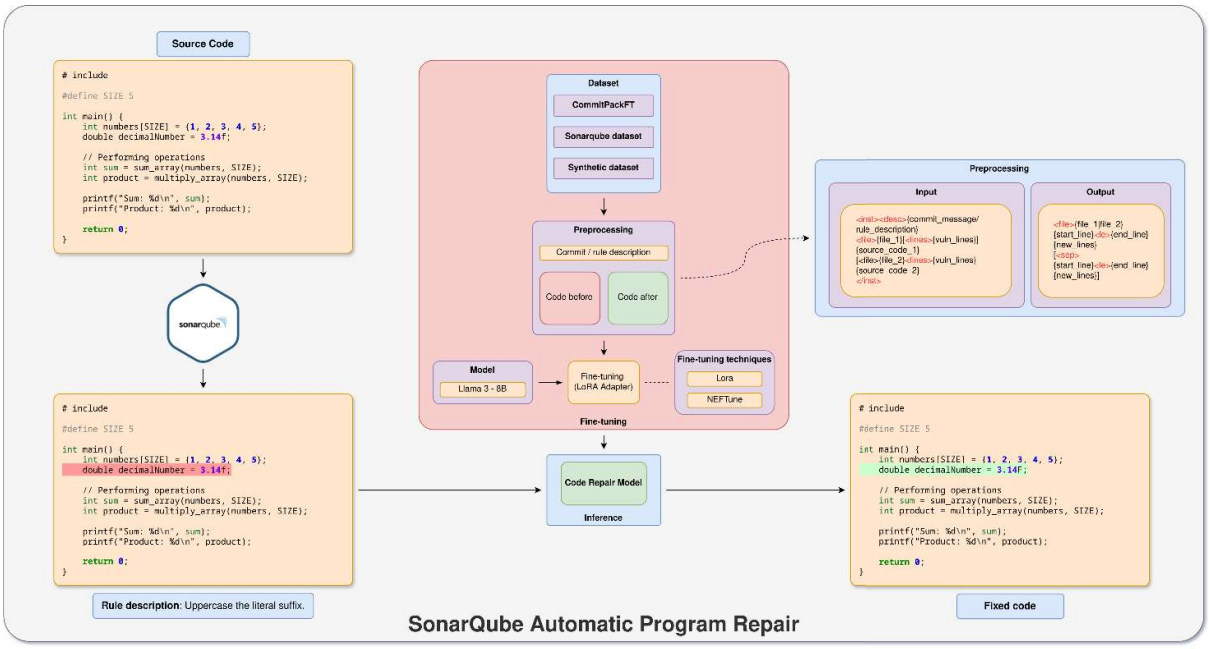

ARTIFICIAL INTELLIGENCE APPLIED TO CODE REPAIR AFTER CODE STATIC ANALYSIS VERIFICATION

Executive summary

Early technology development

New concepts for onboard SW development Campaign

Affiliation(s): Thales Alenia Space España (Prime), and Universidad de Alcalá (Sub 1)

Activity summary:
On-board software quality is very important for the space industry, and Advances in last years in Artificial Intelligence have drastically changed the landscape of automated code refinement.
This work main objective is to create an AI-based solution capable of automatically repairing code with the lowest human intervention, and with better performance than previous non-AI commercial solutions.
Stating with an extensive study on the most recent automated code repair approaches, an AI model has been created and refined, and then evaluated with real SW examples, finalizing with a demonstration execution of the complete model.

Publishing Date: 17/10/2024
Contract Number: 4000140197/22/NL/GLC/ov
Implemented as ESA Initial Support for Innovation

1. Introduction

On-board software quality is crucial for the space industry and is verified from the early stages of development. One of the first steps is static code analysis, performed by tools that assess adherence to coding standards. Developers then need to correct the identified issues, which can be labor-intensive. Reducing this effort is key to accelerating on-board software development.

In recent years, several automated repair tools have emerged, although few have adopted a data-driven approach, despite the availability of millions of open-source projects in popular repositories. With recent advancements in Artificial Intelligence (AI), especially in Natural Language Processing (NLP), innovative approaches have been developed to "translate" buggy code into clean code, showing promising performance.

