The Effects of Semantic Information on LLM-based Program Repair

Shota Hori[1] Shinsuke Matsumoto[1] Yoshiki Higo[1] Shinji Kusumoto[1] Kazuya Yasuda[2] Shinji Ito[2] Phan Thi Thanh Huyen[2]

[1] Graduate School of Information Science and Technology, Osaka University, Japan
[2] Hitachi, Ltd., Japan

Abstract. Large Language Model-based Automated Program Repair (LLM-APR) has recently received significant attention as a debugging assistance. Our objective is to improve the performance of LLM-APR. In this study, we focus on semantic information contained in the source code. Semantic information refers to elements used by the programmer to understand the source code, which does not contribute to compilation or execution. We picked out specification, method names and variable names as semantic information. In the investigation, we prepared eight prompts, each consisting of all combinations of three types of semantic information. The experimental results showed that all semantic information improves the performance of LLM-APR, and variable names are particularly significant.

Keywords: Large language model (LLM), automated program repair (APR), semantic information, prompt engineering, ChatGPT

1 Introduction

Program repair using a large language model (LLM) has recently received significant attention as a debugging assistance [6, 8, 10]. LLM, a language model trained on a substantial corpus, can be applied to various natural language tasks. Since several LLMs include source code as training data, they can be applied to various programming tasks. One of the applications is LLM-based automated program repair (LLM-APR). Most existing APR studies [2, 4] have considered bug fixation as a search problem based on fault localization and source code modification. On the other hand, LLM-APR repairs buggy source code through learning and inference based on massive amounts of data. LLM-APR has strong advantages of natural language interaction and higher repairing performance compared to the existing APR techniques [10].

LLM-APR has the possibility of performance improvement because LLM-APR studies are in the early stages. Sobania et al. have experimentally investigated the reparing performance of LLM-APR with ChatGPT. Their experiments showed ChatGPT can fix 19 out of 40 bugs [8]. One concern is that their prompt contains buggy source code and minimal instruction such as *"Does this program have a bug? How to fix it?"*. Several studies pointed out that prompt design has

a significant impact on generative AI performance [9]. Therefore, prompt design, such as instruction sentences and information in the source code, could be an essential factor for LLM-APR.

This paper focuses on semantic information contained in the source code. We refer to semantic information as a textual notation that programmers use to understand the behavior of source code but does not contribute to compilation or execution. One of the most crucial semantic information is a specification written as JavaDoc in Java or Docstring in Python. Furthermore, method names can be regarded as semantic information that helps developers grasp the method's brief responsibility. Variable names also tell us the meaning of the data. While various other types of semantic information are considered, this paper focuses on these three as semantic information. It is known that this semantic information significantly helps source code comprehension [7]. Our key question is *whether semantic information also helps LLM to understand source code for program repair, as it does programmers?*

This study aims to improve the program repairing performance of LLM-APR. To this end, we investigate the effects of semantic information on LLM-APR. In the experiment, we prepared eight prompt patterns, each consisting of all combinations of three types of semantic information: specification, method name, and variable name. We compared the test pass rate of LLM-APR with each prompt for two types of bugs. The experimental results showed that all semantic information improves the performance of LLM-APR, and variable names are particularly significant. These results suggest that LLM may also be able to understand source code from semantic information, similar to understand source code by programmers.

2 Related Works

Accompanying the advent of generative AI such as ChatGPT, research is conducted to look into the feasibility of LLM-APR. Sobania et al. investigated the program repair capability of LLM-APR using ChatGPT [8]. They employed QuixBugs, a bug dataset obtained from programming contests, and reported 19 successful bug fixes for 40 bugs. For a larger-scale study, Xia et al. investigated the performance of LLM-APR on multiple LLM models and bug datasets [10]. Consequently, LLM-APR successfully fixed 37 bugs out of 40 in QuixBugs. According to another study with traditional APR methods without LLM, integrating ten APR methods resulted in 16 successful for QuixBugs. Therefore, the performance of LLM-APR has already exceeded that of existing APR methods.

