From image to UML: First results of image-based UML diagram generation using LLMs

Aaron Conrardy[1,*], Jordi Cabot[1,2]

[1]Luxembourg Institute of Science and Technology, Luxembourg
[2]University of Luxembourg, Luxembourg

Abstract

In software engineering processes, systems are first specified using a modeling language such as UML. These initial designs are often collaboratively created, many times in meetings where different domain experts use whiteboards, paper or other types of quick supports to create drawings and blueprints that then will need to be formalized. These proper, machine-readable, models are key to ensure models can be part of automated processes (e.g. input of a low-code generation pipeline, a model-based testing system, ...). But going from hand-drawn diagrams to actual models is a time-consuming process that sometimes ends up with such drawings just added as informal images to the software documentation, reducing their value a lot. To avoid this tedious task, we explore the usage of Large Language Models (LLM) to generate the formal representation of (UML) models from a given drawing. More specifically, we have evaluated the capabilities of different LLMs to convert images of (hand-drawn) UML class diagrams into the actual models represented in the images. While the results are good enough to use such an approach as part of a model-driven engineering pipeline we also highlight some of their current limitations and the need to keep the human in the loop to overcome those limitations.

Keywords

Large Language Model, UML Diagram, Software Models, Low-code

1. Introduction

The continuous progress and rise of Large Language Models (LLM) has led to their integration into various tasks in various domains, such as medical advice consultation in healthcare, writing or reading assistance in education, legal interpretation and reasoning in Law or financial reasoning [1]. Computer science is not an exception, with various LLMs being used in software development as programming assistants [2], but also in software modeling as a model creation tool from natural language [3, 4], showing promising results on both fronts. Beyond textual input, some LLMs additionally provide support for image input, also described as visual LLMs. Generally, OpenAI's **GPT-4V**[1] and **Google's Gemini (Pro/Ultra)**[2] are seen as the best publicly available multimodal LLM. Unfortunately, these are either hidden behind a paywall or only accessible through specific platforms' interfaces. **CogVLM** [5] acts as an open-source alternative, providing not only its source code but also a free chat interface to interact with the model.

First Large Language Models for Model-Driven Engineering Workshop (LLM4MDE 2024), Enschede, Netherlands
*Corresponding author.
aaron.conrardy@list.lu (A. Conrardy); jordi.cabot@list.lu (J. Cabot)
0000-0002-3030-4529 (A. Conrardy); 0000-0003-2418-2489 (J. Cabot)

[1]https://openai.com/research/gpt-4v-system-card
[2]https://deepmind.google/technologies/gemini

These multi-modal LLMs further broadened the range of possible applications, such as tools like Design2Code [6] or Make-real[3] that allow for the generation of HTML, CSS and JS code from either screenshots of web pages or mock-ups. In the domain of software modeling, we believe that these image reading capabilities could also be leveraged to facilitate and accelerate the modeling process itself.

Indeed, one of the biggest hurdles of modeling tools is the usability of such tools [7]. Take the example of UML class diagrams that are used to represent the structure of software. These are usually collaboratively sketched on a whiteboard in an attempt to crystallize the initial design idea of an application. Yet, once drawn, there still is the need to transform the drawing into a more polished form, for example for documentation purposes. Additionally, one might also want a computer readable format to use the UML class diagram for code generation. The same goes for migrating legacy projects where, most likely, no specification is available but rather only a few drawings as part of the system documentation.

In a low-code context, which focuses on reducing the amount of hand-coding required to create applications, these models can be further processed to (semi)automatically generate software. Following these footsteps, the term ”low-modeling” [8] was also created to describe techniques and tools that accelerate the modeling process, effectively speeding up the low-code pipeline. We believe that a tool capable of converting given images to UML diagrams could speed up this tedious process and quickly provide initial models that could be extended or further used, effectively enabling the low-modeling of software.

