests. Corroborated by the numbers under *TPR+* and *ATP+* columns, we can see that augmenting the test suite increases the method coverage, and thereby, the TPR and ATP numbers from RQ1 by 5.85% and 2.11%, respectively. Not all the EvoSuite tests have assertions, and even if they do, the quality of the assertions could be lower compared to developer-written tests (e.g., checking trivial properties that result in a test pass). Thereby, we only claim that higher *TPR* and *ATP* of such tests enhance runtime validation, which is still promising in code translation. EvoSuite is incompatible with Java 21, and hence GraalVM, preventing us from using Evosuite-generated tests in AlphaTrans. We anticipate that including it in the loops could have also improved the overall quality of translations.

Summary. Augmenting the existing test suite increases code coverage, thereby exercising and validating more AMFs. Test augmentation can further validate the correctness of 2.11% of fragments not executed by developer tests. The generated tests are more focused and, on average, invoke 48% fewer methods than the developer-written tests.

7.6 RQ5: Ablation Study

We performed three ablation studies to investigate the impact of program transformation, choice of LLM, and program decomposition on the performance of AlphaTrans.

7.6.1 Impact of Program Transformation. We removed the program transformation component of AlphaTrans and executed the entire pipeline. The results in Table 4 show that without transformation (e.g., resolving method/constructor overloading), the performance of AlphaTrans drops significantly: *GS*, *ATP*, and *TPR* values decreased to 4.58% (from 24.50%), 0.81% (from 2.88%), and 3.05% (from 9.76%), respectively. This is because Python does not support overloading and only considers the last method/constructor implementation, resulting in runtime errors or test failures. Also, *GE* values increase due to the interference of overloaded code constructs with GraalVM.

7.6.2 Choice of LLM. For this experiment, we replaced the DeepSeek-Coder-33b-Instruct with GPT-4o and repeated the entire pipeline of AlphaTrans (Table 5). A stronger model such as GPT-4o improves the translation quality —functional equivalence increases from 25.14% to 27.95%. For some projects, the ATP and TPR rates are higher for DeepSeek-Coder-33b-Instruct translations, while for the others, GPT-4o results in higher values. We investigated each LLM's successful AMF

Table 5. Effectiveness of AlphaTrans with GPT-4o in compositional translation and validation. Abbreviations in the table stand for **AMF**: #Application Method Fragments, **SNEF**: Source Non-Exercised Fragments, **GS**: Graal Success, **GF**: Graal Fail, **GE**: Graal Error, **TNEF**: Target Non-Exercised Fragments, **ATP**: Fragments All Test Pass, **OTF**: Fragments One Test Fail, **MTF**: Fragments Many Test Fail, **ATF**: Fragments All Test Fail, **TPR**: Test Pass Rate, **O**: Overall, **RE**: Runtime Error, **AF**: Assertion Failure, and **M1**: Number of *AMFs* that GraalVM could not execute (*GE*) but translated test fragments exercised.

Subjects	Syntax Check (%)	SNEF (%)	GraalVM			Test Translation											M1		Cost ($)
			GS (%)	GF (%)	GE (%)	TNEF (%)	ATP (%)	OTF (%)			MTF (%)			ATF (%)			TPR (%)	All	Some	
								O	RE	AF	O	RE	AF	O	RE	AF				
cli	99.63	5.86	76.92	8.79	8.42	78.39	4.76	3.66	90.00	10.00	6.23	100	0	1.10	100	0	2.47	0	1	19.69
codec	97.79	8.97	42.50	28.53	20.00	74.56	3.82	4.12	78.57	21.43	6.18	30.62	69.38	2.35	73.33	26.67	9.57	2	18	39.97
csv	98.30	9.36	41.28	25.96	23.40	86.81	0	0	0	0	1.70	78.17	21.83	2.13	96.72	3.28	0.98	0	8	16.66
exec	99.60	45.16	35.89	2.42	16.53	44.76	4.84	0	0	0	5.24	0	100	0	0	0	28.62	1	1	3.90
fast-pfor	97.46	45.45	14.71	16.98	22.86	52.94	1.47	0	0	0	0	0	0	0.13	100	0	2.41	2	0	12.93
fileupload	99.48	86.98	9.90	0.52	2.60	4.17	6.77	1.56	0	100	0.52	100	0	0	0	0	54.84	1	3	2.25
graph	98.71	41.22	27.73	19.04	12.01	58.78	0	0	0	0	0	0	0	0	0	0	0	0	0	12.03
jansi	99.76	76.53	8.07	11.98	3.42	23.47	0	0	0	0	0	0	0	0	0	0	0	0	0	3.66
pool	99.71	77.71	7.48	1.32	13.49	21.70	0.59	0	0	0	0	0	0	0	0	0	2.09	0	0	7.33
validator	99.23	36.69	38.24	15.94	9.13	54.95	4.02	2.32	93.33	6.67	1.55	70.59	29.41	0.46	100	0	10.41	0	3	25.53
Total	98.80	43.43	27.83	14.54	14.20	50.64	2.26	1.20	80.36	19.64	1.87	59.39	40.61	0.60	86.59	13.41	6.45	6	34	143.95

