# Using LLMs to extract OCL specifications from Java and Python programs: an empirical study

Hanan Abdulwahab Siala[1], Kevin Lano[1]

[1]King's College London, London, UK

**Abstract**

This paper presents a comprehensive study of the application of several open-source Large Language Models (LLMs) for abstracting Object Constraint Language (OCL) specifications from source code. We aim to provide researchers and developers with insights into the capabilities and limitations of using different LLMs to abstract OCL specifications from code.

We evaluate a collection of open-source LLMs of comparable size (StarCoder2, LLaMA, CodeLlama, Mistral, and DeepSeek) by prompting them to generate OCL specifications for both Java and Python programs. The results show that both Mistral and DeepSeek outperform other LLMs in abstracting OCL specifications from both languages.

**Keywords**

Object Constraint Language (OCL), Machine Learning, Large Language Models (LLMs), Reverse engineering, Java programs, Python programs.

## 1. Introduction

Many organizations depend on software systems to accomplish their tasks. Over time, these systems can become *legacy systems* if they are not properly maintained and evolved to meet new requirements and needs. *Reverse engineering* processes aim to help maintainers understand critical software systems and to support their maintenance and evolution.

Model-Driven Engineering (MDE) [1] facilitates the reverse engineering process by providing high-level abstractions of software systems. The Object Management Group (OMG) has defined open standards to design and develop software using various models and diagrams, including the Unified Modeling Language (UML) [2] and Object Constraint Language (OCL) [3] standards. UML is the most common standard notation for modeling software systems. It consists of various diagrams, including class diagrams, which are the most widely used to represent classes and relationships. Object constraint language (OCL) [3, 4] plays a significant role in understanding software systems. OCL is a textual specification language used to define constraints on models. Initially, it was introduced into UML as a constraint language to address the shortcomings of the diagrammatic notations of UML. Then, it was expanded in its scope and became essential to many MDE approaches [4]. Abstracting OCL specifications from source code remains a challenge that often requires deep expertise, significant effort, and time. As a result, there is a growing need for automated tools that can assist in abstracting OCL specifications from source code files.

Large Language Models (LLMs) are a specific type of machine learning (ML) technology that are pre-trained on massive amounts of text data. Once trained, they can be optimized to perform specific downstream tasks through additional training on demonstration examples. LLMs can be classified into two categories: open-source and closed-source. Open-source LLMs, such as LLaMA [5] and Mistral [6] are relatively inexpensive, and their source code and underlying architecture are publicly accessible. In contrast, closed-source LLMs, such as GPT3&4 [7] and Gemini [8] are expensive, and they are accessible only under specific terms by their companies, like OpenAI and Google. LLMs have had a major impact

*Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10-13, 2025*

hanan.siala@kcl.ac.uk (H. A. Siala); kevin.lano@kcl.ac.uk (K. Lano)

0009-0003-4693-8707 (H. A. Siala); 0000-0002-9706-1410 (K. Lano)

on various fields, including software engineering, where LLMs have been used in various software engineering tasks [9].

Over the past twenty years, there has been a large amount of research concerned with the abstraction of various representations from existing software systems. A recent systematic literature review (SLR) [10] of model-driven reverse engineering (MDRE) approaches over this period identified that OCL specifications get little attention in MDRE approaches compared to other UML models and diagrams, and using LLMs to abstract OCL specifications from source code is rare. Hence, our research investigates the capabilities and limitations of five open-source LLMs for abstracting OCL specifications from programs, intending to identify how successfully LLMs may be utilized in the reverse engineering process.

Our research questions are as follows:

**RQ1** How well do different LLMs perform in abstracting OCL specifications from Java and Python programs?

**RQ2** What are the highest accuracy, consistency, and F1 score percentages achieved by LLMs for the abstraction of OCL specifications?