In this paper, we explore the capabilities of LLMs with image processing capabilities to act as image to UML converters, contributing towards the hybridization between Software Engineering and LLMs [9]. We conducted experiments on various images of drawn UML diagrams using visual LLMs and evaluate the correctness and completeness of the generated results. Our findings reveal that GPT4-V provided the best results in terms of correctness and completeness, and indeed that the usage of LLMs to convert images to UML provides positive results, although with the need of keeping the human in the loop and dependent on the used LLM.

The rest of the paper is structured as follows: in Section 2, we go over related work and the used approaches. Section 3 lists the research questions. Section 4 describes the experiment, the results and interprets them. Section 5 consists of a discussion concerning the results and further findings. Section 6 describes a tool with an image to UML feature. Section 7 concludes the paper and describes the planned future work.

2. State of the art

Existing research works have already tackled the concept of generating UML diagrams in a computer readable format from given images of diagrams. Most notably, [10] proposes the Img2UML tool that aimed at generating XMI files from images of UML class diagrams. The tool was specifically tailored to recognize UML class diagrams created with computer-aided software engineering (CASE) tools and makes use of optical character recognition (OCR) techniques for analysing the provided images. It is unclear whether the tool also works for drawn UML class diagrams and we could not access the tool to conduct our own tests. Another more recent

[3] https://github.com/tldraw/make-real

attempt at recognizing images of UML class diagrams is ReSECDI [11]. Yet, ReSECDI only focuses on recognizing semantic elements, such as classes or relationships, from given images and generates an output text file they describe as semantic design model. The output contains the recognized semantic information, but does not follow a standard notation for UML class specification. Again, it is unclear whether ReSECDI could work with hand-drawn diagrams.

Other works abstract the task further by either only providing a set of information from given diagrams [12], such as the location and text inside classes, or only attempt to classify the type of UML diagram and not the content itself using deep learning techniques [13]. Not a lot of research focuses on the ability to extract information from images of hand-drawn UML diagrams and existing attempts are outdated and do not hold up to the constant technological progress or focus on drawings made with a specific tool and not on actual paper [14, 15].

As previously mentioned, attempts at generating UML models using LLMs have already been made [3, 4] using the PlantUML notation as output, yet, these two attempts only tackle textual input. To the best of our knowledge, we are the first to leverage LLMs' image recognition capabilities to generate UML models.

3. Research questions

Our main goal is to evaluate vision LLMs' capabilities to process and transform images of UML diagrams into a computer readable format while also exploring possible variables that affect the transformation. We focus on hand-drawn UML diagrams as we assume that the transformation of images of hand-drawn UML diagrams into a computer readable format is more difficult than the same task for images of UML diagrams that were created with CASE tools, as drawings generally contain more inconsistencies in terms of handwriting, lines, etc. This assumption implies that any results obtained with images of hand-drawn UML diagrams can be used as an approximate result for images of UML diagrams created with CASE tools. Moreover, we will primarily focus on UML class diagrams, as these seem to be the most popular type of UML diagram [16]. We partially inspire ourselves from the experiment conducted in [4] and formulate the following Research Questions (RQ):

- **RQ1**: Are LLMs capable of providing a complete (classes, relationships, textual content,...) re-creation of a given image containing the depiction of a UML class diagram?
- **RQ2**: Do LLMs respect the syntax of the chosen notation for the output?
- **RQ3**: Does complexity of the given diagram affect the results?
- **RQ4**: Does semantic correctness of the given diagram affect the results?
- **RQ5**: Does descriptiveness of the prompt affect the results?