This work aims to address the lack of effective solutions for low-level languages, such as C, which are common in on-board development. By creating an AI solution capable of automatically repairing code with minimal human intervention, we expect to surpass the limitations of previous commercial tools. Key objectives include developing an automated system for fixing code vulnerabilities in low-level languages, bridging the gap between learning-based automated fixing and on-board development, enhancing performance using advanced models like GPT-3 or CodeT5, and evaluating the impact of automation on development processes and resource efficiency.

2. Project Background

In the space industry, maintaining robust software performance is essential, as failures in extreme environments can lead to significant risks. Automatic Program Repair (APR) plays a vital role in addressing code errors detected by tools like SonarQube, which identifies bugs, vulnerabilities, and code smells. While pinpointing these issues is an essential first step, the ability to generate effective automatic fixes is an ongoing challenge in software engineering.

The state of the art in APR encompasses a variety of methodologies. Traditional approaches include heuristic-based methods, which utilize predefined rules and patterns to create repair patches, and constraint-based techniques that restrict the search space for fixes. While these methods have demonstrated effectiveness, particularly in scenarios involving relatively simple and repetitive defects, they often rely on manually designed templates and require considerable domain-specific expertise to create and maintain. Although template-based approaches could potentially be effective in certain cases, they fall outside the scope of this project due to several reasons. Our proposed approach does not require predefined templates or extensive test suites, making it more flexible and broadly applicable. Evaluating traditional methods would involve significant additional effort in terms of developing, testing, and tailoring templates, which is beyond the focus of this project, particularly given our emphasis on leveraging large language models.

In contrast, modern approaches increasingly leverage machine learning, particularly neural machine translation (NMT). These methods utilize advanced neural networks—like recurrent neural networks (RNNs) and transformers—to model the repair process. Notable examples include CURE and the method-to-method repair model by Tufano et al., which effectively apply NMT architectures to APR.

The field of automatic vulnerability repair poses unique challenges, incorporating general-purpose methods as well as security-specific techniques that aim to generate repair specifications for identified vulnerabilities. Recent advancements in Large Language Models (LLMs) like GPT-4 have significantly enhanced the capabilities of APR, enabling models such as Codex and CodeT5 to understand and manipulate code in ways that resemble human programming logic.

This project seeks to capitalize on these advancements by creating a comprehensive dataset of programming defects, leveraging insights from SonarQube. By treating rule descriptions as explicit instructions for code modifications, we aim to fine-tune selected LLMs to improve their efficacy in generating precise code repairs. This initiative not only addresses existing gaps in the literature regarding APR for low-level languages but also strives to enhance software development practices in the space sector through greater automation and efficiency.

3. Methodology

The initial approach in this work involved fine-tuning Code Llama, a model recognized for its ability to handle large contexts, using datasets such as CommitPackFT and real-world examples of MISRA rule corrections made by human developers. This approach enabled the model to effectively address a range of real-world coding errors and compliance issues. The results showed that Code Llama was able to correct 17.4% of the identified issues, with 22.6% providing partial fixes. However, a significant portion (35.2%) of the generations were incorrect, highlighting areas for improvement. In comparison, ThalesGPT only managed to correct 4.5% of the issues, indicating that Code Llama performed significantly better in understanding and applying fixes to the code.

To improve performance, we switched to the Llama 3 model and expanded the dataset by incorporating new real-world examples from recent projects, as well as a synthetic dataset generated from SonarQube rules. This extended dataset provided more diverse and comprehensive examples of code errors and fixes. With these improvements, the successful correction rate increased to 40%, and incorrect fix rate decreased to 18.1%. These results demonstrate the improved reliability and effectiveness of the refined model in addressing real-world coding issues.

4. Key Findings

The experimental framework established to evaluate our code repair solution has yielded significant insights into the efficacy of various models, particularly in addressing real-world coding errors.

1. **Model Comparison**: The baseline for our evaluations was a customized version of Code Llama, known for its superior handling of lengthy contexts. Its performance was benchmarked against ThalesGPT, a modified version of GPT-3.5. The experiments utilized datasets like the Bug Fix Corpus and the OctoPack, focusing on a variety of code errors and high-quality commit messages. Additionally, Code Llama was trained on datasets containing examples of issues identified through MISRA rules, where corrections were made by human developers. This integration of manually corrected MISRA examples provided the model with a more grounded understanding of real-world coding standards and practices, enhancing its performance in addressing compliance-related issues.
2. **Correction Rates**: Initial results from the Code Llama model indicated it could resolve 17.4% of identified issues, with 22.6% providing partial corrections, but a concerning 35.2% were

incorrect. In contrast, ThalesGPT managed to correct only 4.5% of issues. These findings underscore Code Llama's greater potential in understanding coding conventions and best practices.

