|  |  |
| --- | --- |
| **Student ID** | **Student Name** |
| **2242187** | **Abdulsalam Ahyaf** |
| **2240297** | **Abdulmajeed Abdullah** |
| **2240198** | **Faisal Alghamdi** |
| **2241316** | **Faisal Alqarni** |
| **2240371** | **Feras Ahmad** |

**PAPER 1:  
Exploring and Unleashing the Power of Large Language Models in Automated Code Translation**

**Abstract**  
This research investigates the possibilities of Large Language Models (LLMs), including LLaMA and GPT-3.5, for automated code translation. Because they depend on substantial training materials and rule-based techniques, traditional code translation tools, or transpilers, have limitations with regard to correctness and practical deployment. The study examines how well LLMs succeed in getting over these restrictions and suggests a brand-new framework named UniTrans to improve the caliber of code translation. [1]

**Method**

**UniTrans Framework**

This paper describes a unified code translation framework called UniTrans, which is designed to enhance the performance of Large Language Models (LLMs) in automated code translation. Here is a detailed explanation of the method used in the paper:[1]

1.Test Case Generation Phase

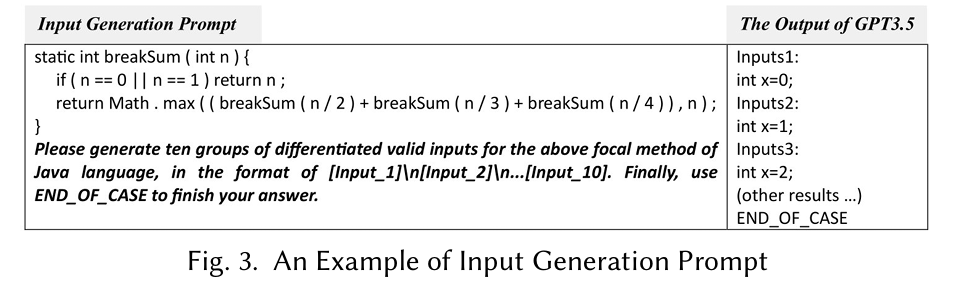
Objective: Generate test cases for the target programs to guide the translation process.

Process:

* + Input Generation Prompt: The LLM is prompted to generate candidate inputs for the source programs.
  + Execution of Source Programs: The valid inputs are selected and executed to obtain corresponding outputs.
  + Formulating Test Cases: The generated inputs and outputs are converted to test cases that fit the target programming languages (PLs). For statically typed PLs, input and output types are explicitly defined.

**Input Generation Prompt:**

"${progsrc} \nPlease generate ten groups of differentiated valid inputs for the above focal method of ${plsrc} language, in the format of [Input\_1]\\n[Input\_2]\\n ... [Input\_10]. Finally, use END\_OF\_CASE to finish your answer."

**Explanation:** This prompt asks the LLM to generate ten valid inputs for the given source program in the specified programming language.

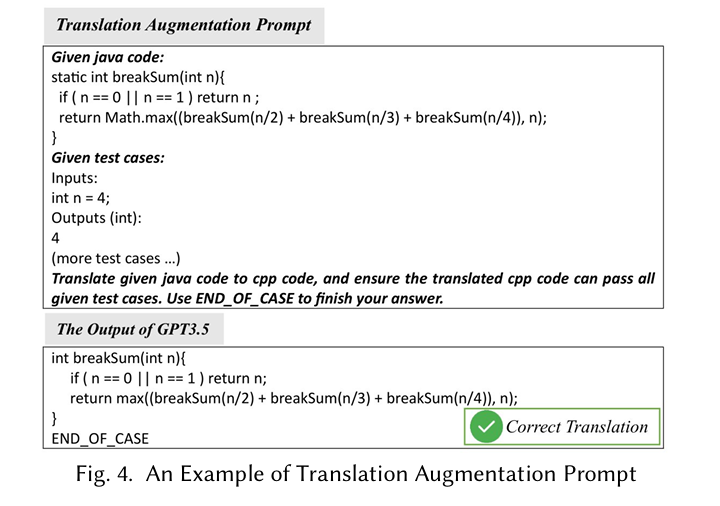
**2.Translation Augmentation Phase**

**Objective:** Use the test cases generated to enhance the accuracy and quality of the code translation.

**Process:**

* **Translation Augmentation Prompt:** The LLM is prompted with the source code and test cases to translate the source code to the target PL, ensuring the translated code passes all given test cases.
* **Execution and Preliminary Inspection:** The translated programs are executed with the test cases to check their correctness. If they pass, the translation is deemed correct; otherwise, they proceed to the next phase.

**Translation Augmentation Prompt:**

"Given ${plsrc} code:\n${progsrc}\nGiven test cases:${TCtar}\nTranslate given ${plsrc} code to ${pltar} code, and ensure the translated ${pltar} code can pass all given test cases. Use END\_OF\_CASE to finish your answer."

**Explanation:** This prompt provides the LLM with the source code and test cases and asks it to translate the source code to the target programming language, ensuring it passes all test cases.

**Translation Repair Phase**

**Objective:** Repair any errors in the translated programs identified during the preliminary inspection.

**Process:**

* Error Analysis: Errors from the execution results are analyzed to extract necessary information such as error messages and line numbers.
* Repair Prompt: The LLM is prompted with the error information and instructed to fix the buggy lines in the code.
* Iterative Repair: The repair process can be repeated multiple times based on feedback from test case execution to ensure the accuracy of the translation.