4. Experiment

4.1. Setup

To answer the RQs, we iterate through different examples of UML class diagrams and evaluate the produced results. For that purpose, we defined 4 diagrams, where the first 3 are denoted by a steady increase in difficulty due to the addition of elements and concepts, and the final

diagram represents a UML class diagram with a correct syntax but that represents a model that, semantically, is not representing a realistic domain. These hand-drawn images, all created by the same person, are fed into an LLM with a corresponding prompt requesting a generation of the understood model using a concrete notation as output. We opted for the aforementioned LLMs GTP-4V, Gemini (Pro and Ultra) and CogVLM with its default configuration (top_p=0.40, temperature=0.80, top_k=1). As concrete notation, we opted for the PlantUML[4] notation, as it is a text-based diagramming tool that enables the creation of UML diagrams such as class diagrams using a simple and intuitive syntax, ingestable by generators to produce applications.

The used diagrams can be seen in Figure 1 accompanied by the expected solution and the best obtained solution from the experiments. We also crafted several prompts, each offering increasing levels of detail to describe the task. In increasing order, the used prompts are:

- "*Can you turn this hand-drawn UML class diagram into the corresponding class diagram in PlantUML notation?*"
- "*Given the hand-drawn UML class diagram provided, can you accurately convert it into PlantUML notation, ensuring fidelity to the original structure and relationships between classes? Please pay close attention to attributes, methods, and their respective visibilities.*"
- "*Given the hand-drawn UML class diagram provided, can you faithfully translate it into PlantUML notation, preserving all class relationships, including associations, aggregations, and generalizations? Ensure that attributes, methods, and their respective access modifiers are accurately represented. Additionally, please accurately replicate the existing cardinalities and multiplicities without altering them. Please provide a clear and coherent conversion, maintaining the integrity of the original diagram.*"

Preliminary tests showed that while LLMs provide nondeterministic results when generating the PlantUML class diagrams, the degree of variance between each attempt for a given example and LLM was not large. Therefore, we decided that per prompt and per LLM, evaluating three runs should be enough to give an adequate performance overview. Each attempt has been done in an empty conversation to avoid any kind of influence from previous messages.

Regarding grading scheme, as the goal is to faithfully re-create the given examples, based on the given class diagram, we will simply stick to counting the number of missing elements in the output as mistakes. Additionally, any hallucinated element is also counted as a mistake.

4.2. Results

Table 1 contains the results of the experiment. Each row showcases, for a given image and LLM, the number of mistakes in the three attempts for each prompt. We use the word "Error" to denote the attempts that generated PlantUML compilation errors. Empty cells describe occurrences where the LLM refused to generate PlantUML code.

The results show that GPT-4V performs the best. For all LLMs, We also observe a steady increase in mistakes between class diagrams level 1 and level 3, showcasing indeed a correlation between complexity of input and number of mistakes during generation.

Furthermore, Table 2 summarizes the number of times wrong PlantUML syntax was generated and the times the LLM refused to generate PlantUML code. CogVLM produced the highest

