Semantic Error Detection in Code Translation Using Knowledge-Driven Static Analysis with AI Chain

Lei Chen[1], Sai Zhang[1], Fangzhou Xu[1], Liang Wan[1] and Xiaowang Zhang[1,*]

[1]College of Intelligence and Computing, Tianjin University, Tianjin 300350, China

Abstract

In the task of code translation, neural network-based models frequently generate semantically incorrect code that deviates from the original logic of the source code. This problem persists even with advanced large models. While a recent approach suggests using test cases to identify these semantic errors, its effectiveness is highly dependent on the quality of the test cases, making it unsuitable for code snippets that lack test cases in real-world scenarios. To automatically locate semantic errors in code translation without valid test cases, we propose the Knowledge-guided Semantic Analysis Framework (KSAF). KSAF decomposes the source and translated code synchronously and performs static analysis to detect semantic errors. This is achieved by leveraging fine-grained knowledge in conjunction with an AI chain-driven Large Language Model (LLM). In a previously studied benchmark of Python programs, our framework based on the GPT-3.5-turbo model achieved a correctness rate of 47.8% through a static evaluation method. This result represents a 37.2% improvement over the baseline using the same base model and a 13.4% improvement in correctness compared to the baseline using the GPT-4-turbo-based model.

Keywords

Large Language Models, Semantic Mistakes, Knowledge Base, Code Translation

1. Introduction

Code translation involves converting a program written in one programming language into another, ensuring that the original functionality remains intact. Neural network models have achieved significant success in this task, but recent studies [1, 2] have found that these models often introduce subtle errors. These subtle errors can be grouped into syntactic and semantic errors. Syntax errors violate the syntax rules of destination languages, which a grammar checker can often identify. In contrast, semantic errors are more subtle and may result in translated code that either fails to execute without violating the target language's syntax or produces outputs that are inconsistent with the original code [3]. For example, as shown in the replace function in Figure 1, s.replace('-', ' ') in Python replaces all occurrences of '-' with ' ', while in JavaScript, it only replaces the first occurrence by default.

Based on this, Wang et al. [3] rely on test cases that can expose semantic errors to analyze code and locate these errors dynamically. However, their method is highly dependent on the quality of the test cases, requiring them to reveal semantic errors effectively, and it cannot

Posters, Demos, and Industry Tracks at ISWC 2024, November 13–15, 2024, Baltimore, USA

*Corresponding author.

2022244117@tju.edu.cn (L. Chen); zhang\protect\TU_sai@tju.edu.cn (S. Zhang); xu\protect\TU_fangzhou@tju.edu.cn (F. Xu); lwan@tju.edu.cn (L. Wan); xiaowangzhang@tju.edu.cn (X. Zhang)

handle code snippets lacking valid test cases. Additionally, In the code translation domain, relying on test cases to execute code is not only costly but also poses potential security risks [4].

To automatically locate semantic errors in code translation in the absence of valid test cases, we propose a framework KSAF, which decomposes the source code and translated code synchronously and statically analyses the code to locate semantic errors with fine-grained knowledge combined with AI chain-driven LLM. Experiments show that our approach can achieve better results. KSAF is the first method to locate semantic errors in code translation without test cases. It only requires API documentation, does not need model training, and is adaptable to low-resource languages.

2. Approach

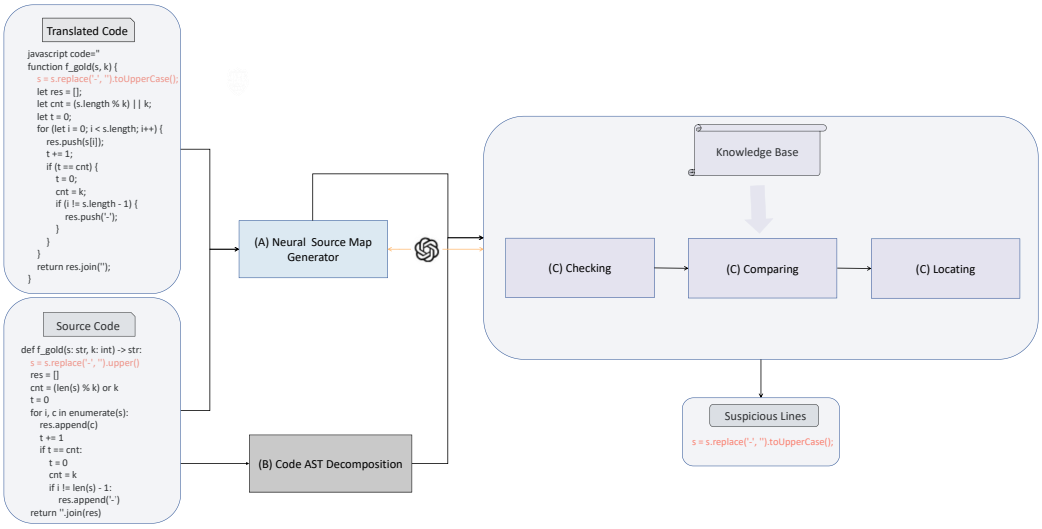

Figure 1: A static analysis framework based on the Large Language Model (LLM).

Figure 1 illustrates the general framework of our approach. We first build an API knowledge base by crawling the official JavaScript documentation [5]. Then, we design a framework based on the knowledge-driven AI chain and code decomposition to locate errors in code translation statically.

In this work, we collect API documents from the online resource [5] through the web crawler tool [6], where each API document is a crawled web page containing rich information such as the name of the API, syntax, parameters, samples, and function descriptions. We only keep half-structured API statements and functional descriptions, where an API statement describes the fully qualified name(FQN) of an API, which serves as a retrieval index for the knowledge base. The functional description contains the functional logic and behavior of that API.