In addition, it is known that prompt design has a significant impact on the performance of LLM [5, 6, 9]. Parasaram et al. investigated the impact of seven facts, such as buggy code's context and GitHub issues, on LLM-APR [6]. They reported each fact aids the performance of LLM-APR. However, it was also revealed that LLM-APR prompts are non-monotonic over facts: adding more facts may degrade the performance of LLM-APR. Therefore, a significant research question in prompt engineering for LLM-APR is: *What specific informa-*

tion, and in what quantity, should be integrated into the prompt to maximize the performance of LLM-APR?

Source code is highly flexible and can be written in various expressions to describe the same content, like natural language context. For example, method names and variable names are elements that strongly depend on the thoughts and habits of programmers [1]. These identifier names are known to greatly help in understanding source code [7]. In addition, specification in JavaDoc helps understand the behavior of a method. However, it is not clear whether specification and identifier names also contribute to LLM-APR, or what specification and identifier names are helpful for understanding the source code of LLM.

3 Experimental Design

3.1 LLM Prompt Patterns

In our experiment, the basic structure of a prompt was a pair of instruction context and buggy source code. The instruction context included that the source code contains a bug and that the task is to fix the bug, like *"This method contains a bug. Please fix it."* Therefore, the usage scenario was where the developer is aware of the existence of a bug but does not know how to fix it.

We prepared eight prompts, which are combinations of the presence or absence of three types of semantic information. In the following section, these eight prompts are called prompt patterns and are expressed in the form of P`-/-/-`. For example, P`s/m/v` represents a prompt with specification, the method name, and variable names, while P`s/-/v` represents a prompt with specification and variable names but without the method name. It should be noted that the granularity of the source code included in the prompt was a method, and the JavaDoc for each method was used as specification.

Concrete examples of P`s/m/v` and P`-/-/-` are shown in Fig. 1 and Fig. 2, respectively. From Fig. 1, it is found that P`s/m/v` has the task context, specification, and buggy method. We can infer the ideal behavior of the target method from each semantic information. By contrast, Fig. 2 shows that P`-/-/-` has the task context and a buggy method without appropriate method and variable names. This available information does not allow the inference of the ideal behavior of the target method. Thus, it is found that semantic information contributes to the understanding of source code by programmers. However, it is unclear whether this also helps LLM understand source code. In a practical program repair situation, the source code would never be represented as Fig. 2, but we prepared this prompt and conducted experiments for our research objective. Besides, in prompt patterns with masked method or variable names, an instruction context such as *"do not change method name and variable names ..."* was added. This intent is that the identifier names will not be converted when LLM repairs the program. This is to restore masked identifier names to their original project state and execute the test when evaluating the success or failure of LLM-APR (described below).

```
<task>
This method follows the next specification.
But the method contains a bug. Please fix it.
</task>

<specification>
Remove an attribute by key. <b>Case sensitive.</b>
@param key attribute key to remove
</specification>

<method>
public void remove(String key) {
   int i = indexOfKey(key);
   if (i == NotFound)
       remove(i);
}
</method>
```

Fig. 1. An example of prompt pattern with all semantic information "P`s/m/v`"

```
<task>
This method contains a bug. Please fix it.
In addition, do not change method names and
variable names such as "$1" when fixing it.
</task>

<method>
public void $1(String $2) {
   int $3 = indexOfKey($2);
   if ($3 == $4)
       $1($3);
}
</method>
```

Fig. 2. An example of prompt pattern without all semantic information "P`-/-/-`"

3.2 Dataset

Two types of bug datasets were used: a relatively large artificial bug dataset and a real bug dataset. Artificial bugs can be experimented on a large scale, but they are different from bugs that actually occur. Therefore, it is necessary to investigate whether the results of artificial bugs are also applicable to real bugs, so we prepared two types of datasets.