**RQ3** What are the common failures encountered in the generation of OCL specifications using LLMs?

The remainder of the paper is organized as follows: related work is presented in section 2, while our methodology is explained in section 3. The evaluation of various LLMs is illustrated in section 4. The threats are outlined in section 5, and conclusions and future work are given in section 6.

## 2. Related Work

Lano and Siala [11] introduced a new approach to translate software systems from one programming language to another using MDE techniques. Firstly, the source code is reverse-engineered into a specification expressed using UML and OCL. Then, these specifications are forward-engineered into the desired target language, aiming to ensure semantic preservation. The AgileUML toolset is used to accomplish the re-engineering process.

Abukhalaf *et al.* [12] performed an empirical study to investigate the reliability of OCL specifications generated from natural language using OpenAI's Codex. They found that using Codex without further contextual information produced low syntactic correctness and semantic accuracy scores (11% and 9%, respectively). Enhancing Codex with few-shot learning improved these figures to 53.2% and 39%.

Siala and Lano [13] presented a reverse engineering approach based on LLMs, called LLM4Models, to abstract OCL specifications from Java and Python programs. The evaluation results show that LLM4Models can abstract OCL from both Java and Python programs. A fine-tuned Mistral LLM is used for this work.

Xie *et al.* [14] investigated several LLMs for the task of abstracting specifications from program comments. They found that the StarCoder2-15B and CodeLlama-13B perform best for this task when few-shot learning enhances the LLM performance.

In contrast to previous studies, we investigate the ability of a range of LLMs to abstract OCL specifications from program code.

## 3. Methodology

### 3.1. Model Selection

To investigate the capability of different LLMs to abstract UML and OCL from Java and Python programs, we experiment on five open-source LLMs, all from Hugging Face [15], which have achieved promising results in various code-related tasks [16]:

1. **StarCoder2**: StarCoder2 [17] models with 3B, 7B, and 15B parameters were trained on 3.3 to 4.3 trillion tokens and thoroughly evaluated on various Code LLM benchmarks [17]. The smallest model (3B) and the largest model (15B) outperform comparable models. StarCoder2-7B is used in our experiment.

2. **LLaMA**: LLaMA is a collection of LLMs in 8B, 70B, and 405B sizes trained on approximately 15 trillion data tokens from publicly available sources. We use Llama-3.1-8B, which was released on July 23, 2024.
3. **CodeLlama**: CodeLlama [18] is a code LLM built on the LLaMA 2 architecture. CodeLlama has been trained on a large dataset of code and natural language text related to code to support both code generation and infilling tasks in different programming languages, including Java, Python, C#, PHP, and C++. CodeLlama-7b-hf is used in our experiment.
4. **Mistral**: Mistral [6] is a decoder-only transformer. It gives good results compared with other LLMs for natural language understanding and generation, and it can be used for coding tasks. Mistral-7B-v0.3 is used in our experiment.
5. **DeepSeek**: DeepSeek-R1 [19] was trained using large-scale reinforcement learning (RL), and it was introduced to address the challenges of DeepSeek-R1-Zero. Multi-stage training and cold-start data were incorporated before RL. DeepSeek is distilled into smaller dense models based on Qwen and Llama, which perform exceptionally well on benchmarks. DeepSeek-R1-Distill-Llama-8B is used in our experiment.

The temperature hyperparameter of 0.2 was set for the chosen LLMs. This reduces the variability of response, increases the determinism of the output, and preserves the ability to provide diverse responses. The LLMs are used without fine-tuning or other forms of additional information (such as few-shot learning) in order to identify their baseline capabilities for the task.

### 3.2. Samples Selection

We collected example programs from the AVATAR [20] and CoTran [21] datasets of Java and Python programs and generated corresponding OCL specifications for these:

1. We selected thirteen random samples of Java and Python programs from the datasets. The programs are real-world solutions to programming problems, involving numeric computations and data-structure processing. The programs are used in our experiment to evaluate the accuracy and consistency of the selected LLMs.
2. We created ground-truth OCL specifications for the sample programs using the AgileUML toolset options to reverse-engineer Java and Python to UML/OCL.

