Leveraging Conversational AI for Accelerating User-Driven Software Testing

Aminata Sabané[1,2], Laura Plein[3], and Tegawendé F. Bissyandé[1,2,3](✉)

[1] CITADEL Interdisciplinary Excellence Centre in Artificial Intelligence for Development, Ouagadougou, Burkina Faso
aminata.sabane@ujkz.bf, tegawende.bissyande@citadel.bf
[2] Université Joseph Ki-Zerbo, Ouagadougou, Burkina Faso
[3] SnT, Université du Luxembourg, Luxembourg, Luxembourg
laura.plein@men.lu

Abstract. This work addresses a research challenge in automating the translation of natural language inputs into programming language specifications. We consider the case of bug reports, which are informally written by users, and that must be specifying into executable test cases for reproducing the bug on the target software. Software bugs are indeed largely reported in natural language by users. Yet, we lack reliable tools to automatically address reported bugs (i.e., enabling their analysis, reproduction, and bug fixing). We therefore build on the recent promises brought by ChatGPT for various tasks, including in software engineering, and establish the following research question: *What if Conversational Artificial Intelligence (AI) models could be used to explore the semantics of bug reports as well as to automate their reproduction?* We evaluate the capabilities of ChatGPT, a state-of-the-art conversational AI, i.e., chatbot, using the popular Defects4J benchmark with its associated bug reports. The results reveal that ChatGPT can generate executable test cases that could trigger 50% of the bugs reported in natural language. These results are promising not only for the research community, but also for practitioners.

Keywords: ChatGPT · Debugging · Translation · Test cases · Bug reports

1 Introduction

Software users are expected to provide feedback on their experience in running programs. Such feedback often leads to various improvements by developers responding to feature requests and bug reports. In this respect, development platforms, such as GitHub, offer tool support for collecting reports and continuously monitoring how developers address them. Unfortunately, various studies have shown that bug reports are under-exploited [1]. Recurrently, indeed, researchers and practitioners point to the general quality of such reports: developers put much effort to "understand" and reproduce the potential bugs that are reported; researchers struggle to build tools for automatically capturing the semantics of the natural language text and transforming them into actionable inputs for existing (testing) frameworks.

A. Sere et al. (Eds.): AFRICOMM 2023, LNICST 588, pp. 81–88, 2025.
https://doi.org/10.1007/978-3-031-81573-7_6

With recent advances in natural language processing techniques, such as the advent of large language models (LLMs), a wide range of tasks have seen machine learning achieve, or even exceed, human performance. Machine translation [2, 3], in particular, has been a very active field where several case studies have been explored beyond language translation. For example, in software engineering, several research directions have investigated the feasibility of leveraging natural language inputs for producing programming artefacts and vice-versa. Some milestones have been recorded in the literature in code summarization [4, 5], program repair [6, 7], and even program synthesis [8]. Nevertheless, bug reports have scarcely been explored. Yet, automating bug reproduction via analysis of bug reports holds tremendous value. In this work, we propose to *study the feasibility of exploiting an LLM for reproducing bugs*. We focus on ChatGPT, which has recently received much attention and presents the advantage that its model has been trained on a large corpus of natural language text as well as source code of software programs.

But *can ChatGPT understand bug reports*? We consider the management of bug reports as an example case where machine learning can be helpful while keeping the human in the loop. "Understanding bug reports" suggests the eventual possibility of reproducing the reported bug. Our prompt is therefore focused on requesting ChatGPT to exploit a bug report's textual content (in natural language) and generate a formal test case (in a programming language). We assume that if ChatGPT can generate a test case that not only is executable but also fails on the associated buggy program version, then ChatGPT may have "understood" (in the sense of "*captured the semantics of unwanted execution behavior reported by the user*"). This assumption is, obviously, an over-approximation of the relevance of the generated test case since the generated failing test case may be based on random inputs that are irrelevant to the reported bug. Nevertheless, it would constitute a first milestone towards automatic test case generation based on user inputs, which reflects realistic and complex user experience.

