Can We Translate Code Better with LLMs and Call Graph Analysis?

Yang Luo[1,2,3*]

[1]National Engineering Research Center for Software Engineering, Peking University, Beijing, China
[2]School of Software and Microelectronics, Peking University, Beijing, China
[3]PKU-OCTA Laboratory for Blockchain and Privacy Computing, Peking University, Beijing, China
luoyang@pku.edu.cn

Abstract

This paper proposes an innovative code translation method aimed at addressing the accuracy issues encountered by large language models (LLMs) in translating code of complex large-scale software projects. The method utilizes the Language Server Protocol to obtain the call graph of the entire codebase, and optimizes the input prompt of the LLM accordingly, significantly improving the correctness of translation at the compilation stage. Moreover, this method introduces the bridged debuggers technique based on the Debug Adapter Protocol and dynamic test case generation, effectively fixing runtime errors. Experiments on multiple mainstream datasets demonstrate that, compared to existing code translation methods and LLMs, this method achieves a significant improvement in translation accuracy.

1 Introduction

With the increasing diversity of software development environments, code translation technology plays a crucial role in supporting language upgrades, cross-platform development, and modernization of legacy systems. Efficiently and accurately migrating code from one language to another is essential for developers. Traditional rule-based translators, while ensuring consistency and accuracy of code translation with a high degree of precision and predictability, rely on exhaustive rule settings and struggle to cover all special cases in programming practices, especially in complex, dynamic programming environments and highly modularized projects.

Meanwhile, machine learning-based approaches, particularly large language models (LLMs) such as ChatGPT [OpenAI, 2022], have shown strong potential in the field of code translation. These methods can understand and convert complex structures and semantics between multiple programming languages, improving the flexibility and accuracy of translation. However, despite the significant effectiveness of LLM-based methods in translating code with simple contexts, they still face challenges in coherence and error handling when dealing with multi-file and multi-module software projects.

*Corresponding author

To address these issues, this paper proposes TransGraph, an innovative code translation approach based on call graphs [Graham *et al.*, 1982] and bridged debuggers. By utilizing the Language Server Protocol [Microsoft, 2016] to obtain the call graph of the entire codebase, TransGraph can comprehensively describe the function call relationships in the program, thereby optimizing the input prompts for LLMs and significantly improving the correctness of translation at the compilation stage. Furthermore, TransGraph introduces the bridged debuggers technique based on the Debug Adapter Protocol [Microsoft, 2021] and dynamic test case generation to effectively fix runtime errors. Experiments on multiple mainstream datasets, including EvalPlus and Avatar show that compared to the baseline method, TransGraph significantly improves the translation success rate, with an average increase of 15.7%.

To the best of our knowledge, we are the first to (1) leverage the call graph structure to explicitly handle contextual information, reducing compilation errors in LLM-translated code; (2) introduce bridged debuggers to synchronize context in source and target code execution, effectively fixing runtime errors; (3) propose a binary search algorithm combined with dynamic testing to greatly improve the efficiency of bug localization in complex code translation.

Our main contributions include:

- Proposing TransGraph, a code translation method based on call graphs and bridged debuggers, effectively addressing the coherence and accuracy issues of LLM-translated code in complex software projects.
- Designing a recursive algorithm based on binary search, combined with dynamic execution comparison of source and target code in bridged debuggers, significantly improving the efficiency of runtime error localization and fixing in translated code.
- Conducting extensive experiments on multiple commonly used datasets, confirming that the TransGraph method significantly improves the success rate of code translation compared to the existing baseline method, with an average increase of 15.7%.

The rest of this paper is organized as follows. Section 2 reviews related work in the field of code translation. Section 3 elaborates on the TransGraph method in detail. Section 4 presents the experimental results of TransGraph on multiple datasets. Section 5 discusses the limitations. Section 6 summarizes the main work of this paper and provides an outlook on future research directions.

2 Related Work

In the field of code translation, rule-based translators have been widely applied, such as the Python-to-Java converter proposed by Melhase et al. [py2, 2024], which ensures accurate conversion between codes through a fixed set of rules. Although these tools provide highly precise and consistent translations, such as Java2CSharp [Jav, 2023] and Sharpen [Sha, 2023], they often lack the flexibility to handle complex and dynamic code structures, frequently requiring manual intervention to address semantic and structural differences during the translation process.

Regarding machine learning-based methods, significant progress has been witnessed in recent years. Statistical machine learning compilers, such as the models proposed by Nguyen et al. [Nguyen *et al.*, 2013] and Karaivanov et al. [Karaivanov *et al.*, 2014], leverage large parallel corpora to optimize code translation by identifying patterns and correlations between programming languages. Additionally, Aggarwal et al. [Aggarwal *et al.*, 2015] applied sentence alignment techniques in the conversion between Python versions. In exploring bidirectional compilers, Schultes [Schultes, 2021] proposed an innovative compilation technique for translation between Swift and Kotlin, while Ling et al.'s CRustS compiler [Ling *et al.*, 2022] emphasizes reducing unsafe expressions during code translation.