### 3.3. Evaluation Criteria for OCL Specifications Abstraction

To evaluate syntactic correctness, we attempted to parse the generated OCL specifications using the USE toolset parser[1] and the ANTLR OCL parser[2].

For semantic correctness, we consider four separate aspects of the abstracted specifications:

**C1:** Are program classes correctly expressed in the generated OCL specifications?
**C2:** Are program operations correctly expressed in the generated OCL specifications?
**C3:** Are program statements correctly expressed in the generated OCL specifications?
**C4:** Are program variables correctly expressed in the generated OCL specifications?

### 3.4. Prompt Engineering for OCL Specifications Abstraction

A single prompt is used as an instruction for each LLM. The prompt was engineered to obtain the requested OCL specifications, ensuring non-redundancy and non-duplication. An Alpaca-style prompt format is used to abstract OCL specifications from Java programs, as shown in Fig. 1.

This version of the prompt produced the most accurate and consistent results. A corresponding prompt is used for Python abstraction. Following the *nucleus sampling* protocol [22], we take the best response from 5 results for each input case and each LLM.

[1] https://sourceforge.net/projects/useocl/
[2] https://github.com/antlr/grammars-v4

```
"""
Below is an instruction that describes a task, paired with an input that provides
further context. Write a response that appropriately solves the following Task:

### Instruction:
Generate an Object Constraint Language (OCL) for the provided Java code. The output
should:
1. Ensure no repeated or redundant operations or classes.
2. Include only the OCL code for the provided Java code.
3. Do not include statements for items not found in the Java code.

### Input:
... source code ...

### Response:
"""
```

**Figure 1:** Prompt schema to generate OCL specifications.

### 3.5. Evaluation Metrics

The selected LLMs are evaluated using key performance metrics, including *accuracy*, *consistency*, and *F1 score* of each LLM. Accuracy is defined as the proportion of source code elements that are correctly represented as OCL elements in the categories C1 to C4 above.

Accuracy is also referred to as *recall*, and it is calculated using Equation (1), where a true positive (TP) is an element correctly translated from code to an OCL element, and a false negative (FN) is an element that is not translated or translated incorrectly.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{1}$$

Consistency is the proportion of elements in the generated OCL that are correctly derived from the source programs. Consistency is important for ensuring traceability and alignment of the derived OCL specifications concerning the source code. Consistency is also referred to as *precision* and is calculated using Equation (2), where a false positive (FP) is an element that appears in the OCL specifications that is not derived correctly from a source code element.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2}$$

Accuracy measures how complete and correct the abstraction process is, while consistency measures the quality of the generated OCL specifications in terms of the absence of spurious elements not derived from the source code.

The F1 score balances both precision and recall, as shown in Equation (3). It considers both false positives and false negatives and can be particularly useful when an LLM produces conflicting results; for example, high precision but low recall, or vice versa.

$$\text{F1 score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

## 4. Results and Discussion

Application of the USE toolset OCL parser[3] to the generated OCL resulted in many errors. While the surveyed LLMs have some knowledge of OCL syntax and the correct use of keywords such as *context*, *pre* and *post*, they fail to produce syntactically-correct OCL in almost all cases because of the incorrect use of types (using program types such as *int*[] instead of OCL types) and the use of invalid operators

[3]https://sourceforge.net/projects/useocl/

such as *in*, ! =, %, or the use of a mixed syntax from different formal and programming languages. Thus, the ANTLR OCL parser, which accepts a generalised OCL syntax, was used to compare the syntactic correctness of the LLMs. The results show that the OCL knowledge of these LLMs seems quite poor (Table 1).

| *LLM* | *Syntax correctness* | |
|---|---|---|
| | *Java* | *Python* |
| CodeLlama | 7.69% | 7.69% |
| DeepSeek | 0% | 0% |
| LLaMA | 0% | 7.69% |
| Mistral | 15.4% | 0% |
| StarCoder2 | 0% | 0% |

**Table 1**
Percentage of syntactically correct results.

For the Java examples, only at most 15.4% of results are syntactically correct OCL, while for the Python examples, the maximum percentage is only 7.69%. In most cases, there are multiple syntax errors in the results, which prevent automated repair of the syntax.