2 Related Work

To address a bug, the first step is to understand it and reproduce it. To demonstrate that there is a bug, the developer has to write a bug-triggering test case since the original test suites are usually scarce and incomplete [9, 10]. Several techniques [11–14] have been developed to help developers with this very time-consuming process. However, these techniques mostly rely on formal specifications.

Recently, some work on test case generation using large language models (LLM) was done by [15] but their TestPilot still requires the functions signature and implementation as prompt. The use of ChatGPT to directly enhance Automated Program Repair (APR) techniques [16–18] highlights even more the potential of this new LLM. Feng et al. [19] have investigated ChatGPT's ability to help developers reproduce the bug while extracting important steps to reproduce the bug from the bug report. However they did not perform any test case generation. Additionally, they still require a human to actually reproduce the bug. In contrast, in our study, we are aiming at using the unprocessed human written bug report as direct input for test case generation, enabling automatic bug reproduction with only a human-in-the-loop to report the bug but not to address it.

3 Experimental Setup

In this section we overview the settings under which we assess the capability of a Large Language Model to translate informal bug reports from real software projects into formal test case specifications that reproduce the buggy behavior. In particular, we present the benchmark, the metrics as well as the experimental design.

3.1 Benchmark

We consider the **Defects4** repository [20], which includes real-world faults from various Java software development projects as enumerated in Table 1. We collect the bug reports associated to these faults. One must mention that not all bug reports were available, and some bugs referred to the same bug report, in that case it was only considered once to avoid bias in the results because of duplicates. We considered Defects4J due to its wide adoption in the software testing and software research communities.

Table 1. Java Projects from Defects4J used in the study.

	Projects					
	Chart	Cli	Closure	Lang	Math	Time
#Faults	26	39	174	64	106	29
#Faults associated to Bug Reports	6	30	127	60	100	19

3.2 Metrics

As introduced, the goal of the study is to assess whether ChatGPT can generate test cases from bug reports. Therefore, our evaluation is focused on measuring the quality of the generated test cases. We thus consider two main metrics:

- ***Executability***: a ChatGPT-generated test case may not even be syntactically correct to be compiled and executed. While often, the generated test case can be made executable after manually implementing small edits (e.g., adding relevant imports), we conservatively consider that executability is a binary metric, and is automatically computed once ChatGPT outputs are yielded (with no manual changes added).
- ***Validity***: an executable test case may or may not fail on the target buggy program. We follow the convention of patch validation in program repair and consider the generated test case to be valid only when it, indeed, fails on the buggy program. Otherwise it is considered as invalid.

3.3 Experimental Design

We rely on the ChatGPT API (version 3.5) for our experiments. We construct the prompt by concatenating two pieces of information: the instruction and the bug report.

The instruction is unique for all queries to ChatGPT and is as simple as follows: "{*write a Java test case for the following bug report:*}". For the bug report, our feasibility study considers that no pre-processing should be applied on the bug report, and the information should not include follow-up comments or attachments. For every prompt, we request the ChatGPT five (5) times and assess the different generated test cases. In practice, before running the generated test cases, the ChatGPT outputs are parsed to clean them from natural language texts (e.g., explanations) that would lead to compilation failures. Afterwards, the test cases are systematically included in the test suite, which is fully executed by the Defects4J test pipeline. Execution results are then logged, allowing us to compute the metrics of executability and validity.

4 Results

Figure 1 provides an illustrative example of a bug report (from the CLI project) and the associated formal test cases (ground truth in Defects4J and generated from ChatGPT). As we can see in this example, ChatGPT is able to reproduce a formal test case from a bug report, which can enable various software automation tasks, such as spectrum-based fault localization, patch validation in program repair, and more generally automated software testing.

Fig. 1. Example of bug report and associated test cases - Cli 17 from Defects4J

On the Defects4J dataset, we compute the proportion of bug reports for which ChatGPT is able to successfully generate test cases.

