2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC) | 979-8-3503-1674-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICBC59979.2024.10634392

SolMover: Feasibility of Using LLMs for Translating Smart Contracts

Rabimba Karanjai*,Lei Xu†, Weidong Shi*
*University Of Houston, TX, USA
{rkaranjai, wshi3}@uh.edu
†Kent State University, OH, USA
xuleimath@gmail.com

***Abstract*—Large language models (LLMs) have showcased remarkable skills, rivaling or even exceeding human intelligence in certain areas. Their proficiency in translation is notable, as they may replicate the nuanced, preparatory steps of human translators for high-quality outcomes. Although there have been some notable work exploring using LLMs for code to code translation, there has not been one for smart contracts, especially when a target language is unseen to the LLM. In this work, we aim to introduce our novel framework SolMover consisting of two different LLMs working in tandem in a framework to understand coding concepts and then use that to translate code to an unseen language. We explore the human-like learning capability of LLMs in this paper with a detailed evaluation of the methodology to translate existing smart contracts from Solidity to a low-resource one called Move. Specifically, we enable one LLM to understand coding rules for the new language to generate a planning task, for the second LLM to follow, which does not have planning capability but does have coding. Experiments show that SolMOver brings significant improvement over gpt-3.5-turbo-1106 and outperforms both Palm2 and Mixtral-8x7B-Instruct. Our further analysis shows us that employing our bug mitigation technique even without the framework still improves code quality for all models.**

***Index Terms*—Smart Contracts, Machine Learning, Machine Translation, Code Transpilation, LLM, Large Language Model**

I. Introduction

Recent advancements in Large Language Models (LLMs) have significantly progressed toward human-like capabilities, particularly in natural language translation, underscoring their potential in code-to-code translation. This study investigates the feasibility of using LLMs for translating smart contracts, focusing on transpilation between Solidity and Move languages, with an emphasis on security. We introduce *SolMover*, a system that leverages multi-step knowledge distillation for translating Solidity code into Move code, incorporating fine-tuning on relevant language textbooks and specifications, and utilizing error-code-guided prompt engineering to enhance code quality. Our approach aims to explore the capabilities of LLMs in learning coding concepts, generating sub-tasks from generalized prompts, producing compilable code in low-resource languages, and mitigating bugs using compiler feedback.

The research questions addressed include:

RQ1: Can LLMs learn coding concepts? **RQ2**: Can the concepts be used to generate a granular subtask from a generalized prompt? **RQ3**: Can we generate code in a low-resource language the LLMs have not been trained on? **RQ4**: Mitigating bugs using compiler feedback. Our contributions are the development of *SolMover* for smart contract translation, demonstration of LLMs' ability to learn and apply coding concepts, the feasibility of generating compilable code in a low-resource language with limited training data, and the use of compiler errors to refine generated code. This work is the first to show the potential of using LLMs for code generation in low-resource languages through concept learning and fine-tuning.

II. Related Work

The related work broadly falls into two categories: code transpilers and LLM-based solutions.

Rule-based Transpilers. These function as precise interpreters between programming languages. They employ a set of defined rules for accurate code conversion (e.g., Python to Java [1]). While offering precision, rule-based transpilers may require manual adjustments for complex language features.

Statistical ML-based Transpilers. These leverage machine learning to identify patterns in large code translation datasets. This allows them to predict translations for new code instances [2], [3], [4].

Transformer and Other ML-based Tools. Transformer models excel in code translation tasks due to their success in natural language translation [5]. Specialized models like CodeBERT [6] and CodeGPT [7] focus on code understanding and generation. Variations like CodeT5 [8] and PLBART [9] offer enhanced precision and versatility.

LLM-based Methods Using Compiler/unit testing. Methods exist that employ unsupervised code translation with self-training and automated unit tests to validate equivalence [10]. Others leverage reinforcement learning with compiler feedback for improved code generation [11].

III. Design

In this section, we detail the SolMover framework succinctly.

As Fig. 1 illustrates, SolMover integrates multiple components. *Block 1* analyzes the Solidity contract to generate the initial prompt using techniques akin to Karanjai et al [12]. *Block 2* leverages a retrieval-augmented approach with Move