**Repair Prompt**

"Given buggy ${pltar} code:\n${progtar}\nGiven test case:${tc\*}\nError message: ${err\_msg}\nFix the buggy line (marked ${com\_symtar} <Buggy Line>) in the buggy ${pltar} code according to the given error message. Use END\_OF\_CASE to finish your answer."

A white page with black text

Description automatically generated**Explanation:** This prompt gives the LLM the error information and asks it to fix the identified bugs in the translated code.

A diagram of a translation process

Description automatically generated

**Engine (AI Tool):**

**The LLMs** used in this study include GPT-3.5, LLaMA-13B, CodeGen, and LLaMA-7B.

**Learning-based Transpilers** used in this study include TransCoder, TransCoder-IR, and TransCoder-ST

Through this structured methodology, UniTrans harnesses both the capabilities of LLMs and the feedback from generated test cases, aiming to enhance translation accuracy, improve readability, and ultimately provide a more reliable automated code translation process. The authors conducted extensive experiments across different programming languages (Python, Java, and C++) to validate the effectiveness of this approach, demonstrating its substantial improvements over existing techniques.[1]Engine (AI Tool)  
The LLMs used in this study include GPT-3.5, LLaMA-13B, CodeGen, and LLaMA-7B. Learning-based transpilers used in this study include TransCoder, TransCoder-IR, and TransCoder-ST. Through this structured methodology, UniTrans harnesses both the capabilities of LLMs and the feedback from generated test cases, aiming to enhance translation accuracy, improve readability, and ultimately provide a more reliable automated code translation process. The authors conducted extensive experiments across different programming languages (Python, Java, and C++) to validate the effectiveness of this approach, demonstrating its substantial improvements over existing techniques.[1]

Datasets  
The dataset used in the paper consists of a collection of code samples for the purpose of evaluating automated code translation capabilities. Specifically, the dataset includes:[1][2]

1. Origin of the Dataset: The dataset is derived from an independent online platform called GeeksforGeeks, which contains numerous coding problems and solutions across multiple programming languages.
2. Breakdown of Unit Tests:
   * Python: 464 unit tests are present.
   * Java: 482 unit tests are included.
   * C++: 467 unit tests are part of the dataset.
3. **Data Cleaning:** The authors noted that many failures observed during preliminary experiments with LLMs were due to data noise and inconsistencies in the unit test scripts and parallel functions. To resolve this, the dataset was manually cleaned by researchers with extensive programming experience in Python, Java, and C++. A total of 252 errors were identified and corrected across the datasets [2]

A screen shot of a computer

Description automatically generated

1. **Test Case Generation Phase: [2]**

**A screenshot of a computer program

Description automatically generated**

1. **Translation Augmentation Phase: [2]**

A screenshot of a computer program

Description automatically generated

1. **Test Case Generation Phase: [2]**

A screen shot of a computer program

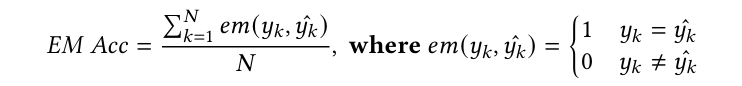
Description automatically generated

**Revaluation metrics:**

The paper employs several evaluation metrics to assess the performance of the proposed UniTrans framework in automated code translation. The key evaluation metrics used include:[1]

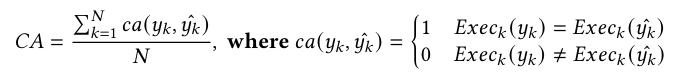
1. **Correctness Accuracy (CA):**

CA measures the proportion of the translated code that is correct based on predefined criteria. It evaluates whether the translated code meets the expected functionality as defined by the corresponding test cases.



where 𝑁 denotes the total number of translation samples,𝑦𝑘 denotes the ground truth of the 𝑘-th sample, and ˆ 𝑦𝑘 denotes the translated program via a certain transpiler for the 𝑘-th sample. For example, given𝑦𝑘 and ˆ 𝑦𝑘, only if they are exactly identical, they can be deemed a correct translation in EM Acc. Thus, EM Acc concludes the lower-bound of the effectiveness of transpilers.

1. **Execution Match Accuracy (EM Acc):**

 This metric assesses how well the translated code performs when executed. It compares the outputs of the translated code against the expected outputs derived from test cases. A higher EM Acc indicates that the translated code functions correctly in execution.

where 𝑁,𝑦𝑘, and ˆ 𝑦𝑘 carry the same meaning as those in EM Acc, 𝐸𝑥𝑒𝑐𝑘(·) denotes the execution result of a program with the test suite of the 𝑘-th sample. For example, even if𝑦𝑘 and ˆ 𝑦𝑘 are not identical literally, they are considered a correct translation in CA, as long as 𝐸𝑥𝑒𝑐𝑘(𝑦𝑘) = 𝐸𝑥𝑒𝑐𝑘( ˆ 𝑦𝑘).

**3. Precision (PR):** Used particularly in the context of iterative repairs, this metric measures the average percentage of unit tests passed by repaired programs. It provides a more granular view of performance compared to CA.