translations to better understand the differences. We observed a huge overlap between successful AMFs and the unique benefits each LLM provides in code translation (Figure 7). GPT-4o handles API translation and type casting better, resolving the first three translation bugs discussed in §7.3. In contrast, it tends to add unnecessary code, mostly due to error handling, which results in a functional mismatch. In the example below, `create2` method can take a None variable, and its implementation performs error handling when necessary. GPT-4o adds unnecessary error handling code, interfering with program logic and resulting in test failures.

```
----------- JAVA SOURCE CODE -----------
public Option create1(OptionGroup optGroup) {
    return create2(optGroup);

}
```

```
---------- PYTHON TRANSLATION ----------
def create1(self, optGroup: OptionGroup) -> Option:
+   if optGroup is None:
+       raise ValueError("optGroup is None")
    return self.create2(optGroup)
```

It is worth noting that using commercial LLMs comes at a cost. The last column of Table 5 (column *Cost*) shows the cost of using GPT-4o for repeating the experiments, resulting in the total cost of $143.95 for translating all the subjects (average cost of $14.39 per project).

7.6.3 Impact of Program Decomposition. For this ablation study, we prompted GPT-4o and DeepSeek-Coder-33b-Instruct *file-by-file* and evaluated translation correctness through test execution and GraalVM (Table 6). Not surprisingly, a considerable percentage of the files exceeded the model context window size, particularly for DeepSeekCoder, with 9.08% of the files encountering this problem. Among the prompted files, 21.36% (19.44% for DeepSeekCoder and 1.92% for GPT-4o) were syntactically incorrect. Translations that passed the syntactic correctness check did not pass any translated test execution, whereas Graal validated 2.56% of the files in total (1.02% for DeepSeekCoder and 1.54% for GPT-4o). Note that file-level Graal validation does not mean that all the methods are correctly translated and validated; only the methods in the class are executed by tests. Manual analysis of the results of this experiment revealed that one prominent culprit of test failures was LLM hallucination with method/variable names. Given that we had no skeleton construction in this baseline, such issues could not be avoided; this demonstrates the usefulness of skeleton construction and incremental translation in AlphaTrans.

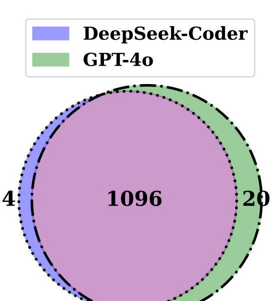

Fig. 7. Functional equivalence intersection of AMFs (Graal Success) between LLMs.

Table 6. Importance of program decomposition. Abbreviations stand for **GS**: Graal Success and **TPR**: Test Pass Rate. Since Java can contain multiple classes in one file, #Files is smaller from #Classes in Table 1.

Subjects	# Files	GPT-4o				DeepSeek-Coder			
		Over Context	Syntax Error	GS	TPR (%)	Over Context	Syntax Error	GS	TPR (%)
cli	51	0	0	1	0	5	6	0	0
codec	136	0	1	7	0	28	46	4	0
csv	33	0	3	0	0	6	8	0	0
exec	54	0	0	0	0	1	4	0	0
fast-pfor	85	1	1	1	0	12	30	1	0
fileupload	43	0	0	0	0	2	4	0	0
graph	159	0	6	3	0	0	17	3	0
jansi	30	0	0	0	0	3	10	0	0
pool	72	0	1	0	0	4	8	0	0
validator	119	0	3	0	0	10	19	0	0
Total	782	1	15	12	0	71	152	8	0

Summary. Omitting program transformation and program decomposition significantly lowers the effectiveness of AlphaTrans. A stronger model such as GPT-4o resolves non-trivial issues concerning type casting and API translation but may result in trivial translation bugs. When possible, users of AlphaTrans can prompt multiple LLMs to achieve better translation performance.