Initially, we also intended to use the USE modeling tool to semantically evaluate the generated OCL specifications from various LLMs to identify if the functionality of the OCL was equivalent to that of the source code. However, it was not possible to process the generated OCL due to the presence of syntax errors in almost all results. Instead, we opted to evaluate the generated OCL manually by inspection. We compared the specifications and the source programs concerning the four aspects C1 to C4 described in Section 3.3 above. Fig. 2 depicts the results of this comparison between the generated OCL specifications and the Java and Python source codes for each considered LLM.

## 4.1. Evaluation of LLMs to Generate OCL Specifications from Java Programs

**Accuracy** Fig. 2a presents the accuracy of various LLMs in abstracting OCL specifications from the 13 selected Java examples. DeepSeek and Mistral achieve the best accuracy scores in general, with DeepSeek achieving some of the highest scores across all the test examples. StarCoder2 and LLaMA follow these LLMs in terms of accuracy results. CodeLlama gives lower accuracy, showing the worst performance on test example 9 (a program to identify if an integer is composite).

**Consistency** Fig. 2b presents the consistency of the same LLMs in abstracting OCL specifications from the same Java examples. Again, DeepSeek performs well across most test examples. StarCoder2 and Mistral also demonstrate good performance. The former demonstrates a minor drop in example 2 (a procedure to binary search a segment of a sorted array), while the latter experiences a large drop in example 9. LLaMA and CodeLlama show lower consistency, with CodeLlama showing a considerable consistency drop in performance in test example 9.

## 4.2. Evaluation of LLMs to Generate OCL Specifications from Python Programs

**Accuracy** As shown in Fig. 2c, Mistral outperforms other LLMs for OCL accuracy across most Python test examples. DeepSeek and StarCoder2 also have competitive accuracy; however, DeepSeek exhibits sharp drops in both test cases 5 (a file processing example) and 13 (a pandas data analysis example). LLaMA and CodeLlama show greater fluctuations, with CodeLlama achieving zero accuracy for five examples.

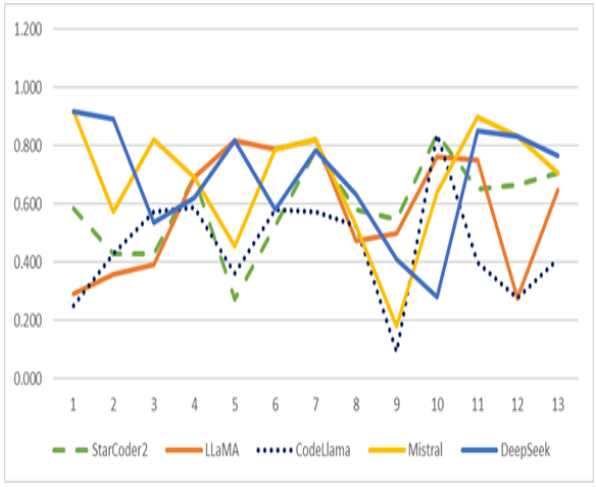
**Consistency** The consistency results for Python are presented in Fig. 2d. StarCoder2 and Mistral have higher and more stable consistency results compared to other LLMs. DeepSeek and LLaMA follow these LLMs, although DeepSeek shows remarkable drops in consistency for examples 5 and 13. CodeLlama has lower consistency overall, with zero consistency in five cases.