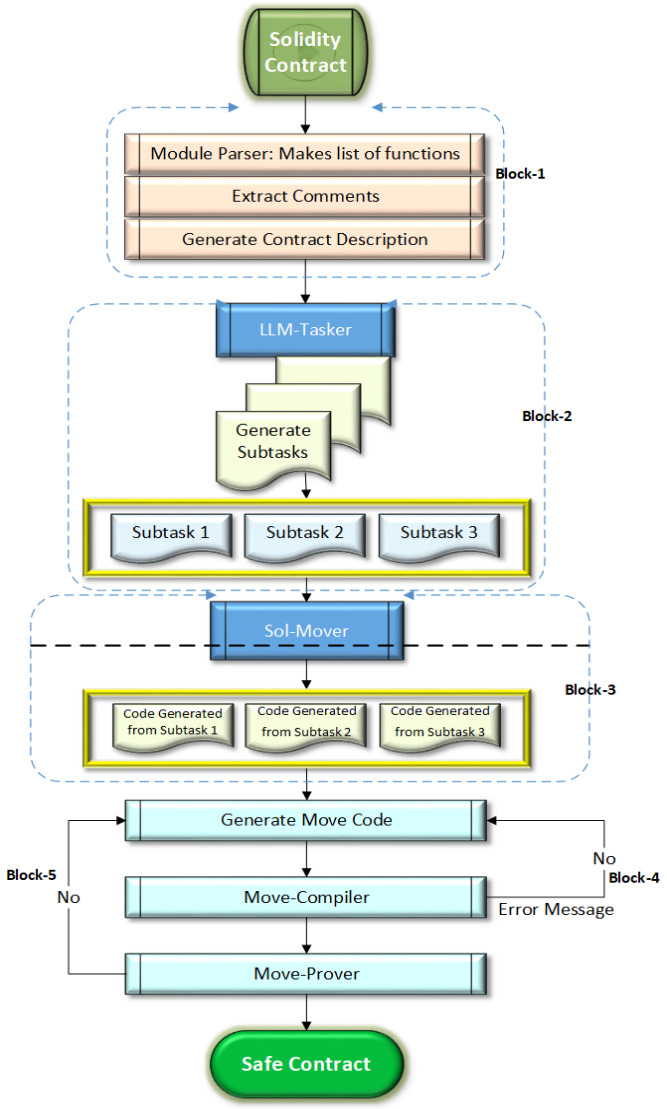

Fig. 1. Framework for **SolMover**. Five stages: (1) Task Creation: Parses Solidity files to generate tasks from comments and keywords. (2) Concept Mining: Uses retrieval-augmented methods to derive sub-tasks from programming guides. (3) Code Generation: Produces code for each sub-task, compiling them into a Move file. (4) Compilation: Compiles the code, returning errors for revision if necessary. (5) Verification: The Move Prover checks the code's provability, returning errors for corrections if needed.

literature to craft sub-tasks, which are refined and compiled into Move code in subsequent steps. The code undergoes compilation in *Block 4*, with failures triggering revisions. Finally, *Block 5* employs the Move Prover to verify code correctness, with iterative corrections for unproven code, aiming for a formally verified contract

IV. Results

Our experiments focused on four LLMs, with only gpt-3.5-turbo-1106 and SolMover producing tangible results, as evidenced by Table I.

LLMs	Compilable Code?	Performance
gpt-3.5-turbo-1106	Y	Mixed
Solmover (Two LLM Combined)	Y	Mixed
Mixtral-8x7B-Instruct	N	Mixes different languages and generate unusable code
LLama2	N	Mixes different languages and generate unusable code
Palm2	N	For Move only generates code snippets and plan

TABLE I
CODE TRANSLATION CAPABILITY OF FOUR LLMS.

LLM	Total Translation Task	Successful Compilation(SC)	SC After Error Feedback	SC after Move Prover Feedback
gpt-3.5-turbo-1106	734	204	229	229
Solmover (Two LLM Combined)	734	313	397	401

TABLE II
SUCCESSFUL TRANSLATIONS.

Subsequent results focus on gpt-3.5-turbo-1106 and SolMover.

A. Successful Smart Contract Translation

A dataset of 734 Solidity contracts underwent translation by both models, evaluating performance by successful compilation and improvements after feedback. Results are summarized in Table II.

Initial SC rates were 204 for gpt-3.5-turbo-1106 and 313 for Solmover. After feedback, SC rates improved to 229 and 397, respectively, with Solmover showing marginal gains post-Move Prover feedback to 401 SCs. This demonstrates Solmover's superior performance and adaptability in translating and refining smart contracts.

The findings underscore Solmover's enhanced learning and adaptability in complex code translation tasks, demonstrating its potential in smart contract development."

V. Conclusion

In this work, we have proposed SolMover, a composite two-LLM-based framework that encodes textbook knowledge in one to generate a granular subtask for the other to generate code based on that. We have evaluated the framework for the code translation task, involving an extremely low-resource code "Move" as a target language for smart contract creation. We have shown how our framework can take existing solidity smart contracts and translate them to generate move smart contracts. Our contributions include introducing a way to encode concepts into an LLM, and showing how it can help LLM translate code into a non-trained source language. We also evaluate our iterative compiler error feedback loop to show that it can help mitigate bugs in the translated code.

Acknowledgement

Part of this work was conducted with the generous grant support from Sui Foundation Academic Research Grant.

References

[1] "py2java: Python to Java Language Translator," https://pypi.org/project/py2java/.

[2] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical Statistical Machine Translation for Language Migration," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 651–654.

[3] S. Karaivanov, V. Raychev, and M. 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*, 2014, pp. 173–184.

[4] K. Aggarwal, M. Salameh, and A. Hindle, "Using Machine Translation for Converting Python 2 to Python 3 Code," PeerJ PrePrints, Tech. Rep., 2015.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[6] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[7] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," *arXiv preprint arXiv:2102.04664*, 2021.

[8] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[9] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified Pre-training for Program Understanding and Generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668. [Online]. Available: https://aclanthology.org/2021.naacl-main.211

[10] B. Roziere, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "TransCoder-ST: Leveraging Automated Unit Tests for Unsupervised Code Translation," in *International Conference on Learning Representations (ICLR)*, 2022. [Online]. Available: https://openreview.net/forum?id=cmt-6KtR4c4

[11] X. Wang, Y. Wang, Y. Wan, F. Mi, Y. Li, P. Zhou, J. Liu, H. Wu, X. Jiang, and Q. Liu, "Compilable Neural Code Generation with Compiler Feedback," in *Findings of the Association for Computational Linguistics: ACL 2022*, 2022, pp. 9–19.

[12] R. Karanjai, E. Li, L. Xu, and W. Shi, "Who is smarter? an empirical study of ai-based smart contract creation," in *2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, 2023, pp. 1–8.