Artificial bug dataset: Artificial bugs were prepared using Mutanerator[3]. Mutanerator is a mutant generation tool for Java programs. In this study, we applied mutant operators described in Table 1. As targets for the experiment, we selected two Java projects: Jsoup[4] and Gson[5]. Both projects have more than 10,000 stars on GitHub, and their quality is high. Therefore, it is considered that they are assigned appropriate semantic information. Furthermore, we picked out methods that have non-zero coverage and have JavaDoc. This is to ensure validity

[3] https://github.com/kusumotolab/Mutanerator (accessed April 7, 2024)
[4] https://github.com/jhy/jsoup (accessed April 20, 2024)
[5] https://github.com/google/gson (accessed April 26, 2024)

Table 1. Mutation operators

Mutation operators	Conversion example	
	Before	After
Conditional boundary	a<b	a<=b
Increments	a++	a−−
Invert negatives	−1	1
Math	a+b	a−b
Negate conditionals	a==b	a!=b

through the test evaluation (described below) and to use JavaDoc directly as a method specification. When Mutanerator was applied to methods that met these requirements, 557 bugs were collected for Jsoup and 383 for Gson.

Real bug dataset: We used Defects4J [3] as real bugs. Defects4J is a dataset of bugs that occurred during real development process of Java projects. We selected Math project within Defects4J due to the high number of bugs present. Furthermore, we picked out bugs whose fixes are within one method, which has a JavaDoc. This is because the source code in the prompt was a single buggy method and used JavaDoc directly as a method specification. We collected 52 bugs that met these requirements.

3.3 LLM Model

We used gpt-3.5-turbo-0125. It is a text input/output model and is capable of handling both natural language and source code tasks. Although we set the upper limit of the number of tokens to 4096, this was not a problem because the upper limit was never exceeded. The temperature parameter of the model sets the randomness. A lower temperature means the model is likely to select tokens with higher likelihood, resulting in more similar samples. We employed temperature of 0 to reduce the randomness of the LLM output and ensure reproducibility.

3.4 Evaluation

To evaluate the LLM-APR performance, we used the number of successful repairs for each prompt pattern. Generative AI has randomness, such that it will output differently even for the same prompts [5]. For a countermeasure, Sobania et al. [8] conducted multiple attempts. Therefore, we also conducted three attempts at the same prompt. Following the study by Sobania et al. [8], if any one of the three outputs succeeds in fixing, the bug is regarded as successfully fixed.

We used test results to determine the success or failure of LLM-APR. In artificial bugs, we regarded the repair as successful if none of the test cases for each project failed. In real bugs, we regarded the repair as successful if the test command provided in the Defects4J succeeded. Initially, masked identifier names in the LLM output were restored to their original project state, This is because the output had identifier names such as '$1', and the test cannot be run. Next, we merged them back into the original source code and run the test.

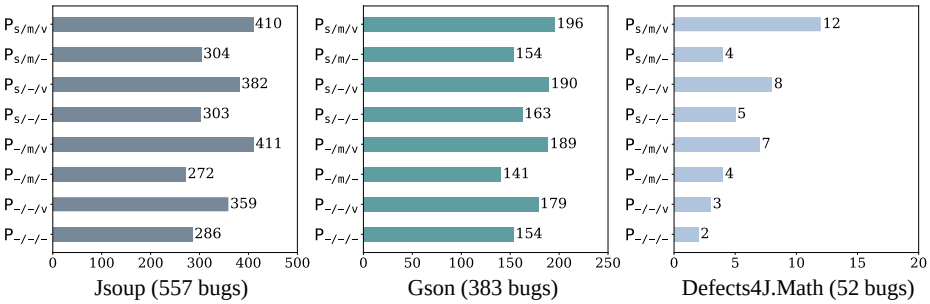

Fig. 3. The number of successful fixes for each bug dataset

4 Results

Fig. 3 shows the number of successful fixes of LLM-APR for each bug dataset. In the following, the number of successes in P`s/m/v` is represented by #P`s/m/v`. For example, in Jsoup, we write #P`s/m/v`=410.