### 4.3. Overall Results

Fig. 3a summarizes the overall (average) accuracy for each LLM across both Java and Python. DeepSeek has the highest accuracy for Java programs, closely followed by Mistral and then StarCoder2 and LLaMA, while Mistral has the highest overall accuracy score to abstract OCL specifications from Python programs, followed by StarCoder2, DeepSeek, and LLaMA. CodeLlama has the lowest accuracy score for both Java and Python programs. It can be noted that the accuracy for Python is usually lower than for Java; this is a common tendency which has been found in program translation and reverse-engineering work, due in part to the lack of explicit types in Python code.

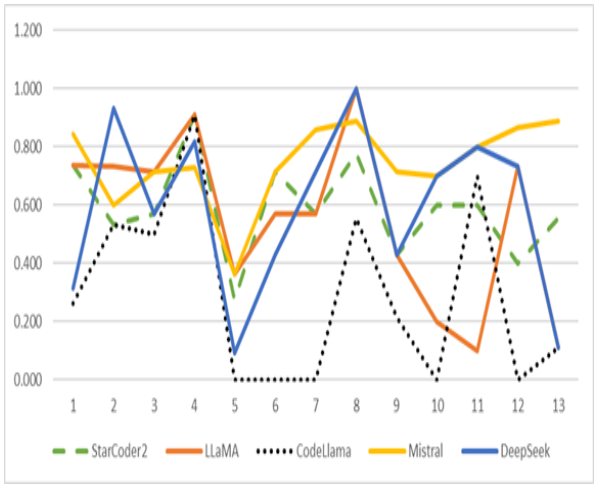
Fig. 3b displays the overall consistency scores across both Java and Python examples. For Java programs, DeepSeek, StarCoder2, and Mistral show the highest consistency, while StarCoder2 and Mistral achieve higher consistency scores for Python. CodeLama has the lowest consistency scores for both Java and Python examples. Again, consistency is lower for Python than for Java.

Finally, Fig. 3c presents the overall F1 score results. Both DeepSeek and Mistral have high F1 scores for Java, whereas StarCoder2 and LLaMA achieve moderate scores. Mistral and StarCoder2 are the best-performing LLMs for Python using this measure. CodeLlama has the lowest score for both Java and Python programs.

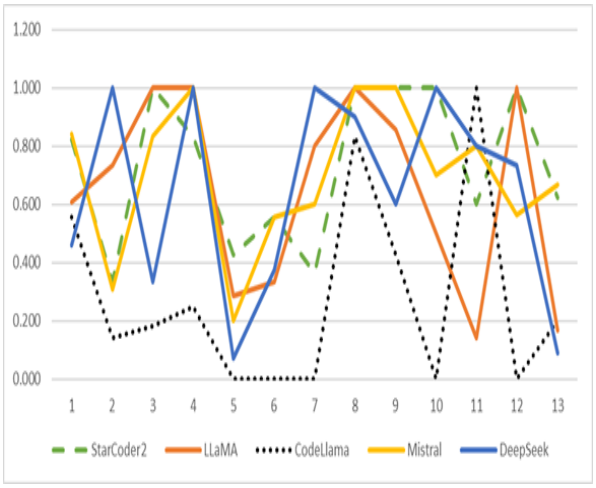



(a) Accuracy for Java examples, numbered 1 to 13.


(b) Consistency for Java examples, numbered 1 to 13.

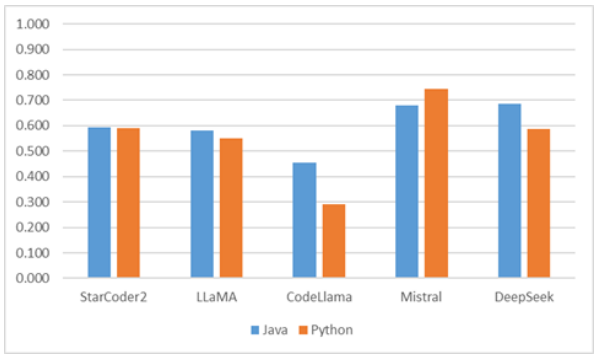



(c) Accuracy for Python examples, numbered 1 to 13.

