Conversational Agent for Code Translation to Rust

QUOC HUNG LE, YUNG HSUAN TENG*, KRISHNA PRASHANT PATEL*, and DHYEY SAMIRBHAI SHAH*, North Carolina state University, USA

ABSTRACT HERE

ACM Reference Format:
Quoc Hung Le, Yung Hsuan Teng, Krishna Prashant Patel, and Dhyey Samirbhai Shah. 2024. Conversational Agent for Code Translation to Rust. 1, 1 (December 2024), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Code translation is the process of transforming code from one programming language to another while preserving its functionality. Automated code translation aims to achieve this without human interference, utilizing either rule-based methods, which rely on code attributes like Abstract Syntax Trees (AST) and Intermediate Representations (IR), or machine learning-based approaches that treat code translation as a natural language translation task. This paper focuses on translating C code to Rust, a modern language known for its memory safety and performance benefits. Despite these advantages, many developers are hesitant to adopt Rust due to the complexity of migrating existing codebases. Automated translation from C to Rust addresses this challenge by facilitating the transition, enhancing maintainability, and expanding application compatibility across multiple platforms

1.1 Structure of the paper

The structure of this paper is designed to systematically explore the process and effectiveness of translating C code to Rust using conversational agents. It begins with an Introduction that outlines the challenges and benefits of automated code translation, particularly highlighting Rust's advantages such as memory safety and performance. Part 2: Motivation delves into the current limitations of automated code translation and underscores the need for improved methods to facilitate software development and maintenance. Part 3: Method describes our innovative approach, which integrates state-of-the-art solutions for code translation repair with a novel multi-agent communication method, detailing the phases of initial translation, compilation validation, fuzz testing, and multi-agent collaboration. Finally, Part 4: Research Questions focuses on evaluating the performance of smaller Large Language Models (LLMs) in comparison to established benchmarks and examines how Multi-Agent Communication (MAC) can enhance their performance relative to larger models. This structured approach allows for a comprehensive examination of our proposed method, from theoretical foundations to practical implementation and evaluation

Authors' Contact Information: Quoc Hung Le, qle3@ncsu.edu; Yung Hsuan Teng, yteng2@ncsu.edu; Krishna Prashant Patel, kpatel43@ncsu.edu; Dhyey Samirbhai Shah, dshah22@ncsu.edu, North Carolina state University, Raleigh, NC, USA.

ACM XXXX-XXXX/2024/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.2 Motivation

The motivation for automated code translation from C to Rust stems from several key factors that address critical needs in software development and maintenance. Firstly, it facilitates the migration of existing projects to integrate with diverse platforms while ensuring compatibility, addressing the growing demand for cross-platform solutions. Secondly, it preserves the usability of systems built on outdated or unsupported programming languages, extending the lifespan of valuable legacy code. Additionally, automated translation addresses the hesitation many developers face in adopting Rust, despite its significant memory safety and performance benefits, by providing a smoother transition path from C.

The benefits of this approach are multifaceted. It contributes to developer growth by familiarizing programmers with modern language models like Rust, bridging the gap between legacy and contemporary programming paradigms. The translation process also enhances platform versatility, expanding application compatibility across multiple platforms and increasing the reach of existing software. Perhaps most importantly, it leads to an enhanced user experience by leveraging the advanced properties of modern programming languages. For instance, Rust's memory safety features are particularly crucial for banks and security-critical applications, where robustness and reliability are paramount. By automating the translation process, organizations can more readily access these benefits without the substantial time and resource investment typically required for manual code migration.

1.3 Research Methodology

Our method for code translation from C to Rust employs a multi-phase approach that leverages the strengths of multiple Large Language Models (LLMs) and iterative refinement techniques. The process begins with an initial translation phase, where Mistral-7b-instruct generates a baseline Rust translation of the input C code. This is followed by a compilation validation phase, which systematically checks the generated Rust code for compilation errors. When failures occur, an error-driven refactoring approach is implemented, utilizing compiler error messages to guide the identification and resolution of specific translation issues.

The third phase involves comprehensive fuzz testing to ensure the reliability and functionality of the translated code. This step helps identify potential semantic discrepancies that might not be apparent through compilation checks alone. The final and most innovative aspect of our approach is the multi-agent collaboration phase, which implements a Mistral-Llama feedback loop. This phase iteratively processes counterexamples to improve translation accuracy and address identified issues.