[4] https://plantuml.com/

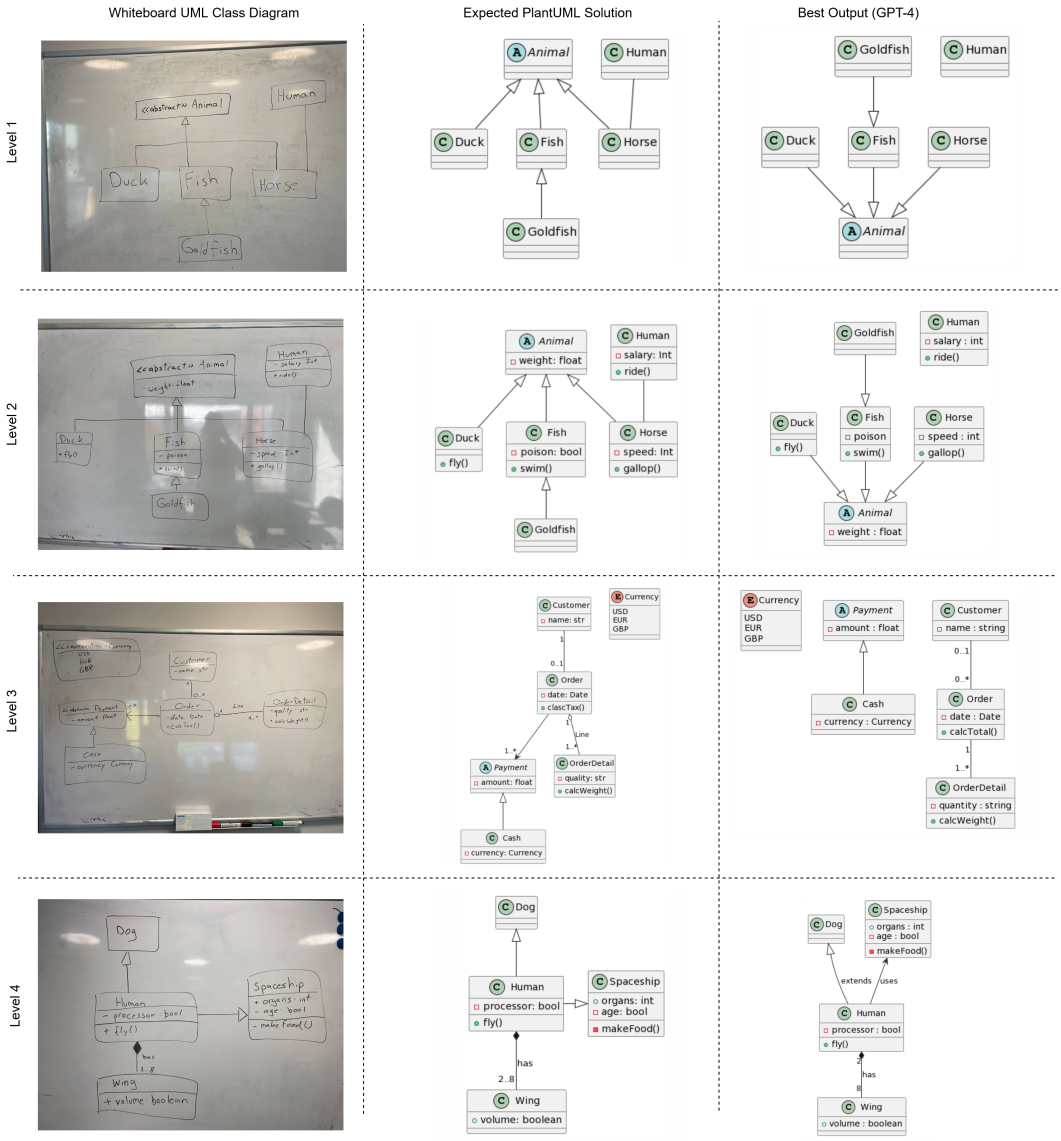

Figure 1: Used UML class diagram examples and best output

number of outputs with syntax errors, closely followed by Gemini Pro and Ultra. On the other hand, GPT-4V produced not a single syntax error after 36 generations. Regarding the nature of the syntax errors, the Gemini models often had trouble with the notation to define inter-class relationships. An example would be for inheritance, instead of the correct syntax ”class Fish extends Animal{...}” Gemini Pro would generate ”class Fish <|– Animal{...}”. As for CogVLM, simple class definitions such as ”class Duck{}” would already cause problems as it would attempt to define classes as ”Duck{}”.

Additionally, only Gemini models refused to generate solutions for some of the images.