4.1 Artificial Bugs

Jsoup: First, since #P`s/m/v`=410 and #P`-/m/v`=411, it was found that specification has few effects on the performance of LLM-APR when appropriate identifier names are assigned in the source code. On the other hand, other pairs with and without specification were #P`s/-/v`=382, #P`-/-/v`=359 and #P`s/m/-`=304, #P`-/m/-`=272. Therefore, specification improves the performance when appropriate identifier names are not assigned. Next, as #P`s/m/v`=410, #P`s/m/-`=304 and, #P`-/m/v`=411, #P`-/m/-`=272, it showed that having variable names increased the number of successes by more than 100 in both cases. Therefore, it can be found that variable names significantly improve the performance of LLM-APR. For other pairs about variable names, we could also see a trend of performance improvement. Finally, since #P`s/m/v`=410, #P`s/-/v`=382 and #P`-/m/v`=411, #P`-/-/v`=359, having the method name increased the number of successes. However, the difference is smaller than variable names, so we can consider that the effects on performance are minor compared to variable names. Additionally, since #P`s/m/-`=304, #P`s/-/-`=303 and #P`-/m/-`=272, #P`-/-/-`=286, this suggests that method names contribute little to the performance when the appropriate variable names do not exist.

Gson: Firstly, since #P`s/m/v`=196 and #P`-/m/v`=189, specification slightly increased the number of successes. For other pairs about specification, results also became better with specification. Therefore, as in the case of Jsoup, specification seems to aid to the performance of LLM-APR. Next, #P`s/m/v`=196, #P`s/m/-`=154 and #P`-/m/v`=189, #P`-/m/-`=141, so it indicated that variable names increased the number of successes. Additionally, for other pairs about variable names, the performance improved. Thus, as Jsoup, variable names significantly improve the performance of LLM-APR. Finally, since #P`s/m/v`=196,

#Ps/-/v=190 and #P-/m/v=189, #P-/-/v=179, in these two cases, the number of successes was larger than with the method name. However, other pairs about method name were #Ps/m/-=154, #Ps/-/-=163 and #P-/m/-=141, #P-/-/-=154. Therefore, this is also similar to Jsoup, the method name does not contribute to LLM-APR when the appropriate variable names do not exist.

4.2 Real Bugs

Defects4J.Math: First, since #Ps/m/v=12 and #P-/m/v=7, this indicated that specification improved the performance of LLM-APR. For the other pairs about specification, #Ps/-/v=8, #P-/-/v=3 and #Ps/-/-=5, #P-/-/-=2, we could see that the performance improvement. Considering that the total number of bugs is 52, the performance improvement due to specification is more pronounced than in the case of artificial bugs. Next, #Ps/m/v=12, #Ps/m/-=4 and #P-/m/v=7, #P-/m/-=4, so variable names increased the number of successes. For the other pairs about variable names, we could also see the same trend. In terms of the rate of successful repairs, the performance improvement due to variable nemes is also more pronounced than artificial bugs. Finally, as #Ps/m/v=12, #Ps/-/v=8 and #P-/m/v=7, #P-/-/v=3, method name also contributed performance improvement. However, the pair of #Ps/m/-=4 and #Ps/-/-=5 showed that did not aid performance. This trend is also observed in artificial bugs, and the investigation of this cause is a subject for future work.