3. **Evaluation Metrics**: The study employed rigorous evaluation metrics to assess both the precision and efficiency of each model. Code Llama demonstrated an ability to process requests in an average of 38 seconds, while ThalesGPT took about 68 seconds. Despite slight differences in speed, both models operated within acceptable time frames for practical deployment.
4. **Correction Quality and Context**: Detailed analysis revealed that the effectiveness of Code Llama varied significantly across different MISRA rules. For instance, it achieved a 100% success rate for some rules but struggled with others, such as the misuse of parameters, where context misinterpretation led to a high rate of incorrect corrections. This highlights the need for improved context awareness in automated corrections.
5. **Cost Analysis**: A preliminary cost evaluation suggested that while ThalesGPT might seem cheaper for low call volumes, Code Llama becomes more economically viable with increased usage due to its fixed-cost model. The annual estimated cost for deploying Code Llama was approximately $4,679, compared to ThalesGPT's variable cost structure.
6. **Model Refinement**: After transitioning to the Llama 3 model with a new dataset based on MISRA 2012 standards, as well as incorporating a synthetic dataset generated from SonarQube rules, its ability to provide accurate corrections significantly improved, raising the successful correction rate from 17.4% to 40%. The number of incorrect corrections dropped from 35.2% to 18.1%, showcasing the model's enhanced reliability and effectiveness in real-world applications.
7. **Deployment of integration**: The repair tool has been integrated into the final OBSW environment, which is hosted on a GitLab server. This integration allows the tool to automatically assist developers by providing detailed code suggestions whenever a new commit is pushed to the repository. Instead of requiring developers to manually input rules, the system is now connected to a SonarQube server that automatically provides the necessary code analysis rules. These rules serve as input for the repair tool to generate fixes. The tool was also packaged into a Docker image, making it more portable and easier to deploy in various environments, following standard engineering practices. Additionally, a GitLab extension was created to fully integrate the tool into the CI/CD pipelines. Each time a new commit triggers a pipeline, the tool runs automatically and generates a code quality report with suggestions for fixing errors identified by SonarQube. These reports are presented to developers in CSV or JSON format, helping them to quickly review and decide on the recommended code fixes. This integration streamlines the development process, making it easier for teams to maintain high-quality code while adhering to coding standards.
8. **Impact on Development Efficiency**: The improvements realized not only automated a considerable portion of the coding correction process but also provided developers with valuable insights for resolving more complex issues. This dual function, combining automated fixes and developer assistance, enhanced compliance with coding standards and ultimately accelerated the development cycle. By reducing manual debugging and code review times, the team's overall efficiency and productivity improved significantly.

5. Conclusion

These findings demonstrate the potential of advanced **LLMs** like **Code Llama** in addressing software quality challenges and ensuring compliance with coding standards, particularly in environments requiring adherence to **MISRA rules**. The integration of human-corrected MISRA issues has further refined the model's capabilities, making it a more effective tool for automated program repair, while also identifying areas for ongoing refinement and development.

6. Future Work

To build on the project's outcomes, further research could focus on improving the model's contextual understanding, especially in complex scenarios such as parameter misuse, where Code Llama showed limitations. One possible recommendation is to expand the dataset to include more diverse real-world coding environments, such as those involving different programming languages or frameworks, which would improve the model's adaptability to different development environments. In addition, techniques such as Reinforcement Learning from Human Feedback (RLHF) or Direct Preference Optimization (DPO), which use human input to further improve the performance of the model, could be explored. Finally, testing the effectiveness of the model on larger, enterprise-level codebases would provide valuable insight into its scalability, making it more applicable to industrial applications. These steps could be implemented by incorporating iterative updates to the tool within the existing GitLab CI/CD integration, ensuring real-time improvements without disrupting developer workflows.

Another potential direction could be to integrate confidence estimation into the model's inference process. For example, the system can use the model's probabilistic outputs to generate a confidence score for each proposed code fix. This confidence metric would provide developers with valuable insights into the reliability of the generated patches, indicating whether a fix is likely valid, potentially incorrect, or uncertain. Including this feature would enhance the tool's utility by guiding developers on which automated fixes can be trusted and which require additional evaluation, thereby optimizing the code review process and improving overall development efficiency.