**Results:**

The paper presents several significant results concerning the performance of various Large Language Models (LLMs) for code translation, as well as the impact of the proposed UniTrans framework. Here are the key findings: [1][2]

1. **Performance of LLMs:**

* GPT-3.5 consistently outperformed all other models tested, achieving an average Correctness Accuracy (CA) of 87.92% and an Execution Match Accuracy (EM Acc) of 18.04% across multiple translation tasks.
* LLaMA-33B also demonstrated strong performance, often surpassing other state-of-the-art learning-based transpilers in terms of both CA and EM Acc for most translation datasets.
* The results showed that as the parameter size of the LLMs increased, their translation performance generally improved.

A table with numbers and text

Description automatically generated

1. **Comparison Between Models:**

* CodeGen and LLaMA-7B, which have similar parameter sizes but different pre-training methodologies, exhibited comparable performance, with CodeGen often performing better in terms of CA, and LLaMA-7B achieving higher EM Acc scores.
* TransCoder-IR outperformed TransCoder-ST regarding EM Acc, while TransCoder-ST performed better concerning CA. This distinction highlights the varying focuses of these models on semantic and lexical correctness.

1. **Effectiveness of UniTrans:**

* The proposed UniTrans framework demonstrated substantial improvements in code translation efficacy for different LLMs.
* GPT-3.5 saw an average improvement of 4.02% in CA and 13.28% in EM Acc when using UniTrans.
* LLaMA-13B achieved average improvements of 19.20% in CA and 36.42% in EM Acc.
* LLaMA-7B showed the highest improvements, with an average of 28.58% in CA and 71.22% in EM Acc.

A table with numbers and a number of data

Description automatically generated with medium confidence **4. Statistical Significance:**

The improvements achieved by UniTrans were statistically significant, indicating that the enhancements were not due to random chance

1. **Impact of Test Cases and Repair:**

* The study found that the Test Case Generation Phase and Translation Augmentation Phase of UniTrans significantly enhanced the LLMs' code translation performance.
* Adding test cases led to improvements in understanding program logic and identifying discrepancies between source and target programs.
* The Translation Repair Phase further fixed errors in the translated programs, contributing to overall better performance.

**Conclusion:**

* The results indicate that UniTrans, combined with LLMs, can significantly improve the accuracy and reliability of automated code translation.
* The findings highlight the potential of using LLMs for code translation tasks and suggest that incorporating test cases and iterative repair can further enhance their performance.
* This research highlights directions for future improvements in code translation methods and the potential for better prompt designs in automated code translation.

**Reflection:**  
  
 **Contributions:**

The paper presents a significant advancement in the field of automated code translation by leveraging the power of Large Language Models (LLMs) such as GPT-3.5 and LLaMA. The primary contributions of this paper include: [1]

1. **Empirical Study:** The paper conducts an extensive empirical study comparing recent LLMs with state-of-the-art learning-based transpilers. This comparison highlights the strengths and weaknesses of LLMs in code translation tasks.
2. **UniTrans Framework:** The introduction of the UniTrans framework is a pivotal contribution. This unified code translation framework enhances the translation accuracy by incorporating auto-generated test cases and iterative repair mechanisms. The framework is shown to be effective across different LLMs and programming languages.
3. **Cleaning and Extending Datasets:** The authors rigorously cleaned and extended existing datasets, ensuring a robust evaluation environment. This effort improves the reliability of the experimental results.
4. **Significant Improvements**: The paper demonstrates that the UniTrans framework can significantly boost the performance of LLMs, achieving substantial improvements in computational accuracy and exact match accuracy.
5. **Practical Implications:** By showcasing the practical utility of LLMs in code translation tasks without the need for extensive fine-tuning, the paper provides a feasible path for real-world applications.

**Limitations:**

Despite its contributions, the paper has a few limitations: [1]

1. **Scope of the Dataset:** While the cleaning of the dataset is commendable, the reliance on a single source (GeeksforGeeks) may limit the diversity and complexity of the coding problems represented. A broader dataset that encompasses various coding styles and real-world scenarios might yield more generalizable findings.
2. **Focus on a Limited Number of Models:** Although the paper includes several prominent LLMs, the exploration of alternative or emerging models could provide a more comprehensive understanding of the capabilities in code translation. Future research could include a wider array of models, especially those that leverage different architectures or training methodologies.
3. **Precision Issues:** The UniTrans framework shows less improvement in handling precision-related errors. This indicates that there is room for enhancement in detecting and correcting minor discrepancies in translated code.
4. **Execution Match Accuracy (EM Acc) Metric:** The use of EM Acc as an evaluation metric raises some concerns, as it might not fully encompass the semantic correctness of translated code. Incorporating additional metrics that evaluate the functional validity or efficiency of the translated programs could provide a more holistic view of translation quality.