Transformers and other ML-based tools such as CodeBERT [Feng *et al.*, 2020] and CodeGPT [Lu *et al.*, 2021] utilize advanced encoder-decoder architectures, first introduced in the pioneering work by Vaswani et al. [Vaswani *et al.*, 2017]. These tools have achieved remarkable results in the field of code translation, generating semantically coherent and accurate code. Wang et al.'s CodeT5 [Wang *et al.*, 2021] and Ahmad et al.'s PLBART model [Ahmad *et al.*, 2021a] further incorporate code semantics, demonstrating versatility across programming languages and natural languages. Roziere et al. [Roziere *et al.*, 2020] introduced TransCoder, a neural transcompiler based on unsupervised machine translation, which can perform function-level translation between C++, Java, and Python without the need for parallel corpora, using only monolingual code. However, its limitation lies in not considering the conventions and readability of the target language. Mariano et al. [Mariano *et al.*, 2022] designed NGST2, an automated translation method from imperative to functional code based on neuron-guided program synthesis, leveraging the trace compatibility assumption between source and target programs. Nevertheless, this method has only been evaluated on specific APIs in Java and Python, and its generalizability requires further validation.

In the latest research on leveraging LLMs for code translation, Roziere et al. [Roziere *et al.*, 2021] introduced an unsupervised code translation technique that combines self-training and automated unit testing. This technique utilizes unit tests to create synthetic parallel datasets for model enhancement, although these tests are not incorporated into the training loss calculation. Wang et al.'s study [Wang *et al.*, 2022] combines reinforcement learning with compiler feedback, providing a different perspective from traditional supervised learning. Furthermore, the research by Haugeland et al. [Haugeland *et al.*, 2021] and Orlanski et al. [Orlanski *et al.*, 2023] explores the impact of LLMs on code translation, particularly in low-resource language environments, highlighting the potential and challenges in this evolving field. Ramos et al. [Ramos *et al.*, 2024] proposed the BatFix method, which combines program repair and synthesis to rectify target code generated by language models, but it relies on test cases and has limited support for complex projects. Nijkamp et al. [Nijkamp *et al.*, 2022] released the CodeGen model series, achieving state-of-the-art performance in Python code generation on the HumanEval dataset, although it primarily focuses on code generation rather than translation tasks. Li et al. [Li *et al.*, 2023] open-sourced the StarCoder model, achieving the best performance among open-source code LLMs supporting multiple languages. Yang et al. [Yang *et al.*, 2024] proposed the UniTrans framework, significantly improving the accuracy of code translation between Python, Java, and C++ using LLMs such as GPT-3.5 and LLaMA by introducing test case-based correction and iterative repair strategies, but the automatic generation of test cases requires further optimization.

As evident from the above, existing code translation methods still exhibit notable limitations. First, a significant portion of the approaches rely on a large number of manually written, comprehensively covering test cases to validate the translated code, which is difficult to achieve in many software projects. Second, some methods have poor support for complex software code translation, particularly in large-scale applications involving multi-layer dependencies and multi-component interactions. In reality, software is often quite complex. Additionally, most of these techniques do not support incremental translation, meaning that encountering errors during the translation process leads to the failure of the entire translation task, rather than interrupting only the erroneous part while allowing other parts to be translated unaffected. This greatly limits the scope of code translation in practical development processes. In contrast, our proposed TransGraph method significantly improves the understanding of complex code structures by integrating call graphs and bridged debuggers, effectively reducing the reliance on external test cases. Moreover, TransGraph performs code translation at the function level, localizing the impact of errors to the specific function without affecting the entire software, thereby enhancing the robustness and flexibility of code translation.

3 Methodology

The architecture of the TransGraph approach is shown in Figure 1. The source code is analyzed through the Language Server Protocol (LSP) to generate a call graph, which is then provided to the TransGraph controller. The controller generates input prompts for the LLM based on the call graph to guide the generation of target code. The generated source binary is inspected by the source debugger, while the generated target code is inspected by the target debugger on the target

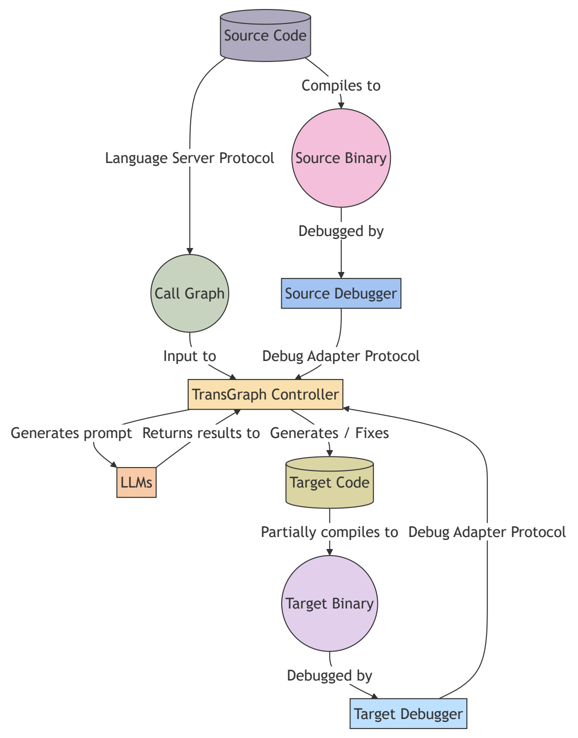


Figure 1: The TransGraph approach.