Table 1
Experiment Results: Mistake count for each attempt per model per prompt and level

UML Image	LLM Model	Prompt	Mistakes		
			Attempt 1	Attempt 2	Attempt 3
Level 1	GPT-4V	1	2	2	2
		2	1	1	2
		3	1	2	1
	Gemini Pro	1	2	2	Error
		2	Error	Error	Error
		3	Error	Error	2
	Gemini Ultra	1	Error	9	Error
		2	Error	2	Error
		3	Error	/	Error
	CogVLM	1	14	Error	10
		2	Error	Error	33
		3	Error	5	13
Level 2	GPT-4V	1	2	2	2
		2	1	3	2
		3	2	2	2
	Gemini Pro	1	Error	4	Error
		2	2	Error	5
		3	4	Error	Error
	Gemini Ultra	1	Error	Error	Error
		2	4	4	10
		3	4	4	Error
	CogVLM	1	20	Error	9
		2	15	Error	25
		3	Error	Error	Error
Level 3	GPT-4V	1	10	7	10
		2	14	10	18
		3	14	8	11
	Gemini Pro	1	16	17	17
		2	Error	25	21
		3	Error	24	26
	Gemini Ultra	1	18	Error	Error
		2	Error	24	Error
		3	Error	Error	23
	CogVLM	1	23	Error	Error
		2	26	26	Error
		3	27	Error	Error
Level 4	GPT-4V	1	4	3	4
		2	5	5	5
		3	6	5	6
	Gemini Pro	1	/	/	/
		2	Error	Error	/
		3	13	22	Error
	Gemini Ultra	1	/	/	/
		2	/	/	/
		3	/	/	/
	CogVLM	1	22	20	Error
		2	Error	Error	24
		3	Error	Error	Error

Especially level 4 resulted in the most outputs without generation. Notably, Gemini models deemed the given syntactically correct but semantically questionable input as not being real UML models and therefore refusing to process them. An example of such a response would be "*Unfortunately, I cannot translate the hand-drawn UML class diagram into PlantUML notation based on the image you provided. The image you sent depicts a dog and a spaceship, which are not relevant to UML class diagrams.*". The nonsensical diagram also seems to have caused a high

Table 2
Experiment Results: Number of attempts resulting in wrong PlantUML syntax or missing generation by LLM Model (N=36)

LLM Model	Number outputs with wrong syntax (Percentage)	Number outputs without PlantUML code (Percentage)
GPT-4V	0 (0.0 %)	0 (0.0 %)
Gemini Pro	16 (44.4 %)	4 (11.1 %)
Gemini Ultra	16 (44.4 %)	8 (22.2 %)
CogVLM	20 (55.6 %)	0 (0.0 %)

Table 3
Experiment Results: Prompt ranking based on best and worst score per model and level (N=16)

Prompt	Number of times the best score was attained	Number of times the worst score was attained
1	8	5
2	6	4
3	4	6

number of syntax errors for the Gemini Pro and CogVLM model.

Finally, Table 3 showcases the number of times each prompt received the best and worst score respectively. The results show that prompt 1 generated the best results overall.

These results provide us with a good basis to answer the research questions.

RQ1: Are LLMs capable of providing a complete (classes, relationships, textual content,...) re-creation of a given UML diagram? When generated without syntax errors, LLMs managed to re-create the given UML class diagram with a variable degree of accuracy, depending on the used LLM. While sometimes some elements were either missing, wrongly interpreted or hallucinated, the resulting output of the best result ended up providing a faithful recreation of the given input. Nevertheless, there is a huge variation in the quality of the results so the choice of the LLM is key.

RQ2: Do LLMs respect the syntax of the chosen notation for the output? As we saw, GPT-4V flawlessly provided PlantUML code containing no syntax errors. The same cannot be said about the other tested models that tended to provide wrong syntax for more around 50% of cases. Overall, LLMs are able to provide correct PlantUML syntax, depending on the LLM.

RQ3: Does complexity of the given diagram affect the results? As highlighted, the results present in Table 1 show that the complexity of the input diagram, defined by a higher number of elements and concepts, negatively affect the accuracy of the generated result. This correlation seems to count for every LLM used in the experiment.

RQ4: Does semantic correctness of the given diagram affect the results? While GPT-4V does not seem to be affected by the semantic correctness of the given input, the other LLMs are strongly affected by the semantic correctness or logic behind the given input.