**Potential for Further Research:**

1. Exploring Additional Prompt Designs: The paper encourages exploring alternative prompt designs tailored for specific tasks within the code translation framework. This is a fertile area for research, as better prompts could lead to improved model understanding and output quality. [1]
2. Broader Evaluation Benchmarks: Future research could establish more comprehensive benchmarks that incorporate a wider variety of programming languages, projects, and coding scenarios, providing a more rigorous evaluation of automated code translation systems. [1]
3. Integration of Hybrid Approaches: Combining learning-based methods with rule-based systems could offer a middle ground, possibly leading to enhanced accuracy and robustness in code translation outputs. Research that explores hybrid architectures and their applicability in real-world scenarios could be particularly impactful. [1]
4. Real-World Application Studies: Conducting case studies in real-world settings would provide insights into the practical challenges and implications of deploying automated code translation systems using frameworks like UniTrans. This could help identify unaddressed issues that arise in operational contexts and inform further improvements to the framework. [1]
5. Advancing Model Interpretability and Transparency: As the use of LLMs becomes more widespread, research into understanding and interpreting the decision-making processes of these models will be essential. This could include developing techniques for visualizing model behavior or establishing frameworks for auditing model outputs for correctness and reliability. [1]

In conclusion, this paper represents a significant advancement in the realm of automated code translation and offers valuable insights for both practitioners and researchers. While it lays a strong foundation for future work, addressing its limitations and exploring new avenues for research will be key to furthering the field and enhancing the reliability and effectiveness of automated translation technologies. [1]

[1] Yang, Z., Liu, F., Yu, Z., Keung, J. W., Li, J., Liu, S., ... & Li, G. (2024). Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, *1*(FSE), 1585-1608.‏

[2] [yz1019117968/FSE-24-UniTrans: Source Code for "Exploring and Unleashing the Power of Large Language Models in Automated Code Translation"](https://github.com/yz1019117968/FSE-24-UniTrans)

**PAPER2:  
Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code**

**Authors:** Rangeet Pan, Ali Reza Ibrahim Zada, Rahul Krishna, Divya Sankar, Lambert Pougeum Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, Reyhaneh Jabbar Vand  
**Affiliation:**   
IBM Research, University of Illinois Urbana-Champaign

**Abstract**

This paper explores the limitations and challenges of Large Language Models (LLMs) in code translation, focusing on their performance across various programming languages. A large-scale empirical study involving the translation of 1,700 code samples from five programming languages (C, C++, Go, Java, and Python) reveals that LLMs produce correct translations only 2.1% to 47.3% of the time. The authors identify 15 categories of translation bugs and propose a prompt-crafting approach to improve translation accuracy by an average of 5.5%.

.

**Method**

1- **Empirical Study:** The research involves a large-scale empirical study, translating 1,700 code samples across five programming languages: C, C++, Go, Java, and Python.

2 - **Bug Taxonomy:** The authors categorize translation failures into 15 distinct types, analyzing the root causes of these bugs.

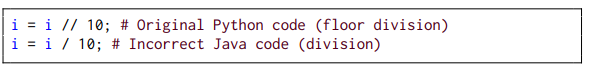
3- **Prompt-Crafting Approach:** A new approach is proposed that enhances contextual information provided to the LLMs during translation, aiming to reduce errors.

صورة تحتوي على نص, لقطة شاشة, رقم, الخط

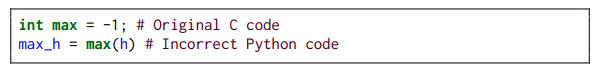
تم إنشاء الوصف تلقائياً

**For example:**

1-**Incorrect use of operators:**



2- **Inclusion of logic not in source code:**

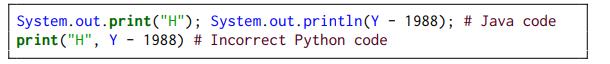


3- **Incorrect input parsing:**

صورة تحتوي على نص, لقطة شاشة, الخط, خط

تم إنشاء الوصف تلقائياً

4- **Output formatting:**



**Dataset**

**1-Composition:**

The dataset consists of 1,700 executable code samples.

It covers five programming languages: C, C++, Go, Java, and Python.

2- **Sources:**

**Benchmarks:**  
 The dataset is derived from three established code benchmarks:

1-CodeNet

2-Avatar

3-EvalPlus

**Real-World Projects:**   
It also includes code samples from two real-world projects:

1-Apache Commons CLI

2-Python Click

**3-Characteristics:**

The dataset includes over 10,000 tests designed to evaluate the functionality and correctness of the code.

It features 43,000+ translated code snippets, ensuring a diverse range of examples for translation tasks.

**4-Accessibility:**

The dataset is publicly available, which facilitates further research and experimentation in the area of code translation and LLM evaluation.

**Evaluation Metrics:**

1. **Correctness Rate:**

The primary metric for evaluating the success of code translations is the correctness rate. This measures the percentage of translated code snippets that:

**1: Compile successfully.**

**2: Pass runtime checks.**

**3: Successfully execute all relevant tests.**

**2. Type and Frequency of Bugs:**

The evaluation also includes metrics related to the type and frequency of identified bugs. By analyzing how often different types of bugs occur, the study can identify common pitfalls in LLM translations.

**3- Prompt Effectiveness:**

The effectiveness of the proposed prompt-crafting technique is measured by comparing the success rates of translations before and after implementing the new prompting strategies. This provides insight into how contextual information impacts translation accuracy.

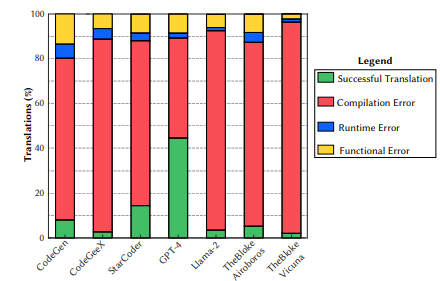
**Results**

صورة تحتوي على نص, لقطة شاشة, رقم, الخط

تم إنشاء الوصف تلقائياً

صورة تحتوي على نص, لقطة شاشة, رقم, الخط

تم إنشاء الوصف تلقائياً



**1-Translation Success Rates**:   
  
The study reveals that LLMs have a success rate ranging from 2.1% to 47.3%, with notable differences in performance based on the source and target languages.