binary. The two debuggers are connected to the controller via the Debug Adapter Protocol (DAP), forming bridged debuggers. This setup allows for rapid correction of the target code when deviations are detected at runtime.

3.1 Call Graph Based Translation

Due to the coherence issues inherent in LLMs, when using LLMs to translate complex, multi-file code, inconsistencies between the preceding and following code often arise. To address this problem, we first obtain the call graph of the entire codebase through the Language Server Protocol, which includes all involved classes, functions, global variables, and their relationships. Specifically, the call graph is defined as follows: a directed graph $G = (V, E)$, where V represents the functions and methods in the program, and $E \subseteq V \times V$ represents the calling relationships, i.e., if function f calls function g, there exists a directed edge (f, g).

To generate the call graph, we utilize the standard interfaces provided by LSP to obtain language-specific information through static code analysis, such as code completion, go to definition, find all references, etc. Most mainstream programming languages have mature LSP implementations maintained by official organizations or communities, such as clangd for C/C++, jdt.ls for Java, Pyright for Python, and so on.

After obtaining the call graph, we generate specific LLM input prompts for each node (function). The prompt includes the code of the function, other functions it calls, referenced global variables (including class member variables), caller code, and other contextual information. To prevent prompts from becoming too long, we set a maximum length threshold (e.g., 32k tokens), and when this threshold is exceeded, low-priority contexts (such as comments) are truncated until the length limit is met. Through these carefully designed prompts, LLMs can better understand the structured view of the code, effectively handle large and complex codebases, significantly reduce compilation errors, and improve the correctness of code translation.

A key goal of code translation is successful compilation, i.e., achieving zero compilation errors. For medium to large software projects, achieving this goal through LLM-based code translation is quite challenging. The main challenges include:

Instability of LLM output. The results generated by LLMs can be unstable, and it cannot be guaranteed that a specific compilation error will be resolved after multiple attempts. Moreover, while resolving one compilation error, new compilation errors may be introduced in other locations.

Insufficient context understanding. LLMs often struggle to fully understand the context of the code, especially when the code involves cross-module calls. Even if the code within each module is error-free when isolated, combining them may lead to errors due to mismatched parameter counts or types.

Serial efficiency issues in handling compilation errors. Most programming languages stop the compilation process when encountering a compilation error. LLMs attempt to correct the code based on the error information and then recompile, but until the current compilation error is resolved, the location of the next error cannot be determined, hindering the possibility of using techniques like multi-threading to handle multiple compilation errors in parallel.

To address the above challenges, we propose a call graph-based code translation method, as shown in Algorithm 1. In the directed graph of the call graph, we traverse from nodes with smaller degrees to larger degrees and apply LLM techniques to translate each node (function). The translated function code, along with the functions it calls, input/output variables, and global variables, are placed into a main function to form a new codebase and attempt compilation. All variables are declared and initialized to null values. Since each translated function forms an independent codebase, there are no dependencies between them, making it easy to achieve parallel processing using multi-threading/multi-processing techniques. We have verified the feasibility of this strategy in practice, and compared to single-threaded processing, the translation speed can be increased several fold (depending on the available CPU cores).

This approach limits compilation errors to the scope of functions. Since the context within a function is usually simpler than the context of cross-module calls, the probability of LLM translation errors is relatively lower. Furthermore, LLM translations at the function level have no dependencies, allowing for parallel processing. In implementation, we allocate an independent process for each function's translation. Through this method, even if 100% successful compilation cannot be achieved for the entire codebase, we can still achieve zero compilation errors in a considerable portion of local code functions. The remaining compilation errors can be repaired

Algorithm 1 Code Translation

```
Input: Source code
Output: Translated target code
procedure TRANSGRAPH($sc$)
    $cg \leftarrow LSPAnalyze(sc)$
    $sb \leftarrow Compile(sc)$
    $sd \leftarrow AttachDebugger(sb)$
    for $f$ in $GetFunctions(cg)$ do
        $p \leftarrow GeneratePrompt(cg, f)$
        $tc[f] \leftarrow LLMTranslate(p)$
        $tb \leftarrow CompileWithMain(tc[f])$
        $td \leftarrow AttachDebugger(tb)$
        while ExistsVariableDiff(sd, td) do
            $ei \leftarrow IdentifyErrors(sd, td)$
            $p \leftarrow GeneratePrompt(ei)$
            $tc[f] \leftarrow LLMTranslate(p)$
            $tb \leftarrow CompileWithMain(tc[f])$
            $td \leftarrow ReAttachDebugger(tb)$
        end while
    end for
    return $tc$
end procedure
```