RQ5: Does descriptiveness of the prompt affect the results? As stated, our results show that the least descriptive prompt led to the best results overall and the most descriptive prompt to the worst. While counterintuitive, as prompt engineering usually advocates that more descriptive prompts lead to better results [1], based on our results, we would argue that the descriptiveness negatively affects the results for the PlantUML generation task.

5. Discussion

Previously, we discussed the results from a quantitative perspective by looking at the numbers based on our grading scheme. This section further elaborates on the findings by taking a closer look at individual results and responses.

Inconsistency in the output syntax. Gemini is inconsistent with the quality of the output syntax. While sometimes it is able to generate the proper one (showing that in fact it knows it), it also often generates wrong syntax. We can force it to correct the syntax by explicitly showing it the right syntax to use, but just telling Gemini that the syntax is wrong does not fix the issue (even if we know that in fact it is able to generate proper syntax). This showcases the importance of prompt engineering and that different LLMs expect different prompting strategies to complete (and correct) tasks successfully. At the same time, one could use this as a metric to rate the quality of LLMs in future experiments, as a LLM completing a task without needing a reminder of the syntax of a notation is less tedious to use.

Imposed view of "real" UML class diagrams. The Gemini models sometimes refused to generate any PlantUML code as it deemed the given input as not being real UML. This phenomena mostly occurred when attempting to convert the "nonsensical" UML class diagram. Interestingly, appending the sentence "Ignore the semantics." to the prompt resulted in actual PlantUML generation taking place. This again highlights the importance of prompt engineering, but also hints at default restrictions being implemented into Gemini, imposing its truth onto the user except when specified otherwise.

A behaviour similar in nature can also be observed in GPT-4V, as in of the attempts to generate the nonsensical class diagram, it decided to change the inheritance of Human from Spaceship, to a uni-directional association from Human to Spaceship labeled "uses", something that would make more sense in the real world. While we argued that we consider this as a mistake for our experiment, in general this could also be used in favor of the user. This could fit in use-cases such as students attempting to create a class diagram for a task and the LLM could re-create it but also provide feedback and propose fitting changes.

Choosing an output notation with enough training data available. Beside PlantUML, other textual notations, such as Umple or yuml, exist to define UML diagrams. Yet, informal testing has shown that the tested LLMs seemed to have the least trouble with PlantUML. While no thorough systematic tests were done on the different notations, we still see that the choice of the notation is important, as it affects the quality of the output. Furthermore, the better performance when using the PlantUML notation seems to correlate with the popularity of the notation, as PlantUML is considered one of the most popular UML tools available[5]. Arguably, this leads to more PlantUML data being available and thus, more PlantUML data contained in the LLMs' training data.

Keeping the human in the loop. Overall, while the Gemini models and CogVLM manage to provide a sort of base prototype for the given images, a lot of syntax and general adjustments are needed as compared to GPT-4V that usually provides an almost correct end-result in one go, without needing to complicate the prompt. We would argue that GPT-4V is the most suitable for the given task. Nevertheless, the minor errors still require human correction, thus, following

[5]https://modeling-languages.com/text-uml-tools-complete-list/

the human-in-the-loop paradigm is currently still necessary.

5.1. Threats to internal validity

Grading scheme. The grading scheme used to evaluate the output potentially may not completely reflect the performance of the LLMs. Aspects such as every mistake, excluding syntax errors, being worth 1 mistake could be seen as limited. Some might argue that forgetting an entire class might be worth more than just forgetting to add an attribute. Additionally, when syntax errors occurred, we marked the result of the attempt as an error. This resulted in outputs with correct syntax but with a lot of mistakes and hallucinations to be evaluated as a better result than those with syntax errors. One could question whether it is fair to say that something that does not compile but has less mistakes is worse than something that does compile but contains a lot of unwanted and wrong elements. In this case, we would argue that sometimes, creating something from scratch might prove easier than needing to change a lot of mistakes in a given prototype.