**2-Bug Identification**:   
The taxonomy identifies 15 categories of translation bugs, including issues related to syntax, semantics, and logic.

**3- Prompt Improvement:**   
The application of a prompt-crafting technique shows an average improvement of 5.5% in translation success rates, particularly benefiting from providing additional context related to previous translation failures.

**4-Comparison with Non-LLM Techniques:**   
The analysis shows that traditional non-LLM-based techniques have unique strengths and weaknesses compared to LLM-based approaches, suggesting that a hybrid solution may be optimal for code translation tasks.

**Reflection**

This research highlights the ongoing challenges faced by LLMs in the field of code translation, underscoring the necessity for innovative approaches such as prompt crafting and bug taxonomy development. The findings suggest that while LLMs exhibit promise, significant improvements are required to make them reliable for practical code translation tasks. The authors provide valuable insights and data to encourage further research and development in this area, fostering collaboration within the software engineering community.

**References:**

Pan, R., Ibrahim zada, A. R., Krishna, R., Sankar, D., Wassi, L. P., Merler, M., ... & Jabbarvand, R. (2024, April). Lost in translation: A study of bugs introduced by large language models while translating code. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (pp. 1-13).

https://arxiv.org/abs/2308.03109

**Paper 3:  
  
TRANSAGENT: An LLM-Based Multi-Agent System for Code Translation**

**Authors:**  
Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, & Yiling Lou

**Affiliation:**  
Department of Computer Science, Fudan University, China

**Abstract**

Code translation is essential for software migration, system modernization, and cross-platform development. Traditional rule-based methods require significant manual effort and often result in poor readability. Learning-based approaches leverage parallel datasets but suffer from data scarcity and high computational costs. Large Language Models (LLMs) provide a promising alternative but often introduce syntax and semantic errors.

To address these challenges, TRANSAGENT introduces a multi-agent system that improves LLM-based code translation using an iterative error detection and correction framework. Unlike single-pass LLM translations, TRANSAGENT employs four specialized agents—Initial Code Translator, Syntax Error Fixer, Code Aligner, and Semantic Error Fixer—working together to refine translations. By integrating control-flow analysis and execution alignment, TRANSAGENT enhances accuracy, reduces errors, and outperforms existing translation models such as UniTrans and TransCoder.

**Introduction**

Code translation plays a critical role in software engineering by enabling seamless interoperability between different programming languages. Traditional translation methods, such as rule-based techniques, rely on predefined syntactic mappings, which are labor-intensive and prone to producing unreadable code. Learning-based approaches, such as deep learning models trained on parallel datasets, have improved translation quality but still struggle with data limitations and high training costs (Rozier et al., 2020). The recent rise of LLMs has introduced a new paradigm for automated code translation. However, LLM-generated translations often contain syntax errors, functional discrepancies, and debugging challenges (Pan et al., 2024).

To address these challenges, TRANSAGENT employs a multi-agent system that enhances LLM-based code translation by iteratively detecting and correcting errors. It uses four specialized agents to localize faulty code, apply structured fixes, and refine translations. By leveraging execution alignment, TRANSAGENT ensures functional equivalence between source and translated programs, significantly improving translation effectiveness.

**Methodology**

**1. Initial Code Translator**

This agent generates an initial translation using an LLM and produces test cases to validate functional correctness. It ensures that the translated code maintains structural integrity while adhering to the syntax of the target language.

**2. Syntax Error Fixer**

Using compiler or interpreter feedback, this agent identifies syntax errors and applies structured fixes. The iterative correction process enhances the syntactic validity of the translated code before execution.

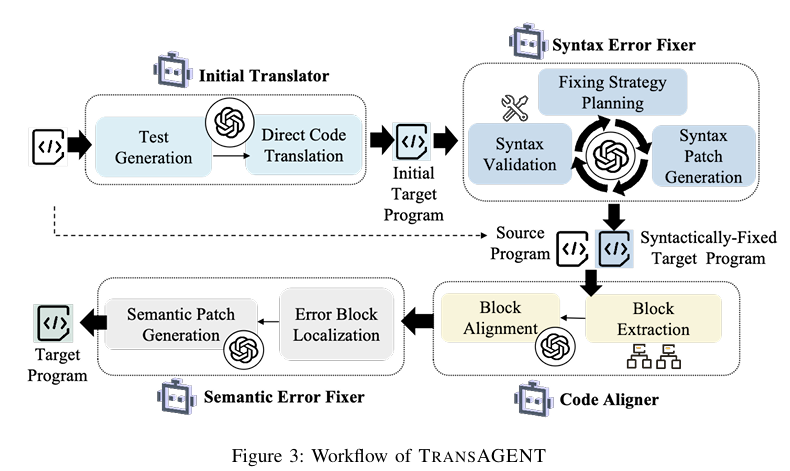
**3. Code Aligner**

To improve alignment accuracy, this agent maps corresponding code blocks between source and target programs using control-flow analysis and LLM-based reasoning. This block-level mapping method ensures that translations remain functionally consistent.