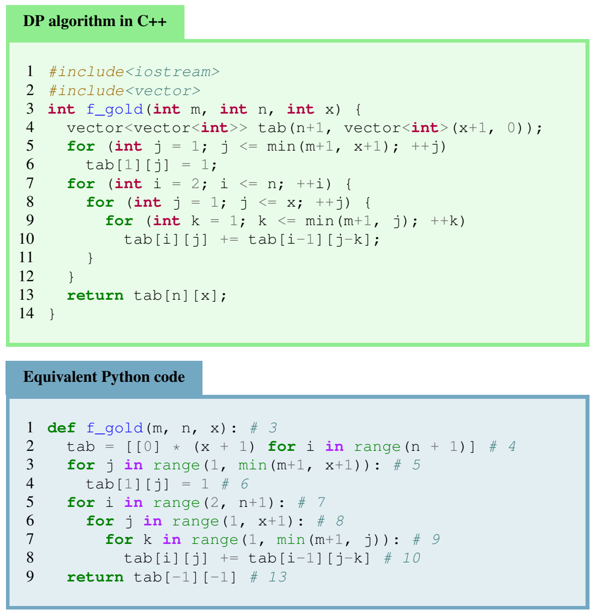
using manual or other methods. This is valuable for translating medium to large software projects.

3.2 Run-time Error Correction via Bridged Debuggers

Errors that occur during the runtime phase of code are often more difficult to detect and fix than compilation errors. In engineering practice, regression testing is commonly used to detect runtime errors. However, many projects lack comprehensive test suites, making it difficult to cover all code paths, especially some complex deep paths.

We propose a method that combines bridged debuggers and dynamic test case generation to address runtime errors. This method does not rely on existing test cases and only requires that the source code and target code can be compiled, run, and debugged. Almost all mainstream languages have corresponding debuggers, and source code is usually debuggable. In this step, we select target functions that compiled successfully in the previous step. It is worth mentioning that TransGraph is based on dynamic execution rather than static analysis to correct runtime errors, enabling it to obtain the actual types and values of variables at runtime without the need for cumbersome static type inference in advance.

Runtime errors often arise when the execution results of certain specific statements in the target code are inconsistent with the source code, such as different variable values, leading to overall program deviations. Therefore, locating the specific statements causing the deviations is key to fixing such errors. An intuitive approach is to run the source code and target code simultaneously in debug mode, perform single-step debugging, and monitor variable states in real-time. Once inconsistent variable values are found at a certain line of code, it can be determined that a runtime error exists at that location. The following are three main challenges faced in this process:



DP algorithm in C++

```
1  #include<iostream>
2  #include<vector>
3  int f_gold(int m, int n, int x) {
4    vector<vector<int>> tab(n+1, vector<int>(x+1, 0));
5    for (int j = 1; j <= min(m+1, x+1); ++j)
6      tab[1][j] = 1;
7    for (int i = 2; i <= n; ++i) {
8      for (int j = 1; j <= x; ++j) {
9        for (int k = 1; k <= min(m+1, j); ++k)
10         tab[i][j] += tab[i-1][j-k];
11     }
12   }
13   return tab[n][x];
14 }
```

Equivalent Python code

```
def f_gold(m, n, x): # 3
  tab = [[0] * (x + 1) for i in range(n + 1)] # 4
  for j in range(1, min(m+1, x+1)): # 5
    tab[1][j] = 1 # 6
  for i in range(2, n+1): # 7
    for j in range(1, x+1): # 8
      for k in range(1, min(m+1, j)): # 9
        tab[i][j] += tab[i-1][j-k] # 10
  return tab[-1][-1] # 13
```

Figure 2: Code translation from C++ to Python for the DP algorithm.

The target code may not be able to run due to compilation errors. To address this issue, we select compilable functions from the call graph, starting from nodes with an in-degree of zero, and extract the function and all its related code (including other functions it calls and global variables used) to form an independent program. Subsequently, we execute its main function in debug mode. All input variables of the function (including global variables) are initialized to zero values.

How to ensure consistent variable contexts between the target code and source code. We employ a context synchronization technique based on bridged debuggers. Since the target code only contains the execution environment of the target function, the initial values of all input and output variables may differ from the actual values in the source code. To address this issue, during the single-step debugging process of bridged debuggers, we synchronize the values of input variables (including parameters and global variables) from the source code debugger to the variables with the same names in the target code debugger, ensuring that the variable names, scopes, types, and values are completely consistent between the two before function execution.