8 Related Work

There are generally two main approaches to translating code from one programming language to another: using transpilers and statistical machine translation and leveraging language models.

8.1 Code translation using non-LLM-based approaches.

In this domain, tools like C2Rust [25], CxGo [56], Sharpen [44], and Java2CSharp [26] have been developed to translate code from C to Rust, C to Go, and Java to C# respectively. However, a recent study [42] revealed that, apart from C2Rust and CxGo, other tools lack proper maintenance. For CxGo, language models outperform traditional approaches, whereas for C2Rust, language models generate safer but less effective code, aligning with the primary goal of translating C code to Rust. In terms of statistical machine translation, works by Nguyen *et al.* [37–39], and Chen *et al.* [7] focus on translating Java to C#. Additionally, deep learning approaches have been utilized for code translation [47, 48]. However, none of these efforts have tackled the translation of real-world Java projects to Python.

8.2 Code translation using LLMs.

Recently, large language models have been employed for code translation [8, 29, 42, 55, 59, 62], demonstrating high success rates on crafted examples but poor performance on real-world projects. Other studies [3, 63] have also utilized language models for code translation, mainly focusing on crafted benchmarks. Recently, there have been works that use transpiler output to guide the code translation [60]. However, the limitation of such work is the availability of robust and well-maintained transpilers, which, in many cases, may not be a feasible solution. Nitin et al. [40] introduced a specification-based translation, where natural language specification has been captured from the source code, which helps the translation process. Whereas Yang et al. [61] used tests to assist the translation. However, compared to previous works, the major differences are (a) the first attempt to translate a real-world project, (b) modular translation, and (c) a validation-guided translation approach.

9 Threats to Validity

Like most approaches, AlphaTrans possesses some limitations and comes with a list of threats to the validity. In this section, we will discuss how we mitigated various threats.

External Validity. One of the key external threats is the generalizability of AlphaTrans. We built and evaluated the first version of AlphaTrans to translate from Java to Python. However, our pipeline is generic, and with minimal effort, the current implementation can translate Java programs to more target languages (e.g., languages supported by GraalVM). Furthermore, the majority of the tools that we used support a large set of programming languages such as JavaScript, Ruby, C/C++, and Rust. Another threat is that AlphaTrans removes certain third-party libraries before translation. Translating repository-level code along with their libraries, especially when the target language does not have equivalent libraries, is a challenging problem that we plan to address in our future work.

Internal Validity. One threat can be the manual validation of the translated types. To address that, several authors have verified the types individually and consulted API documents when necessary. Another threat is that, while all successes reported by the GraalVM validation are true successes, we may have *underestimated* the capabilities of AlphaTrans by a considerable margin due to the significant proportion of errors caused by limitations in the GraalVM validation approach. To mitigate this threat, we manually augment the universal type map to support a comprehensive translation of types. That said, we have implemented *driver* code templates to provide a mechanism for adding the support for more types by the users of AlphaTrans if needed.

Construct Validity. In order to mitigate construct validity, AlphaTrans is built and validated with well-vetted tools like, GraalVM [41], JaCoCo [51], Python coverage [5], CodeQL [20], etc.

10 Concluding Remarks

In this paper, we introduced AlphaTrans. This neuro-symbolic approach combines the power of static analysis and abilities of LLMs in code synthesis to automate repository-level code translation and validation. AlphaTrans decomposes the program into smaller fragments and translates the fragments in the reverse call order, originally building the project in the target language. In addition to syntax check, AlphaTrans implements two levels of validation through GraalVM and test translation. AlphaTrans is the first approach to translate and validate the entire repository, and we envision several research directions to advance repository-level code translation and validation.