In the Neural Source Map Generator module, as shown in Figure 1 A, the source code,

Table 1
Performance of baseline methods and KSAF on benchmarks.

Method	$\mathcal{S}_{sem}$	$\mathcal{S}_{sub\_sem}$	
		$\mathcal{S}_{hid}$	$\mathcal{S}_{dif}$
Few Shot +GPT-3.5-turbo	4.3%	3.1%	2.6%
CoT + GPT-3.5-turbo	10.6%	13.0%	14.0%
Few Shot + GPT-4-turbo-preview	32.1%	36.2%	34.5%
CoT + GPT-4-turbo-preview	34.4%	44.6%	45.1%
KSAF+GPT-3.5-turbo	**47.8%**	**46.4%**	**45.1%**

translated code, and a fixed prompt are input into the LLM. This module produces a mapping between atomic fragments in the source code and their corresponding parts in the translated code, providing an ordered list of these atomic fragments.

In the Code AST Decomposition module, as shown in Figure 1 B, the abstract syntax tree(AST) of the source code is traversed to extract "subtrees" from eight types of nodes as sub-code [7]. Using the mapping list from Module A, the corresponding translated code for each sub-code is obtained. Each sub-code pair is then passed to the next module.

After obtaining the sub-code and its corresponding translated code, KSAF uses LLM for static analysis to identify semantic inconsistencies between the source and translated code. We designed a knowledge-driven LLM AI Chain workflow, as shown in Figure 1 C, which includes three steps: Checking, Comparing, and Locating, all using the same LLM. In the Checking step, KSAF inputs the source code, translated code, and a fixed prompt into the LLM to extract the fully qualified names (FQN) of operators and APIs, then passes the results to the Comparing step. In the Comparing step, the FQNs are linked with an offline-built API knowledge base to obtain the corresponding API function descriptions. These descriptions and the results from the Checking step form a prompt fed into the LLM to precisely summarize the differences in operators and APIs between the source and translated code. In the Locating step, the Comparing step results, source code, and translated code are input into the LLM as a prompt to identify suspicious code lines that might cause semantic inconsistencies between the source and translated code.

3. Experiments

In this module, our objective is to compare the effectiveness of KSAF with other methods. To ensure fairness in the experiments, we selected methods that, like KSAF, do not require test cases for static code analysis. Specifically, we chose the widely recognized prompt-based methods that aim to fully leverage the potential of foundational models: LLMs with Few-Shot Learning [8]: A few examples are provided as demonstration examples in the prompt to guide the LLM in achieving better performance on the task. LLMs with Chain of Thought (CoT) [9]: By appending "Let's think step by step" at the end of the prompt, the LLM is prompted to explain the reasoning or steps before providing the final answer.

We used the dataset (excluding test cases) and metrics [3] of Wang et al. to evaluate our method and baseline. Where $\mathcal{S}_{sem}$, $\mathcal{S}_{hid}$, and $\mathcal{S}_{dif}$ denote the ratios of successfully identified errors to the total number of semantic errors, hidden errors, and errors leading to results

that differ from the source code output, respectively. Semantic errors are when the code is syntactically correct but logically flawed, causing the program to behave in a way that is not expected. Hidden errors are a special kind of semantic error, which usually can't be immediately localized to a specific fix, even when running test cases. Errors leading to results that differ from the source code output are also a type of semantic error, which does not cause a runtime error but causes the output of the translated code in unit tests to be inconsistent with the source code [3].

As shown in Table 1, our method outperforms all baseline approaches. Additionally, the method proposed by Wang et al. is unable to handle code without test cases, resulting in zero values for all metrics. And following the experimental setup of previous work [3], we found that our framework KSAF detected an average of 3.0 suspicious lines, which represents 16.5% of the total lines of code. This indicates that users typically need to review only 1 to 3 lines to understand and fix semantic errors.

4. Conclusion

This paper propose a method based on code AST decomposition and fine-grained knowledge combined with an AI chain-driven LLM to locate semantic inconsistencies between source and translated code. This method effectively handles code without test cases. We plan to extend our approach to multi-language datasets and conduct comprehensive experiments to further validate KSAF's effectiveness in the future.

Acknowledgments

This work was supported by the Project of Science and Technology Research and Development Plan of China Railway Corporation (N2023J044).

References

[1] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, R. Jabbarvand, Lost in translation: A study of bugs introduced by large language models while translating code, in: 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), IEEE Computer Society, 2024, pp. 866–866.

[2] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, R. Sharma, Jigsaw: Large language models meet program synthesis, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1219–1231.

[3] B. Wang, R. Li, M. Li, P. Saxena, Transmap: Pinpointing mistakes in neural code translation, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 999–1011.

[4] W. Yan, Y. Tian, Y. Li, Q. Chen, W. Wang, Codetransocean: A comprehensive multilingual benchmark for code translation, in: Findings of the Association for Computational Linguistics: EMNLP 2023, 2023, pp. 5067–5089.

[5] Mozilla, Javascript reference, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference, 2024. Last accessed: March 11, 2024.
[6] beautiful-soup 4, Beautiful soup 4, 2024. https://beautiful-soup-4.readthedocs.io/en/latest/.
[7] T. Hu, Z. Xu, Y. Fang, Y. Wu, B. Yuan, D. Zou, H. Jin, Fine-grained code clone detection with block-based splitting of abstract syntax tree, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 89–100.
[8] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Advances in neural information processing systems 33 (2020) 1877–1901.
[9] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, Y. Iwasawa, Large language models are zero-shot reasoners, Advances in neural information processing systems 35 (2022) 22199–22213.