Running the entire code using single-step debugging has low efficiency. In compute-intensive functions, the time overhead of single-step debugging is particularly evident. To improve performance, we adopt an algorithm similar to binary search, as shown in Algorithm 2. The core idea is to reduce the number of single-step executions by setting breakpoints at the ”midpoint” of the source code and the corresponding location in the target code. The ”midpoint” here refers to the median value of all possible breakpoint locations in the source code. If the target code has comments that correspond one-to-one with the line numbers of the source code, the ”midpoint” can be determined accordingly; otherwise, the

Algorithm 2 Recursive Binary Search Debugging

```
1: Input: $sDebugger, tDebugger, s, e, ctx$
2: Output: Discrepancy location
3: procedure RBSD($sDebugger, tDebugger, s, e, ctx$)
4:     $m \leftarrow (s+e)/2$
5:     $RestoreContext(ctx)$
6:     $ExecuteTo(m)$
7:     if $ExistsDiscrepancy(sDebugger, tDebugger)$ then
8:         if $s == e$ then
9:             return $m$
10:        end if
11:        return RBSD($sDebugger, tDebugger, s, m, ctx$)
12:    else
13:        $SaveContext(m)$
14:        return RBSD($sDebugger, tDebugger, m+1, e, ctx$)
15:    end if
16: end procedure
```

middle line of the target code can be heuristically selected as the "midpoint." When a breakpoint is hit, the values of key variables on both sides are compared. If the values on both sides are equal at this point, the problem may occur in the latter half; if the values are unequal, the problem may occur in the former half. By recursively applying the "binary search" strategy, the problem can eventually be localized to a more precise code interval. Compared to simple single-step execution, the time complexity of this method is reduced from $O(n)$ to $O(logn)$, where n is the number of lines of code. Additionally, as shown in Figure 2, by constructing specific LLM prompts during code translation and generating comments with line number mappings and variable name mappings, we solve the problem of line number correspondence between the source code and target code. Through these comments, we can achieve the mapping of code breakpoints and variable values.

It should be noted that we only assume that the LLM maintains a roughly similar function body structure between the source code and target code, allowing it to rename local variables, adjust their order, or inline/extract within a small scope during translation, without imposing strict requirements. LLMs usually handle these compiler optimization-level equivalent transformations well. If line numbers no longer match due to excessive optimization, our method will gracefully fall back to simple single-step execution.

Once a specific error is located, the source code, target code, error line number, error variables, and other contextual information of the current function can be organized into a new prompt for subsequent target code repair. Figure 2 shows an example of code translation and debugging from C++ to Python. For the case where the execution result of line 8 in Python is inconsistent with line 10 in C++, we generate the following repair prompt:

```
In the translation from C++ to Python, the
↪  result of line 8 in Python (corresponding
↪  to line 10 in C++):
tab[i][j] += tab[i-1][j-k]
is inconsistent with the C++ code.
The relevant variables have the following
↪  values: i = 3, j = 5, k = 2, m = 4, n =
↪  6, x = 8
Please correct this line of Python code.
```

After replacing the existing target function code and recompiling, we repeat the previous debugging iteration process until all runtime errors are eliminated. Throughout the entire debugging process, snapshots of variables at all breakpoints are recorded and used to quickly restore the execution context during binary search, avoiding repeated execution.

4 Evaluation

4.1 Experiment Setup

The experiments in this study covered three programming languages: C/C++, Java, and Python. The compilation environments included gcc, OpenJDK, and CPython, and the debugging tools included gdb, jdb, and pdb. TransGraph integrates with VSCode using the Language Server Protocol to extract call graphs for these programming languages. In addition, it controls debuggers for various languages through the Debug Adapter Protocol. The LLMs used in this paper include OpenAI's GPT-3.5 (ChatGPT) [OpenAI, 2022], GPT-4 [Achiam *et al.*, 2023], Llama 2 70B [Touvron *et al.*, 2023], StarCoder [Li *et al.*, 2023], and Claude 3 Sonnet [Anthropic, 2024]. The datasets used in this paper include: CodeNet [Puri *et al.*, 2021], Avatar [Ahmad *et al.*, 2021b], EvalPlus [Liu *et al.*, 2024], Apache Commons CLI [apa, 2023] (Java), Click [pyt, 2023] (Python), and HumanEval-X [Zheng *et al.*, 2023].

4.2 Influence of Different LLMs and Languages

We evaluated the code translation effectiveness of four mainstream LLMs on the CodeNet dataset using the TransGraph method, covering GPT-3.5, GPT-4, StarCoder, and Claude 3 Sonnet, with testing scenarios including C to Java, Java to Python, and Python to C language conversions. Thirty experiments were conducted for each translation, and the results are shown in Figure 3. In this paper, we define successful translation as: 1) the translated code can be successfully compiled, 2) the translated code does not produce exceptions or crashes at runtime, and 3) the translated code is functionally equivalent to the original code, i.e., it produces the same output for the same input. The LLM prompts used in this experiment are as follows:

```
Translate the following code from
↪  {source_language} to {target_language}.
↪  The translated code should be correct,
↪  efficient, and idiomatic.
{source_code}
```