**4. Semantic Error Fixer**

By comparing runtime values in both source and target programs, this agent pinpoints faulty code blocks and applies fine-grained corrections. This process minimizes logical inconsistencies and enhances execution reliability.

**5. Benchmark for Evaluation**

To ensure unbiased evaluation, a new benchmark dataset was created using programming challenges released after August 2023. This dataset prevents contamination from LLM training data and enables reliable assessment of translation quality.   


**Dataset**

Existing code translation benchmarks often suffer from data leakage, as many datasets are sourced from public programming competitions, making them susceptible to pretraining contamination. Additionally, some benchmarks focus on only two programming languages, limiting their generalization scope. Others, such as CodeNet, contain incomplete or incorrect code samples, reducing their reliability for evaluating translation performance (Ahmad et al., 2023).

To address these limitations, TRANSAGENT's benchmark dataset consists of 614 translation tasks covering three widely used languages: Java, Python, and C++. It includes 210 Python-to-Java, 200 Python-to-C++, and 204 Java-to-C++ translation pairs. Each task contains manually verified code samples that maintain functional equivalence across languages. Additionally, each task is supplemented with 10 automatically generated and validated test cases, achieving over 98% line coverage. This dataset provides a realistic, challenging test bed for evaluating LLM-based code translation techniques and mitigates biases found in previous datasets.

**Evaluation Metrics**

The effectiveness of TRANSAGENT is evaluated using the following key metrics:

* Computational Accuracy (CA): Measures whether the translated program produces the same output as the original for all test cases.
* CodeBLEU: Evaluates the structural similarity between source and translated code.
* Mapping Accuracy: Assesses how well the Code Aligner maps corresponding blocks between source and translated programs.
* Efficiency: Measures the average translation time per example and the number of iterations required to fix errors.
* Generalization: Evaluates TRANSAGENT’s performance across different LLMs, including Llama3-8B, ChatGLM2-6B, and DeepSeekCoder-6.7B.

**Results**

* Higher Translation Accuracy: TRANSAGENT outperforms both UniTrans and TransCoder in Computational Accuracy (CA) and CodeBLEU.
* Significant Python-to-Java CA Improvement: Achieved a 33.3% increase (from 56.2% to 89.5%) compared to UniTrans.
* Improved Code Mapping: The Code Aligner enhanced mapping accuracy by 39.6% over TransMap.
* Enhanced Semantic Error Fixing: The Semantic Error Fixer significantly improves runtime correctness compared to prior approaches.
* Increased Efficiency: TRANSAGENT requires only 19s per example, making it faster than UniTrans (24s per example).
* Better Generalization: Achieved over 30% accuracy gains when applied to weaker models like ChatGLM2-6B.

A table with numbers and text

Description automatically generated

**Conclusion**

TRANSAGENT represents a significant advancement in LLM-based code translation by introducing a multi-agent framework for detecting and correcting translation errors. By integrating syntax and semantic error fixing, control-flow-based mapping, and iterative refinement, TRANSAGENT outperforms state-of-the-art translation models while maintaining efficiency. Future work will explore expanding the dataset to additional languages and refining error-fixing strategies to further enhance translation robustness.

**References**

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2023). Summarize and generate to back-translate: Unsupervised translation of programming languages. *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*.

Pan, R., Ibrahimzada, A. R., Krishna, R., Sankar, D., et al. (2024). Lost in translation: A study of bugs introduced by large language models while translating code. *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*.

Rozier, B., Lachaux, M.-A., Chanussot, L., & Lample, G. (2020). Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems (NeurIPS)*.  
  
**Paper 4:**  
**Summary of "Unraveling the Potential of Large Language Models in Code Translation: How Far Are We?"**

**Authors**: Qingxiao Tao, Tingrui Yu, Xiaodong Gu, Beijun Shen  
**Affiliation**: School of Software, Shanghai Jiao Tong University, China

**Idea**

This study investigates the effectiveness of Large Language Models (LLMs) in code translation, focusing on their limitations due to inadequate training on parallel multilingual code data. The authors propose a novel benchmark, **PolyHumanEval**, which includes 14 programming languages to provide a comprehensive framework for evaluating LLM performance in code translation. The research aims to uncover the capabilities and limitations of LLMs in translating code between different programming languages.

**Methodology**

**Benchmarking**:   
  
The PolyHumanEval benchmark was developed by extending the existing HumanEval dataset to include 14 programming languages: C++, C#, Dart, Go, Java, JavaScript, Kotlin, PHP, Python, Ruby, Rust, Scala, Swift, and TypeScript. Each solution was handcrafted to ensure semantic consistency across languages.

**Experimental Setup**:  
  
 The study evaluated multiple LLMs, including CodeLlama (7B, 13B, and 34B) and StarCoder, across 26 translation tasks. The experiments involved translating code snippets from various source languages into target languages and vice versa.

**Prompt Designs**:  
  
 Different prompt configurations were tested to enhance translation effectiveness. These included prompts with target function signatures and required library imports to provide additional context to the LLMs.

**Dataset**

The **PolyHumanEval** benchmark features 2,296 verified solutions for 164 programming problems. The dataset was created through a meticulous process that involved:

* Handcrafting solutions for each programming problem.
* Validating each solution through peer reviews and extensive testing to ensure correctness and semantic alignment.
* Employing a rule-based tool to automatically generate equivalent test programs across all selected languages, ensuring accuracy and consistency.