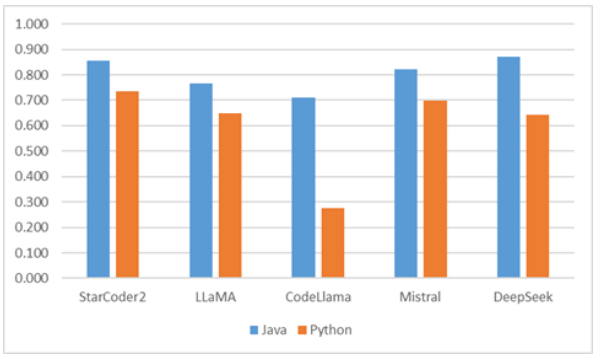



(d) Consistency for Python examples, numbered 1 to 13.

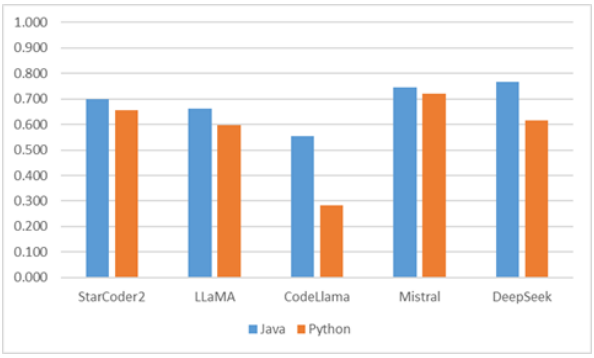
**Figure 2:** Accuracy and consistency results for Java and Python examples.




(a) Overall accuracy for Java and Python examples.




(b) Overall consistency for Java and Python examples.




(c) Overall F1 score for Java and Python examples.

**Figure 3:** Overall accuracy, consistency, and F1 score.

### 4.4. Discussion

With regard to the research questions of Section 1, we can conclude that:

**RQ1** Some LLMs, particularly the Mistral and DeepSeek versions considered here, attain a good level of accuracy and consistency for abstracting OCL specifications from real-world Java and Python programs, which could facilitate the reverse-engineering of Java and Python.

**RQ2** DeepSeek achieves over 85% consistency and nearly 70% accuracy for Java abstraction (F1 score 76%), whilst Mistral achieves over 70% accuracy and 70% consistency for Python abstraction (F1 score 71%).

**RQ3** Despite the overall high accuracy and consistency of the Mistral, DeepSeek, and StarCoder2 LLMs, they also exhibit errors in most cases. Syntactic errors occur frequently in results.

The LLMs generally perform better with smaller and simpler code examples. An example where there is high accuracy and consistency is the Mistral abstraction of the following Java example (linear search, Java example 1):

```
static int findElement(int[] elements,
    int size, int target) {
  int index;
  for (index = 0; index < size; index++) {
    if (elements[index] == target) {
      return index;
    }
  }
  return -1;
}
```

Mistral successfully infers two correct pre and post-conditions for this case (although the precondition `target >= 0` is invented by the LLM):

```
pre: size >= 0 and target >= 0

post: result = -1 implies
    (size = 0 or
     not (exists i : Integer | elements[i] = target))
```

However, there is also incompleteness in this abstraction example, because the LLM is unable to successfully express the case where an index satisfying $elements[i] = target$ is found. Note also that the syntax of the *exists* predicate here is not valid according to the OCL standard.