Our TransGraph method conducted experimental comparisons using these four LLM models. GPT-4, as an industry-leading LLM, significantly outperformed GPT-3.5, reflecting GPT-4's powerful semantic understanding and code generation capabilities. Moreover, LLMs specifically designed for code-related tasks, such as StarCoder, did not significantly outperform general-purpose LLMs like GPT in the field of code translation. Additionally, the translation success rates in various language conversion scenarios also showed significant differences. For example, the translation success rate from Java to Python was relatively high, which may be because Python, as a dynamic language, lacks a strict compila-

Dataset	TransCoder	NGST2	BatFix	CodeGen	StarCoder	GPT-4	Llama 2	UniTrans	TransGraph (Ours)	
									GPT-3.5	GPT-4
CodeNet	29.8	32.4	49.9	19.5	36.7	61.4	12.8	65.7	71.4	**76.2**
Avatar	37.2	39.9	55.3	6.2	12.9	63.3	3.1	67.6	68.9	**72.5**
EvalPlus	25.1	28.4	27.8	14.6	18.0	73.7	1.9	77.8	75.3	**79.1**
Commons CLI	9.7	13.5	15.3	3.7	5.2	15.4	3.0	19.1	12.5	**17.8**
Click	5.5	8.1	13.9	2.1	3.6	11.2	1.4	14.9	18.8	**22.4**
HumanEval-X	23.4	26.9	28.4	9.5	11.8	35.2	3.3	38.6	45.4	**52.6**

Table 1: Successful translation (in %) of various code translation approaches

Dataset	Vanilla	w/o BD	w/o Referenced Vars	w/o Called Funcs	w/o Caller Code	TransGraph
Avatar	55.3	66.5	62.1	64.3	67.2	68.9
EvalPlus	58.8	73.7	68.4	71.2	74.1	75.3
HumanEval-X	28.4	42.9	37.3	40.8	43.6	45.4
Average	47.5	61.0	56.0	58.8	61.6	63.2

Table 2: Ablation study results. The number is the successful translation Rate (%) on each dataset

tion stage, thereby reducing the occurrence of compilation errors. In contrast, the success rate of conversion from Python to C/C++ was relatively low, mainly because Python's dynamic features reduce the accuracy of call graph generation, and due to the lack of type information, the process of translating to C/C++ requires correctly inferring variable types, which greatly increases the complexity of code translation and significantly increases the likelihood of translation errors. We defined the successful translation metric: successfully translated code needs to pass compilation, runtime checks, and functional correctness tests, ensuring that the translated code has the same functionality as the original code.

4.3 Translation Error Analysis

We conducted code translation tests on three datasets, CodeNet, EvalPlus, and HumanEval-X, for some traditional code translation methods (TransCoder [Roziere *et al.*, 2020], NGST2 [Mariano *et al.*, 2022]) and LLM-based methods (e.g., BatFix [Ramos *et al.*, 2024], CodeGen [Nijkamp *et al.*, 2022], StarCoder [Li *et al.*, 2023], GPT-4 [Achiam *et al.*, 2023], Llama 2 [Touvron *et al.*, 2023], UniTrans [Yang *et al.*, 2024]), and categorized the error causes. The results are shown in Figure 4. In this experiment, except for Llama 2 which itself is an LLM, other methods used GPT-4 if an LLM was needed to exclude the influence of LLM performance differences. Specifically, we further divided compilation errors into: syntax errors, type errors, symbol resolution errors, etc.; runtime errors were divided into: null pointer exceptions, array out of bounds, division by zero errors, etc.; functional errors were divided into: inconsistent output, inconsistent control flow, resource leaks, etc. It can be seen from the figure that compilation errors are the main type of translation errors, usually accounting for more than 40%. Compared with other methods, TransGraph has a lower proportion of compilation errors.

4.4 Performance Analysis

We evaluated the performance of various code translation methods on different datasets, and the results are shown in Table 1. TransCoder and NGST2, as some traditional code translation methods, already have a quite good performance, but there is still a certain gap compared with LLM-based methods. In contrast, LLM-based methods such as BatFix, CodeGen, StarCoder, GPT-4, Llama 2, UniTrans, and TransGraph showed higher success rates. We evaluated the performance differences of TransGraph when using two different LLMs, GPT-3.5 and GPT-4. The results showed that the translation success rate of TransGraph was further improved when using GPT-4, which is partly attributed to the more powerful processing capabilities of the GPT-4 model itself. Nevertheless, TransGraph still outperformed other baseline methods on most datasets even when using GPT-3.5. This confirms the effectiveness of our proposed optimization strategies based on call graphs and bridged debuggers.