Our method distinguishes itself through several key features. It integrates multiple LLMs (Mistral and Llama) in a coordinated manner, enabling efficient and iterative repair processes. Additionally, it implements a counterexample-driven feedback mechanism that systematically addresses translation errors through targeted improvements. Finally, the approach maintains lightweight configurations throughout the system, ensuring cost-effectiveness and scalability. This comprehensive methodology aims to enhance the accuracy and reliability of C to Rust code translation, addressing challenges such as semantic preservation and error handling in translated code.

2 Previous Works

2.1 Explanation of Previous Works

Automated Repair of Programs from Large Language Models (2023)
Fan et al. [2] systematically explored the potential of Automated Program Repair (APR) techniques in fixing errors in code generated by large language models (LLMs), such as Codex. Using the LMDefects dataset, comprising 113 Java tasks from LeetCode, their study revealed that 57% of bugs were algorithm-related, while 11% were syntax errors. APR tools such as TBar and Recoder were evaluated for their effectiveness, with Recoder outperforming TBar by successfully fixing 8 tasks compared to TBar's 6. Despite these successes, APR tools were constrained

by limited patch spaces, inability to handle multi-line fixes, and lack of program dependency awareness. A significant insight was that combining repair tools, such as leveraging Codex-edit mode with TBar, could provide more comprehensive solutions by synthesizing multi-hunk patches. This work underscores the value of hybrid approaches and points to the critical need for enhanced fault localization and semantic reasoning for robust program repair.

Towards Translating Real-World Code with LLMs (2024)
Eniser et al. [1] investigated the application of LLMs for translating real-world code from C and Go to Rust, employing their FLOURINE tool for translation and validation. The study tested five state-of-the-art models, including GPT-4 and Claude 3, on code extracted from seven open-source projects. Results indicated that the best-performing model achieved a success rate of 47% in producing semantically correct Rust translations. Feedback strategies, such as counterexamples and hinted restarts, contributed to a 6–8% improvement in translation success rates. A notable achievement of the study was its innovative use of cross-language differential fuzzing to validate I/O equivalence without relying on pre-existing test cases. This enabled FLOURINE to identify and correct critical translation errors effectively. However, challenges such as Rust's stringent safety requirements and handling of complex user-defined types remained significant. The study demonstrated the potential of feedback-driven iterative refinement for improving translation accuracy and reliability in real-world applications.

2.2 Motivation

These prior works provided the foundational basis for the methodology implemented in our project, *Conversational Agent for Code Translation from C to Rust*. Fan et al.'s research highlighted the limitations of LLM-generated code and the importance of integrating semantic reasoning into repair workflows. Their findings on the use of APR tools to address algorithmic misalignment and syntax errors directly influenced our decision to implement conversational LLM repair strategies. Moreover, their demonstration of the efficacy of combining tools such as Codex-edit mode with APR tools inspired us to explore multi-model collaboration for enhanced translation accuracy.

Eniser et al.'s work provided a robust framework for evaluating code translations, particularly through their innovative application of cross-language differential fuzzing. This approach informed our strategy for validating the correctness and functionality of Rust translations. Additionally, their insights into feedback-driven refinement motivated our use of counterexample-based improvement loops to iteratively enhance translation results. By addressing challenges such as semantic preservation and error handling in translated code, our project aims to contribute to the growing body of work on LLM-based automated code translation, with a specific focus on transitioning legacy C codebases into modern, memory-safe Rust implementations.

These works collectively guided the development of our methods, emphasizing robust validation mechanisms, iterative feedback strategies, and the integration of multiple models to handle complex real-world code scenarios effectively.

3 Methodology

3.1 Approach Overview

Our proposed approach integrates the state-of-the-art solution for code translation repair and an innovative LLM communication method. The approach comprises four distinct phases: initial translation, compilation validation, fuzz testing, and multi-agent collaboration.