Hallucinations are also produced; for example, in the LLaMA application to Java example 3 (finding the equilibrium index of an array), part of the result appears to be plausible, but semantically the postcondition is actually a mistaken interpretation of the code functionality:

```
context EquilibriumIndexFinder:
   operation: findEquilibriumIndex(array : Sequence(Int), length : Int) : Int
      pre: length = array->size()
      post: result = array->asSequence()->select(i | i = length/2)->one()
```

For other examples, such as cases with more complex functionality or that involve file processing, random number generation, or other aspects for which suitable OCL abstractions do not exist, all of the LLMs fail to produce satisfactory results. A common failure occurrence in such cases is that the LLM effectively returns a syntactic variant of the input program as its answer. In almost all cases, the result format is incorrect according to the OCL standard and cannot be parsed by an OCL parser.

Table 2 summarises the types of errors encountered in LLM outputs.

| *LLM* | *Incompleteness* | *Hallucination* | *Copies source* | *Spurious* |
|---|---|---|---|---|
| CodeLlama | 11.5% | 11.5% | 30.8% | 42.4% |
| DeepSeek | 4% | 7.7% | 30.8% | 11.5% |
| LLaMA | 30.8% | 4% | 38.5% | 15.4% |
| Mistral | 7.7% | 7.7% | 38.5% | 7.7% |
| StarCoder2 | 23% | 15.4% | 38.5% | 4% |

**Table 2**
Errors in LLM code abstraction to OCL.

Fine-tuning or other forms of supervised retraining of the LLMs could help to reduce such errors and to improve the syntactic and semantic correctness of the result.

## 5. Threats to Validity

In this section, we present the potential threats that could affect the results of our empirical study.

### 5.1. Response of models

In some cases, we obtained responses from LLMs that are not related to OCL specifications. We mitigated this by re-running the query (up to 5 times) until we obtained a reasonable response.

### 5.2. Evaluation of the responses

Automated evaluation of the produced specifications was not possible due to the failure of results to conform to the OCL grammar in most cases. Thus, we carried out manual analysis. Human error and bias may be introduced when the generated OCL specifications are evaluated manually. This manual evaluation process might produce inconsistent or inaccurate results. However, this was mitigated by having the second author inspect cases where the first author was unsure about the results.

### 5.3. Scope of the experiment

The sample Java and Python programs were randomly selected from published program repositories that have been widely used for other machine learning research on code. The examples span different types of programming problems, from numerical computations to data-structure processing and file processing. Thus, they are representative of programs that could be encountered in real-world reverse engineering.

We have used only open-source LLMs in our investigation, and there is a risk that more powerful LLMs are not included. However, this threat was partially solved by including open-source LLMs from different LLM families. Also, the latest powerful open-source DeepSeek LLM is included in our experiment.

## 6. Conclusion and Future Work

This paper empirically studied the use of LLMs to abstract OCL specifications from Java and Python code. The results show that both Mistral and DeepSeek outperform the other LLMs in abstracting OCL specifications. By contrast, CodeLlama has low accuracy and significant inconsistency in the abstraction process. In general, LLMs have problems with the correct abstraction of OCL from code due to the limited expressiveness of OCL compared to Java and Python, and due to incomplete LLM knowledge of OCL syntax and semantics.

In future work, we aim to expand our work to include other programming languages like C++, C#, COBOL, and others. We also intend to fine-tune open-source LLMs (e.g., Mistral or DeepSeek) on large-scale datasets of these programming languages, together with their corresponding OCL specifications, to improve the performance of the LLMs in reverse engineering tasks.

## 7. Data Availability

The data used in this study - including Java and Python programs and their corresponding OCL specifications generated by the surveyed LLMs - are available in our online repository [23].

## Acknowledgment

Hanan Siala acknowledges the financial support provided by the Libyan Ministry of Higher Education and Scientific Research. She also acknowledges the use of resources provided by King's Computational Research, Engineering, and Technology Environment (CREATE).

## Declaration on Generative AI

The authors used StarCoder2, LLaMA, CodeLlama, Mistral, and DeepSeek LLMs to extract OCL specifications from program code. All outputs were reviewed and validated by the authors, who take full responsibility. No proprietary/third-party confidential data were provided to these tools.