Table 2 summarizes the metrics. On average, executable test cases were obtained for 50% of the bug reports across all projects. The validity of the generated test cases varies greatly from one project to another, which can be explained by the different source and format of user-written bug reports. Unfortunately, a commonly known issue

of some Math project versions is that the test suite is compiling, but their execution is never-ending. Therefore, there were 10 Math bug reports for which we could not determine the *validity* of the generated test cases, which explains the lower validity results for this project. Overall, we got valid test cases for 30% of the bug reports, which is really promising. Among the executable test cases, we observed that 59% were valid. This shows that once the initial *executability* challenge is passed, the tests generated by ChatGPT are in fact valid, highlighting its understanding. Meaning that ChatGPT was able to understand the semantics of the user written bug report and translate them into a bug-triggering test case. These results highly motivate further research in this domain.

Table 2. Generation performance for ChatGPT: percentage of bug reports where we successfully generated at least one test case.

		Percentage of generation success		
Project	# of bug reports	Overall executability	Overall validity	Validity among executable
Chart	6	33%	17%	50%
Cli	30	53%	37%	69%
Closure	127	46%	28%	59%
Lang	60	60%	43%	72%
Math	100	43%	15%	35%
Time	19	84%	68%	81%
Total	342	50%	30%	59%

Further manual investigations also highlighted that *executability* and *validity* can in most cases be fixed with minor modifications (e.g., adding relevant imports or changing duplicated function names).

Findings: The experimental results based on ChatGPT APIs show that a large language model (LLM) can take bug reports as inputs and produce test cases that are executable in 50% of cases. Beyond *executability*, about 30% of the bugs could be reproduced with valid test cases. Specifically, over half (59%) of the executable test cases were valid test cases.

These results, which are based on an off-the-shelf LLM as-a-service, show promises for automated test case generation, beyond unit testing, leveraging complex information from user-reported bugs.

5 Discussion

Overall, our empirical study validates the hypothesis that ChatGPT can "understand" bug reports: given a bug report, it can extract its semantics and translate them into formal test cases. However, some challenges that should be addressed in the future remain. This

section will cover some limitations and suggest some research directions for future work to increase the amount of executable and valid test cases.

5.1 Threats to Validity

In this study, ChatGPT, an openly available LLMs known to have been trained on public data is exploited to generate test cases for bugs of the Defects4J dataset. As Defects4J is a standard benchmark that is publicly available, it is likely that parts of the dataset have been part of the model's training data, this is a threat to the validity of the results as it reflects a data leakage problem. To address this concern, we performed manual investigations of the generated test cases to ensure their difference with the original tests, confirming ChatGPT's capability of understanding the semantics of the bug report. Still, in further iterations of this work we will assess ChatGPT's performance on newly reported faults. Additionally, due to the randomness of ChatGPT, it is essential to verify that ChatGPT is correctly replying to the given prompt before starting the experiments.

5.2 Limitations & Future Work

In this study the *Executability* only reflects if a generated test case is directly executable or not. This doesn't reflect the amount of effort for a human to make it executable. After manually reviewing the generated test cases, it has been shown that most can be made executable through the modification of one or two lines of code. The most common issues are usually: missing imports, duplicate function names, or the use of a deprecated function. Those limitations could systematically be fixed in future work (e.g. with prompt engineering) significantly increasing the amount of executable test cases.

For future iterations of this study we suggest to investigate the potential of ***fine-tuning LLMs*** to translate the informal bug reports into formal test cases with a higher rate of executable and valid test cases.

In our experiments ***bug reports*** where collected and directly used as prompt to demonstrate the feasibility and applicability of our idea to address real software faults reported by the users. Nevertheless, pre-processing the textual data might be beneficial to keep ChatGPT focused on the main context of the bug report, therefore increasing the executability and validity of the generated test case.