Our experimental results indicated that in real bugs, semantic information contributes more to the performance of LLM-APR than artificial bugs. One possible reason is that LLM can easily infer artificial bugs from the structural characteristics of the source code. For example, assume a part of the artificial bug source code such as "`for(int i=0; i<n; i--){...}`". In this case, regardless of whether variable names are present or not, it can be inferred that there is a high possibility of a bug in either "`i<n`" or "`i--`". This is because this conditional expression would likely lead to an infinite loop. By contrast, it is difficult to infer real bugs from the structural characteristics of the source code. Most of the bug fixes in Defects4J.Math involved adding new conditional branches, calling methods defined outside of the target method, changing calculations, and so on. It is challenging for even developers to speculate on how to fix those bugs without knowing the ideal behavior of the method. This is also true for LLM, and it is difficult to repair real bugs without relying on semantic information.

5 Threats to Validity

Various prompting strategies, such as Chain of Thought, might affect the results. We conducted the experiment in the simplest method, but this is a future research topic. In addition, the training data for gpt-3.5-turbo-0125 is not public, so Jsoup and Gson might be included in the training data. This data leakage might lead to overestimating the accuracy of LLM-APR. However, even with Ps/m/v, Jsoup and Gson respectively failed to repair about 27% and 49%. This means that not all of the results of this paper are due to the data leakage.

6 Conclusion and Future Work

In this study, we investigated the effects of semantic information on LLM-APR. We selected specification, method names, and variable names as semantic information. The experimental results showed that all semantic information improves the performance of LLM-APR, and variable names are particularly significant.

As future work, we consider to investigate what identifier name is effective to LLM-APR performance. Schankin et al. investigated the characteristics of identifier names that help developers better understand source code [7]. They reported that developers understand source code more quickly with long identifier names that have more explanation than with short identifier names. We will check which identifier names are valid for LLM-APR performance. This future work will provide more practically useful results for LLM-APR.

Acknowledgements This research was partially supported by JSPS KAKENHI Japan (JP24H00692, JP21H04877, JP21K18302, JP23K24823, JP22K11985, 21K11829)

References

1. Alsuhaibani, R.S., Newman, C.D., Decker, M.J., Collard, M.L., Maletic, J.I.: On the naming of methods: A survey of professional developers. In: Proceedings of International Conference on Software Engineering. pp. 587–599 (2021)
2. Higo, Y., Matsumoto, S., Arima, R., Tanikado, A., Naitou, K., Matsumoto, J., Tomida, Y., Kusumoto, S.: kGenProg: A high-performance, high-extensibility and high-portability apr system. In: Proceedings of Asia-Pacific Software Engineering Conference. pp. 697–698 (2018)
3. Just, R., Jalali, D., Ernst, M.D.: Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of International Symposium on Software Testing and Analysis. pp. 437–440 (2014)
4. Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: TBar: Revisiting template-based automated program repair. In: Proceedings of International Symposium on Software Testing and Analysis. pp. 31–42 (2019)
5. OuYang, S., Zhang, J., Harman, M., Wang, M.: LLM is like a box of chocolates: the non-determinism of chatgpt in code generation. ArXiv p. arXiv:2308.02828 (2023)
6. Parasaram, N., Yan, H., Yang, B., Flahy, Z., Qudsi, A., Ziaber, D., Barr, E., Mechtaev, S.: The fact selection problem in llm-based program repair. ArXiv p. arXiv:2404.05520 (2024)
7. Schankin, A., Berger, A., Holt, D.V., Hofmeister, J.C., Riedel, T., Beigl, M.: Descriptive compound identifier names improve source code comprehension. In: Proceedings of Conference on Program Comprehension. pp. 31–40 (2018)
8. Sobania, D., Briesch, M., Hanna, C., Petke, J.: An analysis of automatic bug fixing performance of chatgpt. In: Proceedings of International Workshop on Automated Program Repair. pp. 23–30 (2023)
9. White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C.: A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv p. arXiv:2302.11382 (2023)
10. Xia, C.S., Wei, Y., Zhang, L.: Automated program repair in the era of large pre-trained language models. In: Proceedings of International Conference on Software Engineering. pp. 1482–1494 (2023)