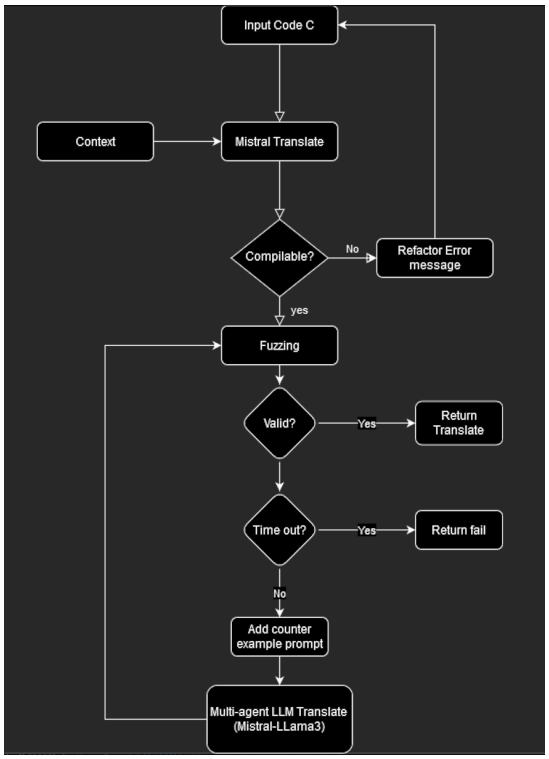
In the initial translation phase, we leverage Mistral Instruct to generate Rust code from the input C code. This foundation serves as the starting point for our iterative refinement process, ensuring we begin with a baseline translation that captures the core functionality of the original code.

The compilation validation phase follows, where we systematically verify the generated Rust code's compilation success. When compilation failures occur, our system implements an error-driven refactoring approach. This process utilizes compiler error messages as guidance to identify and address specific translation issues, ensuring the output maintains syntactic correctness and adheres to Rust's strict safety requirements.

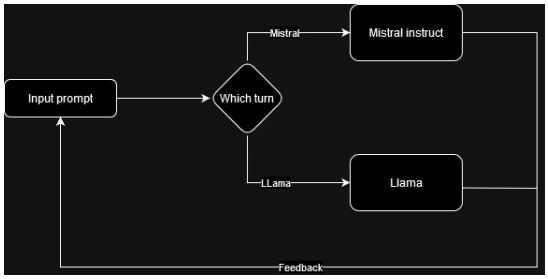
The third phase involves comprehensive fuzz testing to ensure the translated code's reliability. This phase serves two critical purposes: first, it performs randomized input testing to validate the functionality of the translated code, and second, it verifies that the outputs maintain consistency with the original C code's behavior. This systematic validation helps identify potential semantic discrepancies that might not be apparent through compilation checks alone.

The final and most innovative aspect of our approach is the multi-agent collaboration phase, which implements a Mistral-Llama feedback loop. The system iteratively processes counterexamples, using them to improve translation accuracy and address any identified issues.

Our approach distinguishes itself through several innovative features. First, we integrate multiple LLMs (Mistral and Llama) in a coordinated manner, enabling efficient and iterative repair processes. Second, we implement a counterexample-driven feedback mechanism that systematically addresses translation errors through targeted improvements. Finally, we maintain lightweight configurations throughout the system, ensuring cost-effectiveness and scalability.



(a) System workflow diagram



(b) multi-LLMs communication architecture

3.2 Experiment Setting

Our experimental evaluation utilizes established datasets from prior code translation research [X] to enable direct comparison with existing approaches. The evaluation framework comprises four key components: datasets, translation model parameters, fuzzing configuration, and multi-LLM settings, designed to comprehensively assess our proposed method's effectiveness.

- Datasets
  We evaluated our approach on two distinct C libraries:
  - `libopenaptx`: A comprehensive C library designed for audio processing applications, consisting of 31 source files
  - `opl`: A specialized C library focused on sound card emulation, comprising 81 source files
- Translation Model Configuration
  For the initial compilable code translation phase, we employed the following configuration:
  - Model: Mistral-7b-instruct
  - Maximum repeat iterations: 12
- Fuzzer Parameters
  The fuzzing component was configured with the following parameters to ensure thorough testing:
  - Maximum initial input size: 32768 bytes
  - Execution timeout: 7 minutes per test case
  - Maximum retry attempts: 3
- Multi-LLM Configuration
  The multi-LLM fixing phase utilized two language models working in tandem:
  - Models: Llama 3.2-3b-instruct and Mistral-7b-instruct
  - Maximum rounds: 4
- Prompts
  The following are two main prompts that we utilize in our project:

4 Experiment results

In this section, we address two research questions that are closely related to the challenges presented by our problem.

RQ1: How do previous methods perform with single "small" Large Language Models?

In Flourine [1], most of the LLM engines utilized are large-scale models, such as GPT-4, Claude, and Gemini. While these models demonstrate state-of-the-art performance, they come with significant resource demands that make them impractical for deployment on less capable machines. This limitation creates a barrier for applications requiring scalability and efficiency on devices with constrained computational capacity.

To address this challenge, we investigated the potential of smaller models by incorporating Mistral-7B-Instruct into our evaluation pipeline. This model was selected for its reduced size and computational requirements, making it a more accessible alternative for resource-limited environments. The primary goal was to determine whether smaller LLMs could achieve comparable results to their larger counterparts while maintaining performance quality.