4.5 Ablation Study

To verify the effectiveness of each key design in TransGraph, we conducted ablation experiments. Specifically, on the complete TransGraph system based on GPT-3.5, we respectively removed the bridged debugger function and call graph information (including referenced variable information, called function information, and caller code information), resulting in four variants: w/o BD, w/o Referenced Vars, w/o Called Funcs, and w/o Caller Code. In addition, we also implemented an original code translation system based solely on GPT-3.5 (Vanilla) as a baseline. Table 2 shows the successful translation rates of each system on three datasets. It can be seen that introducing referenced variable information improves translation accuracy by an average of 7.3%, introducing called function information improves translation accuracy by an average of 4.4%, while the improvement from introducing caller code information is relatively small. This indicates that the information within functions, especially the referenced variables, is more critical for code understanding and correct translation. Further adding bridged debuggers improves the accuracy by another 2.2%, and the complete TransGraph system achieves an average accuracy of 15.7% higher than the original GPT-3.5, confirming the effectiveness of our

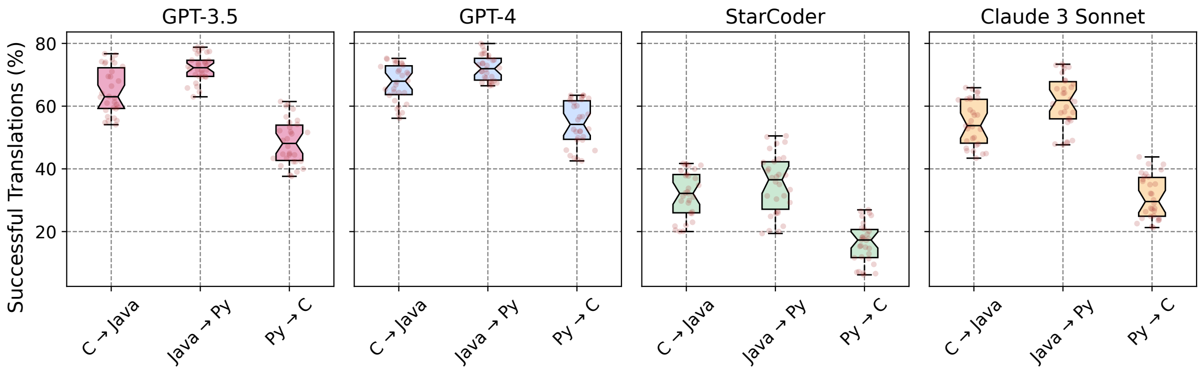


Figure 3: Translation effectiveness of different LLMs and programming languages for CodeNet dataset.

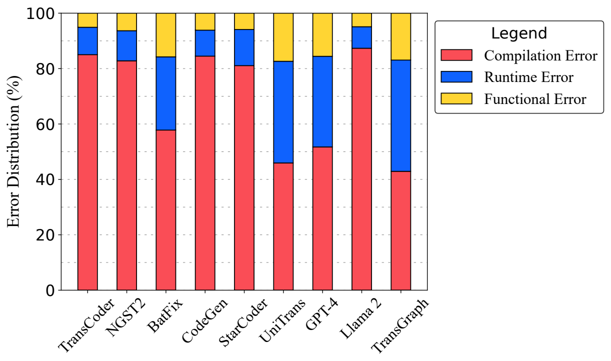


Figure 4: Outcome of code translations averaged over CodeNet, EvalPlus and HumanEval-X datasets.

proposed method.

Analyzing the roles of the two key mechanisms in TransGraph, the impact of referenced variables is the largest, followed by called functions, while the role of caller code is relatively small. This indicates that the information within functions is more critical for code understanding and correct translation. The bridged debugger mainly reduces runtime errors, especially those exceptional situations not covered by test cases. Traditional code translation methods have difficulty handling such boundary cases, while the bridged debugger greatly improves the efficiency of fixing runtime bugs by synchronizing execution contexts between the source code and target code and quickly locating difference points using binary search.

5 Discussion

Despite the encouraging results achieved by TransGraph on code translation tasks, it still has some limitations. First, TransGraph has currently only been experimentally evaluated between mainstream languages such as C++, Java, and Python, and its support for some emerging or niche programming languages remains to be further verified. This limitation may be due to the significant differences in characteristics between different programming languages, and many niche languages lack mature LSP protocol implementations, which brings difficulties to call graph extraction.

Secondly, TransGraph relies on the LSP protocol to extract call graph information during the translation process, but for some dynamic languages (such as JavaScript), their dynamic features may lead to inaccurate call graphs extracted by static analysis, which in turn affects translation performance. To address this problem, one possible solution is to combine dynamic analysis techniques to capture actual function calls occurring at runtime and dynamically update the call graph. Although this method may introduce some runtime overhead, it is expected to improve translation accuracy.

Moreover, despite the introduction of function-granularity incremental code translation, the time overhead of translation for ultra-large-scale (e.g., millions of lines) engineering projects cannot be ignored. The current TransGraph implementation is primarily based on a single-machine multi-core CPU, but in scenarios with sufficient computing power, using GPUs or even multi-machine distributed environments can significantly improve the scale and efficiency of parallel translation. This requires introducing customized scheduling strategies in the compilation and debugging stages to balance the load and reduce communication overhead.