One of the major challenges of repository-level code translation is identifying suitable library APIs in the target PL. Often, equivalent Python APIs may not exist, requiring new code generation or translation of the library API itself. Even if similar libraries exist, the logic of libraries might be different in two PLs. AlphaTrans supports translating frequently used APIs and aims to build a generic pipeline. Supporting all the libraries in the pipeline remains an open challenge we aim to address in future work. Furthermore, while the idea of compositional translation and validation is PL-agnostic, the static analysis makes the extension of AlphaTrans to translating from other source projects challenging. Devising LLM-enabled or PL-agnostic static analysis approaches can benefit code translation approaches such as AlphaTrans. We also showed that the quality of the source project test suite can significantly impact the translation validation results. As part of our future work, we plan to integrate an LLM-based test generator into the AlphaTrans pipeline to advance the validation component.

11 Data Availability

Artifacts and implementation of AlphaTrans are publicly available [32].

References

[1] 2024. Black: Python Code Formatter. https://github.com/psf/black

[2] 2024. PyLint Static Code Analyzer for Python. https://pypi.org/project/pylint/

[3] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2023. AVATAR: A Parallel Corpus for Java-Python Program Translation. In *ACL*. ACL, Toronto, Canada, 2268–2281.

[4] Andrés Bastidas Fuertes, María Pérez, and Jaime Meza Hormaza. 2023. Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry. *Applied Sciences* 13 (2023), 3667.

[5] Ned Batchelder. 2024. Coverage.py. https://pypi.org/project/coverage.

[6] Dante Broggi and Yi Liu. 2023. On the Interoperability of Programming Languages via Translation. In *CSCE*. IEEE, Las Vegas, NV, USA, 2579–2585.

[7] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree Neural Networks for Program Translation. In *NIPS*. Curran Associates Inc., Red Hook, NY, USA, 2552 – 2562.

[8] Peng Di, Jianguo Li, Hang Yu, Wei Jiang, Wenting Cai, Yang Cao, Chaoyu Chen, Dajun Chen, Hongwei Chen, Liang Chen, et al. 2024. CodeFuse-13B: A Pretrained Multi-lingual Code Large Language Model. In *ICSE-SIEP*. ACM, New York, NY, USA, 418–429.

[9] George Dony, Girase Priyanka, Gupta Mahesh, Gupta Prachi, and Sharma Aakanksha. 2010. Programming Language Inter-Conversion. *International Journal of Computer Applications* 1, 20 (2010), 63–69.

[10] Hadeel A. Osman Eman J. Coco and Niemah I. Osman. 2018. JPT : A Simple Java-Python Translator. *CAIJ* 5, 2 (2018), 1–18.

[11] The Apache Software Foundation. 2024. Apache Commons CLI. https://github.com/apache/commons-cli

[12] The Apache Software Foundation. 2024. Apache Commons Codec. https://github.com/apache/commons-codec

[13] The Apache Software Foundation. 2024. Apache Commons CSV. https://github.com/apache/commons-csv

[14] The Apache Software Foundation. 2024. Apache Commons Exec. https://github.com/apache/commons-exec

[15] The Apache Software Foundation. 2024. Apache Commons FileUpload. https://github.com/apache/commons-fileupload

[16] The Apache Software Foundation. 2024. Apache Commons Graph. https://github.com/apache/commons-graph

[17] The Apache Software Foundation. 2024. Apache Commons Pool. https://github.com/apache/commons-pool

[18] The Apache Software Foundation. 2024. Apache Commons Validator. https://github.com/apache/commons-validator

[19] 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* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[20] GitHub. 2024. CodeQL. https://codeql.github.com

[21] GraalVM. 2024. Polyglot API. https://www.graalvm.org/latest/reference-manual/polyglot-programming.

[22] Giovani Guizzo, Jie M. Zhang, Federica Sarro, Christoph Treude, and Mark Harman. 2023. Mutation analysis for evaluating code translation. *Empirical Software Engineering* 29 (2023), 23 pages.

[23] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196

[24] Ali Reza Ibrahimzada. 2024. Program Decomposition and Translation with Static Analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 453–455.

[25] Immunant. 2024. C2Rust Transpiler. https://github.com/immunant/c2rust.

[26] Paul Irwin. 2024. Java to CSharp Converter. https://github.com/paulirwin/JavaToCSharp.

[27] Suman Jain and Inderveer Chana. 2015. Modernization of Legacy Systems: A Generalised Roadmap. In *ICCCT*. ACM, New York, NY, USA, 62–67.