**Evaluation Metrics**

The primary evaluation metric used in this study is **computational accuracy (CA)**, which assesses whether the source and generated code produce the same output when given identical inputs. The evaluation focuses on the first generated result for each translation task (i.e., pass@1).

**Results**

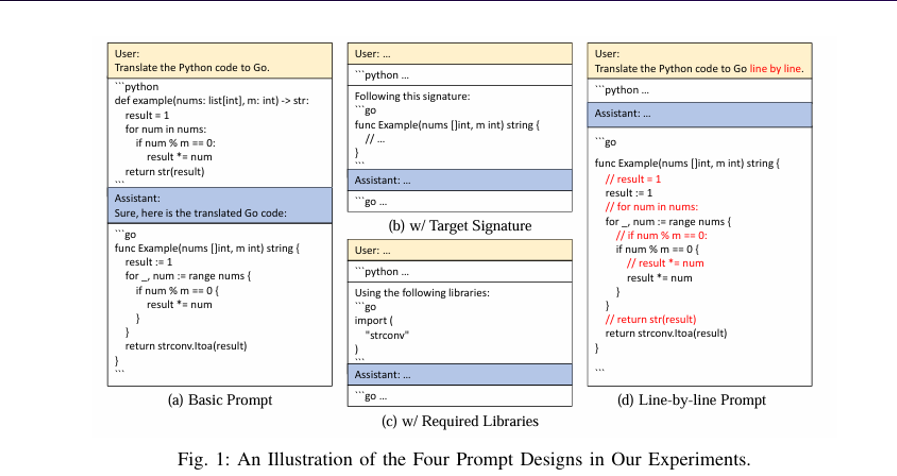
LLMs demonstrated higher computational accuracy when translating code into Python, achieving an average CA score of 86.16 for translations to Python, while the score dropped to 66.93 for translations from Python to other languages.

The intermediary translation method utilizing Go showed significant improvements in translation accuracy, validating its role as an effective lingua franca for code translation tasks.

Self-training methods led to substantial optimization, with verified self-generated datasets improving performance across both targeted and broader translation tasks.

The results also indicated that prompt designs significantly affected translation effectiveness, with prompts that included target function signatures yielding better performance.

* **An Illustration of the Four Prompt Designs in Their Experiments**

****

**Results of Different Prompts**

* The performance of CodeLlama-13B under various prompting conditions is summarized below:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Prompt Design | C++ | C# | Dart | Go | Java | JS | Kotlin | PHP | Ruby | Rust | Scala | Swift | TS | Avg. |
| Basic | 78.05 | 85.98 | 75.61 | 87.80 | 85.98 | 82.32 | 81.71 | 81.10 | 82.93 | 82.93 | 85.37 | 80.49 | 81.71 | 82.46 |
| w/ Target Signature | 81.10 | 91.46 | 82.32 | 90.85 | 87.80 | 83.54 | 85.37 | 82.32 | 90.24 | 87.20 | 88.41 | 84.76 | 84.76 | 86.16 |
| w/ Required Libraries | 81.71 | 86.59 | 85.37 | 90.24 | 87.80 | 81.71 | 85.98 | 80.49 | 87.20 | 89.63 | 85.98 | 81.71 | 84.76 | 85.32 |
| Line-by-line Prompt | 77.44 | 88.41 | 69.51 | 85.37 | 82.32 | 79.88 | 79.27 | 78.05 | 81.71 | 72.56 | 80.49 | 81.10 | 77.44 | 79.50 |

**Reflection**

The findings underscore the ongoing challenges that LLMs face in code translation, emphasizing the need for diverse, high-quality training data and innovative methods, such as intermediary translation and self-training. The asymmetrical performance of LLMs indicates that tailored approaches must consider the unique characteristics of various programming languages. The identification of Go as an effective intermediary language highlights the importance of language selection in improving translation efficacy.

Overall, this study significantly contributes to the understanding of LLMs in code translation, laying a foundation for future research aimed at refining techniques to enhance their capabilities. By providing open-source resources, the authors promote collaboration and innovation within the research community, fostering further advancements in this dynamic field.

**Towards Translating Real-World Code with LLMs: A Study of Translating to Rust**

**Authors:** Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, Daniel Kroening

**Affiliation:** Department of Computer Science, Fudan University, China

**1.1 Idea**

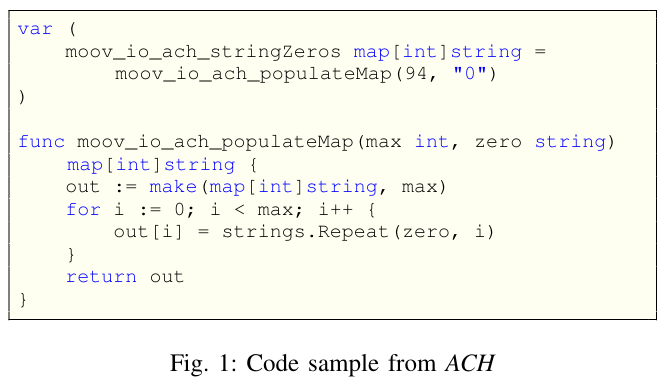
The study investigates the effectiveness of large language models (LLMs) in translating real-world code into Rust. It focuses on assessing the capabilities of five state-of-the-art LLMs—GPT-4, Claude 3, Claude 2.1, Gemini Pro, and Mixtral—highlighting the challenges and methodologies for achieving accurate and idiomatic translations.