6 Conclusion

In this paper, we propose TransGraph, a code translation method that combines call graphs and bridged debuggers. To overcome the accuracy limitations of LLMs in code translation, TransGraph systematically handles dependencies between variables and functions by constructing call graphs, significantly reducing compilation errors in the target code. In addition, this method effectively detects and fixes runtime errors by leveraging bridged debuggers. The experimental results show that TransGraph performs excellently in reducing compilation and runtime errors, especially when dealing with complex multi-file code projects. Tests conducted on multiple datasets, including EvalPlus and Avatar, indicate that compared to the existing baseline method, TransGraph significantly improves translation accuracy, with an average increase of 15.7%. Future work directions include expanding language coverage, improving call graph extraction for dynamic languages, and exploring more efficient incremental translation methods.

Acknowledgments

This work was supported by the National Key R&D Program of China under Grant No. 2022YFB2703301.

Ethical Impact

As a code translation technique, the ethical impact of TransGraph is relatively limited. Unlike natural language processing tasks, code translation mainly targets the developer community, and the translated content usually does not involve sensitive topics. Despite this, we still need to be vigilant about the potential misuse of this technology for unauthorized code copying or piracy. To mitigate such risks, we suggest adding appropriate license information to the code generated by TransGraph to remind users to comply with relevant intellectual property laws and regulations. In the long run, code translation is expected to improve the interoperability between programming languages, making it more convenient for developers with different language backgrounds to communicate and collaborate, and lowering the threshold for learning new languages, thus promoting the inclusive development of the software industry.

References

[Achiam *et al.*, 2023] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[Aggarwal *et al.*, 2015] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. Using machine translation for converting python 2 to python 3 code. Technical report, PeerJ PrePrints, 2015.

[Ahmad *et al.*, 2021a] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[Ahmad *et al.*, 2021b] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*, 2021.

[Anthropic, 2024] Anthropic. Claude 3 sonnet. https://www.anthropic.com/news/claude-3-family, 2024.

[apa, 2023] Apache commons cli. https://commons.apache.org/proper/commons-cli/, 2023.

[Feng *et al.*, 2020] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[Graham *et al.*, 1982] Susan L Graham, Peter B Kessler, and Marshall K McKusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[Haugeland *et al.*, 2021] Sindre Grønstøl Haugeland, Phu H Nguyen, Hui Song, and Franck Chauvel. Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 170–177. IEEE, 2021.

[Jav, 2023] Java 2 csharp translator for eclipse. https://sourceforge.net/projects/j2cstranslator/, 2023.

[Karaivanov *et al.*, 2014] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM international symposium on new ideas, new paradigms, and reflections on programming & software*, pages 173–184, 2014.

[Li *et al.*, 2023] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[Ling *et al.*, 2022] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R Cordy, and Ahmed E Hassan. In rust we trust: a transpiler from unsafe c to safer rust. In *Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings*, pages 354–355, 2022.

[Liu *et al.*, 2024] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[Lu *et al.*, 2021] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[Mariano *et al.*, 2022] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. Automated transpilation of imperative to functional code using neural-guided program synthesis. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.

[Microsoft, 2016] Microsoft. Language server protocol. https://microsoft.github.io/language-server-protocol/, 2016.

[Microsoft, 2021] Microsoft. Debug adapter protocol. https://microsoft.github.io/debug-adapter-protocol/, 2021.

[Nguyen *et al.*, 2013] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 651–654, 2013.

[Nijkamp *et al.*, 2022] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[OpenAI, 2022] OpenAI. Chatgpt. https://www.openai.com/chatgpt, 2022.

[Orlanski *et al.*, 2023] Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. Measuring the impact of programming language distribution. In *International Conference on Machine Learning*, pages 26619–26645. PMLR, 2023.

[Puri *et al.*, 2021] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

[py2, 2024] py2java: Python to java language translator. https://pypi.org/project/py2java/, 2024.

[pyt, 2023] Click. https://click.palletsprojects.com/en/8.1.x/, 2023.

[Ramos *et al.*, 2024] Daniel Ramos, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. Batfix: Repairing language model-based transpilation. *ACM Transactions on Software Engineering and Methodology*, 2024.

[Roziere *et al.*, 2020] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33:20601–20611, 2020.

[Roziere *et al.*, 2021] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021.

[Schultes, 2021] Dominik Schultes. Sequalsk—a bidirectional swift-kotlin-transpiler. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 73–83. IEEE, 2021.

[Sha, 2023] Automated java-¿csharp coversion. https://github.com/mono/sharpen, 2023.

[Touvron *et al.*, 2023] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[Wang *et al.*, 2021] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[Wang *et al.*, 2022] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. *arXiv preprint arXiv:2203.05132*, 2022.

[Yang *et al.*, 2024] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1585–1608, 2024.

[Zheng *et al.*, 2023] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, 2023.