## References

[1] S. Kent, Model driven engineering, in: Integrated Formal Methods: Third International Conference, IFM 2002 Turku, Finland, May 15–18, 2002 Proceedings, Springer, 2002, pp. 286–298.

[2] OMG, OMG Unified Modeling Language (UML), Version 2.5.1, 2017. https://www.omg.org/spec/UML.

[3] OMG, Object Constraint Language 2.4 Specification, OMG document formal, 2014. https://www.omg.org/spec/OCL/2.4/About-OCL.

[4] J. Cabot, M. Gogolla, Object constraint language (OCL): a definitive guide, in: International school on formal methods for the design of computer, communication and software systems, Springer, 2012, pp. 58–90.
[5] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, G. Lample, LLaMA: Open and efficient foundation language models, 2023. URL: https://arxiv.org/abs/2302.13971. arXiv:2302.13971.
[6] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al., Mistral 7b, arXiv preprint arXiv:2310.06825 (2023).
[7] W. Zhao, et al., A survey of large language models, arXiv 2303.18223v10 (2023).
[8] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, et al., Gemini: A family of highly capable multimodal models, 2024. URL: https://arxiv.org/abs/2312.11805. arXiv:2312.11805.
[9] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, LLMs for software engineering: a systematic literature review, arXiv 2308.10620 (2023).
[10] H. A. Siala, K. Lano, H. Alfraihi, Model-driven approaches for reverse engineering - a systematic literature review, IEEE Access (2024). doi:10.1109/ACCESS.2024.3394732.
[11] K. Lano, H. A. Siala, Using model-driven engineering to automate software language translation, Automated Software Engineering 31 (2024). URL: https://doi.org/10.1007/s10515-024-00419-y. doi:10.1007/s10515-024-00419-y.
[12] S. Abukhalaf, M. Hamdaqa, F. Khomh, On Codex prompt engineering for OCL generation: An empirical study, in: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), 2023, pp. 148–157. doi:10.1109/MSR59073.2023.00033.
[13] H. A. Siala, K. Lano, Towards using LLMs in the reverse engineering of software systems to object constraint language, in: Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2025. URL: https://conf.researchr.org/home/saner-2025.
[14] D. Xie, B. Yoo, N. Jiang, M. Kim, L. Tan, X. Zhang, J. Lee, How effective are Large Language Models in generating software specifications?, in: SANER 2025, 2025.
[15] Hugging Face, Hugging face, Online, 2016. Available at: https://huggingface.co/ [Accessed Mar. 2025].
[16] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, J. Chen, A survey of LLMs for code, arXiv (2024).
[17] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, et al., StarCoder 2 and the stack v2: The next generation, 2024. URL: https://arxiv.org/abs/2402.19173. arXiv:2402.19173.
[18] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al., Code llama: Open foundation models for code, arXiv preprint arXiv:2308.12950 (2023).
[19] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, X. Zhang, X. Yu, Y. Wu, Z. F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, et al., DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, 2025. URL: https://arxiv.org/abs/2501.12948. arXiv:2501.12948.
[20] W. Ahmad, M. Tushar, S. Chakraborty, K.-W. Chang, AVATAR: a parallel corpus for Java-Python program translation, arXiv:2108.11590v2 (2023).
[21] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, V. Ganesh, CoTran: An LLM-based code translator using reinforcement learning with feedback from compiler and symbolic execution, arXiv:2306.06755v4 (2024).
[22] A. Holtzman, J. Buys, L. Du, M. Forbes, Y. Choi, The curious case of neural text degeneration, in: ICLR 2020, 2020.
[23] H. A. Siala, K. Lano, Online repository for Java and Python programs with OCL specifications, https://doi.org/10.5281/zenodo.15108575, 2025. Accessed: May 2025.