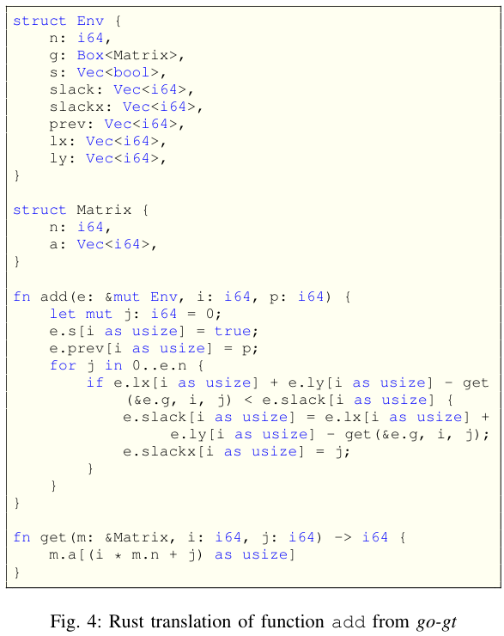
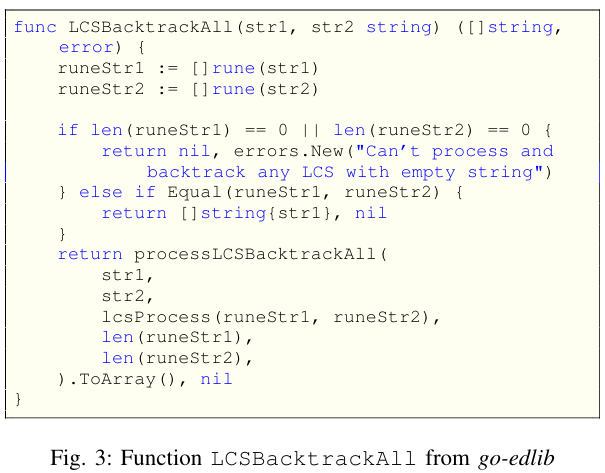
**1.2 Method**

Tool Development: The authors developed FLOURINE, an end-to-end code translation tool capable of producing validated Rust translations. Once the translation compiles, FLOURINE employs cross-language differential fuzzing to test the I/O equivalence between the source program and the Rust translation.

**1.3 Datasets**

The study utilized code samples extracted from seven open-source projects written in C and Go, selected for their relevance to low-level programming tasks. The projects included:

1. ACH: Go library implementing a reader, writer, and validator for banking operations.
2. geo: Go library implementing common geometry functions and interval arithmetic.
3. libopenaptx: C library for audio processing.
4. opl: C library for sound card emulation.
5. صورة تحتوي على نص, لقطة شاشة, الخط, رقم

   تم إنشاء الوصف تلقائياًgo-gt: Go library for graph algorithms.
6. go-edlib: Go library for string comparison and edit distance algorithms.
7. triangolatte: Go library for 2D triangulation.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Project | Language | #Benchmarks | Min LoC | Max LoC | Avg LoC | Min # Func | Max # Func | Avg # Func |
| ACH | Go | 121 | 43 | 194 | 64 | 3 | 7 | 3.4 |
| geo | Go | 67 | 13 | 70 | 35 | 3 | 7 | 4.1 |
| libopenaptx | C | 31 | 13 | 173 | 69 | 1 | 9 | 2.9 |
| opl | C | 81 | 19 | 460 | 67 | 1 | 15 | 2.8 |
| go-gt | Go | 43 | 9 | 213 | 51 | 1 | 16 | 3.5 |
| go-edlib | Go | 36 | 13 | 597 | 62 | 1 | 25 | 3.1 |
| triangolatte | Go | 29 | 9 | 164 | 38 | 1 | 10 | 2.5 |

TABLE I: Benchmark details

**1.4 Evaluation Metrics**

Translation Success Rate: The percentage of successfully translated benchmarks that compile and pass the fuzzing tests.

Linter Warnings: Analysis of idiomatic usage in the generated Rust code, evaluated using Clippy, Rust's linter.

Feedback Strategy Effectiveness: Comparison of initial and final success rates when using various feedback strategies.

**1.5 Results**

The most successful LLM, Claude 2, achieved a translation success rate of 47%.

Overall, LLMs achieved success rates ranging from 21% to 47% across 8160 code translation experiments.

Feedback strategies improved success rates by up to 8% on average.

Larger programs tended to have lower success rates, and the study found that counterexamples, particularly those generated by fuzzers, were ineffective as feedback for the LLMs.

|  |  |
| --- | --- |
| Translation Success Rates by LLM | |
| LLM | Success Rate (%) |
| Claude 2 | 47.7 |
| Claude 3 | 43.9 |
| GPT-4 Turbo | 36.9 |
| Gemini Pro | 33.8 |
| Mixtral | 21.0 |

**2.1 Reflection on Contributions, Limitations, and Potential for Further Research**

The study introduces FLOURINE, a novel tool for validating Rust translations without handwritten test cases, marking the first substantial exploration of using large language models (LLMs) for translating real-world code beyond synthetic benchmarks. It also develops a unique cross-language differential fuzzing method to assess translation quality.

Github Repository link

<https://github.com/abmj77/CCSW325-codetranslation.git>