[28] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. 2013. Cloud Migration Research: A Systematic Review. *IEEE Transactions on Cloud Computing* 1 (2013), 142–157.

[29] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Las Vegas, NV, USA, 1529–1541.

[30] Ravi Khadka, Belfrit V Batlajery, Amir M Saeidi, Slinger Jansen, and Jurriaan Hage. 2014. How Do Professionals Perceive Legacy Systems and Software Modernization? In *ICSE*. ACM, New York, NY, USA, 36–47.

[31] Musawwer Khan, Islam Ali, Wasif Nisar, Muhammad Qaiser Saleem, Ali S Ahmed, Haysam E Elamin, Waqar Mehmood, and Muhammad Shafiq. 2022. Modernization Framework to Enhance the Security of Legacy Information Systems. *Intelligent Automation & Soft Computing* 32 (2022), 543–555.

[32] Intelligent CAT Lab. 2025. Artifact Website. https://github.com/Intelligent-CAT-Lab/AlphaTrans.

[33] Kevin Lano and Hanan Siala. 2024. Using model-driven engineering to automate software language translation. *Automated Software Engineering* 31 (2024), 59 pages.

[34] Daniel Lemire. 2024. JavaFastPFOR. https://github.com/lemire/JavaFastPFOR

[35] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[36] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *TACL* 12 (2024), 157–173.

[37] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical Statistical Machine Translation for Language Migration. In *FSE*. ACM, New York, NY, USA, 651–654.

[38] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Migrating Code with Statistical Machine Translation. In *ICSE Companion*. ACM, New York, NY, USA, 544–547.

[39] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-Conquer Approach for Multi-phase Statistical Migration for Source Code. In *ASE*. IEEE, Las Vegas, NV, USA, 585–596.

[40] Vikram Nitin, Rahul Krishna, and Baishakhi Ray. 2024. SpecTra: Enhancing the Code Translation Ability of Language Models by Generating Multi-Modal Specifications. arXiv:2405.18574

[41] Oracle. 2024. GraalVM. https://www.graalvm.org.

[42] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *ICSE*. ACM, New York, NY, USA, 866–866.

[43] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158. https://doi.org/10.1109/TSE.2017.2663435

[44] Mono Project. 2023. Sharpen - Automated Java->C# coversion. https://github.com/mono/sharpen.

[45] pytest dev. 2024. Pytest. https://www.pytest.org.

[46] Lili Qiu. 1999. *Programming Language Translation.* Technical Report. Cornell University, USA.

[47] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *NIPS*. Curran Associates Inc., Red Hook, NY, USA, 20601 – 20611.

[48] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging Automated Unit Tests for Unsupervised Code Translation. arXiv:2110.06773

[49] Fuse Source. 2024. Jansi. https://github.com/fusesource/jansi

[50] Charles Spearman. 1961. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology* 15 (1961), 72 —- 101.

[51] The JaCoCo Team. 2024. Java Code Coverage. https://www.eclemma.org/jacoco/

[52] The JUnit Team. 2024. JUnit. https://junit.org/junit5/

[53] Andrey A Terekhov and Chris Verhoef. 2000. The Realities of Language Conversions. *IEEE Software* 17 (2000), 111–124.

[54] TIOBE. 2023. TIOBE Index. https://www.tiobe.com/tiobe-index.

[55] Sindhu Tipirneni, Ming Zhu, and Chandan K Reddy. 2024. StructCoder: Structure-Aware Transformer for Code Generation. *Transactions on Knowledge Discovery from Data* 18 (2024), 1–20.

[56] Go Transpile. 2024. C to Go Translator. https://github.com/gotranspile/cxgo.

[57] Tree-Sitter. 2024. Tree-Sitter Library. https://tree-sitter.github.io/tree-sitter/

[58] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Onward!* ACM, New York, NY, USA, 187–204.

[59] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A Comprehensive Multilingual Benchmark for Code Translation. In *EMNLP*. ACL, Singapore, 5067–5089.

[60] Aidan ZH Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024. VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-Shot Learners. arXiv:2404.18852

[61] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *FSE* 1 (2024), 1585–1608.

[62] Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code Translation with Corrector via LLMs. arXiv:2407.07472

[63] Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual Code Snippets Training for Program Translation. *AAAI* 36 (2022), 11783–11790.