To fix a bug, the first step is to reproduce it with a bug-triggering test case which has been proven feasible in our study. To reach to directly fix a newly reported bug, further research should be made on how we can improve and automatically deploy APR tools to not just automatically address the bug but fix it.

6 Conclusion

Large language models have recently gained substantial popularity, thanks to the public release of ChatGPT, whose potential as a disruptive technology has been largely advertised. The literature has empirically studied various capabilities of the inner language model for various natural language processing tasks. In this work, we investigate the feasibility of leveraging the inner language of the GPT model to translate informal bug

reports into formal test case specifications. The promising results, in terms of *executability* (i.e., the test case is syntactically correct), and {*validity*} (i.e., the test case actually makes the program fail), suggest that ChatGPT can " *understand*" the semantics of bug reports. This finding is essential as it opens new research directions with large language models, towards automating test case generation with a human in the loop (for writing bug reports).

Acknowledgments. This work was conducted as part of the Artificial Intelligence for Development in Africa (AI4D Africa) program, with the financial support of Canada's International Development Research Centre (IDRC) and the Swedish International Development Cooperation Agency (Sida).

References

1. Bissyandé, T.F., et al.: IEEE 24th international symposium on software reliability engineering (ISSRE). IEEE **2013**, 188–197 (2013)
2. Lopez, A.: Statistical machine translation. ACM Computing Surveys (CSUR) **40**, 1–49 (2008)
3. Stahlberg, F.: Neural machine translation: A review. Journal of Artificial Intelligence Research **69**, 343–418 (2020)
4. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR) **51**, 1–37 (2018)
5. Hu, X., Li, G., Xia, X., Lo, D. Jin, Z.: Deep code comment generation, In: Proceedings of the 26th conference on program comprehension, pp. 200–210. (2018)
6. Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**, 56–65 (2019)
7. Monperrus, M.: Automatic software repair: A bibliography. ACM Computing Surveys (CSUR) **51**, 1–24 (2018)
8. Gulwani, S., Polozov, O., Singh, R., et al.: Program synthesis, Foundations and Trends®. Programming Languages **4**, 1–119 (2017)
9. Le, X.-B.D., Bao, L., Lo, D., Xia, X., Li, S., Pasareanu, C.: On reliability of patch correctness assessment, In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp. 524–535. (2019)
10. Xiong, Y., Liu, X., Zeng, M., Zhang, L., Huang, G.: Identifying patch correctness in test-based program repair, In: Proceedings of the 40th international conference on software engineering, pp. 789–799. (2018)
11. Anand, S., et al.: An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. **86**, 1978–2001 (2013)
12. Taneja, K., Xie, T.: Diffgen: Automated regression unit-test generation, In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE, pp. 407–410. (2008)
13. Thummalapenta, S., Xie, T., Tillmann, N., De Halleux, J., Schulte, W.: Mseqgen: Object-oriented unit-test generation via mining source code, In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 193–202. (2009)
14. Fraser, G., Arcuri, A.: Evosuite: On the challenges of test case generation in the real world. IEEE sixth international conference on software testing, verification and validation IEEE **2013**, 362–369 (2013)

15. Schäfer, M., Nadi, S., Eghbali, A., Tip, F.: Adaptive test generation using a large language model, arXiv preprint arXiv:2302.06527 (2023)
16. Xia, C.S., Wei, Y., Zhang, L.: Automated program repair in the era of large pre-trained language models, In: Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery (2023)
17. Xia, C.S., Zhang, L.: Conversational automated program repair, arXiv preprint arXiv:2301.13246 (2023)
18. Sobania, D., Briesch, M., Hanna, C., Petke, J.: An analysis of the automatic bug fixing performance of chatgpt, arXiv preprint arXiv:2301.08653 (2023)
19. Feng, S., Chen, C.: Prompting is all your need: Automated android bug replay with largelanguage models, arXiv preprint arXiv:2306.01987 (2023)
20. Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., Maia, M.A.: Dissection of a bug dataset: Anatomy of 395 patches from defects4j, In: Proceedings of SANER (2018)