As shown in Table 1, our experiments revealed a notable performance gap when using smaller LLMs like Mistral-7B-Instruct. There was a significant drop in accuracy and output quality compared to the larger models. These results highlight the current limitations of smaller LLMs in achieving parity with state-of-the-art models for complex tasks, suggesting that the trade-off between model size and performance remains a critical challenge.

```
<code>
use std::convert::TryInto;
…
</code>

This code does not compile. Here are some error
messages contained in <error-message> tags

<error-message>
error[E0689]: can't call method `wrapping_add` on
ambiguous numeric type `{integer}`
…
</error-message>

Fix the above compilation errors.

Here are some constraints contained in <list> tags
that you should respect:

<list>
1. Use only safe Rust.
2. Don't use raw pointers.
3. Use box pointer whenever possible. Box pointers are
preferable to other alternatives.
4. Try not to use Traits if possible. I would not like
to have Traits in resulting Rust code.
5. Try not to use Generics if possible.
</list>
```

(a) Prompt for translate if compile error exist

Human:

 # Preamble
You are given a C/Go program and its faulty Rust translation. We need to repair the faulty Rust program.

 # Code to be translated
{C/Go Program}

 # Code to be repaired
{Faulty Rust Program}

 # Instruction
Make changes to the given code to obtain expected outputs for the given test inputs.

 # Constraints
Here are some constraints that you should respect:
...

 # Counterexamples
CE1
CE1

Assistant:
{LLM generated rust translation}

Human:
That is incorrect on the following inputs:
Counterexamples
CE1
CE2

Assistant:

(b) Prompt for fixing the fuzz counter examples found

Table 1. Flourine compare between mistral-7b-instruct to other LLMs

2*	total fails		compile errrors		fuzz errors		counter examples		total success	
	libopenaptx	opl	libopenaptx	opl	libopenaptx	opl	libopenaptx	opl	libopenaptx	opl
gemini	24	51	8	10	13	38	3	3	7	30
gpt4	**18**	46	0	2	18	44	0	0	13	35
claude2	24	43	5	1	13	36	6	6	7	35
claude3	**18**	46	3	2	13	42	2	2	13	35
mixtral	24	51	6	12	7	21	11	18	7	30
Mistral-7b (RQ1)	31	63	12	15	15	45	4	3	0	20

This finding underscores the need for further research to bridge the gap by optimizing smaller LLMs or introducing novel techniques that can enhance their capabilities. By doing so, we aim to make powerful language models more accessible and practical for a wider range of applications.

RQ2: How does our new method improve existing method with small LLMs?

Our second research question focuses on understanding how to enhance the performance of Flourine by leveraging smaller large language models (LLMs) through multi-contextual integration. By incorporating a diverse range of contexts generated from different LLMs, we aim to address specific challenges in achieving accurate and efficient results.

In Table 2, we provide results using Mistral-7B-Instruct and Llama3-3B-Instruct models for conversational tasks. These two models were chosen for their complementary strengths and scalability, which allow us to

Table 2. Flourine compare to conversational agent translate

2*	total fails		compile errrors		fuzz errors		counter examples		total success	
	libopenaptx	opl	libopenaptx	opl	libopenaptx	opl	libopenaptx	opl	libopenaptx	opl
Mistral-7b (RQ1)	31	63	12	15	15	45	4	3	0	20
conversational agents (RQ2)	28	56	10	13	14	25	4	18	3	25

explore the potential of multi-model collaboration. Specifically, the integration of contexts from these two models demonstrates tangible improvements. For example, this approach translated to 3 additional successful examples when working with the libopenaptx benchmark and 5 examples from the opl benchmark.

This improvement highlights the effectiveness of multi-context synthesis, where the diversity of perspectives provided by smaller LLMs enhances problem-solving capabilities. The findings also suggest that our approach can optimize resource usage by utilizing more smaller models can comparable to larger LLMs, paving the way for favor more LLMs to improve translate task rather than using bigger LLMs.

5 Replication package

The following package will show you the artifacts of the flourine base and our result. For download codebase to replicate, please download the flourine artifact in advace in this link and follow README file. For produce result from our works, please download the new translate result and update on utils file in this link, and use function get_result and add parameter accordingly.

References

[1] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. arXiv:2405.11514 [cs.SE] https://arxiv.org/abs/2405.11514

[2] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. arXiv:2205.10583 [cs.SE] https://arxiv.org/abs/2205.10583