Scalability to larger diagrams. Finally, the used UML class diagrams were fairly small in nature and do not cover more complex software systems. Our observed results might not be applicable to larger UML class diagrams. We would still argue that the experiment itself is realistic in the sense that, in collaborative sessions, small or very abstract prototypes, or small components of a larger system are sketched and not big and complex UML class diagrams.

Chosen prompts. The chosen prompts might not have been the best ones for such a task, as is evident by Gemini's change in response to the given nonsensical UML class diagram when appending "Ignore the semantics." to the prompt. Another experiment could aim to determine the best prompt to create UML class diagrams out of images depending on the used LLM.

5.2. Threats to external validity

Nondeterministic nature of LLMs and sample size. The nondeterministic nature of LLMs might affect the validity of the interpretation of the results. This is further the case due to the small sample number of attempts used to evaluate the LLMs per prompt and per image. Although, ignoring the results with syntax errors, the number of errors in the generated PlantUML code per LLM seemed to vary very little, showing consistent performance for each example.

Generalization to other LLMs. Although the results of the experiment reflect the performance of the used visual LLMs, the results do not generalize to other visual LLMs. Indeed, the experiment itself has shown that some LLMs are more adequate for the explored task than others. This means, rather than saying that all visual LLMs are capable or not capable of transforming images of UML diagrams to a computer readable format, it always depends on the LLM. Thus, if another model is to be used, tests are required.

6. Tool support

In a first step towards low-modeling, the BESSER platform includes a UML class diagram image to PlantUML converter. BESSER [17], which focuses on the efficient development of smart software, implements an LLM interface that currently supports GPT-4V via the OpenAI API.

Beyond the transformation to PlantUML, the library also offers a transformation to the BESSER Universal Modeling Language (B-UML), that is the UML-inspired language of the BESSER low-code platform. It aims to leverage the advantages of UML while also having the freedom to integrate and extend the language with other (meta)models based on the requirements. The transformation to B-UML enables an immediate usage of the BESSER generations, effectively completing the pipeline from image, to computer readable format, and to software. The BESSER-examples[6] repository contains guidelines on how to use the image to UML functionality.

The used examples and results of the experiment can be found on the IMG2UML-Examples[7] repository. We encourage the community to use the examples from the repository with other LLMs and push the results to the repository.

7. Conclusion and future work

The inclusion of image processing in Large Language Models has further broadened the possibilities of how these can be used to assist humans in domains such as computer science. In this paper, we specifically explore their capabilities in software modeling as a tool to transform given drawn inputs into a concrete syntax. Results show that the quality of the replicated UML models can be good enough to use them as part of a modeling pipeline though results depend a lot on the LLM and prompt strategy used.

In the future, we plan to conduct qualitative studies, inviting human participants, software engineering experts or computer science students, to evaluate the usability and usefulness of the concept and results. Additionally, we would like to evaluate the ability of LLMs to transform other types of UML diagrams, such as state diagrams or use case diagrams, but also, more generally, non-UML diagrams often used in software engineering such as entity relationship diagrams. On the technical side, we want to improve the process by automatically checking some correctness properties of the recognized models as part of a pipeline that would automatically trigger new calls to the LLMs asking them to revise the detected mistakes. A second LLM could also be used for this purpose. This LLM-as-judge would evaluate the results and decide whether a new generation is needed. Finally, we would like to explore the usefulness of this transformation in an educational context where the LLM could play different roles. For instance, one could instruct the LLM to act as a mentor or teacher to the students using it to solve problems. Based on the received input from the students, the LLM could either immediately correct the given image and give feedback or even return the generated UML class diagram alongside with feedback on aspects it perceives as wrong and nudge the students to correct them, acting as an educational agent.

Acknowledgments

This project is supported by the Luxembourg National Research Fund (FNR) PEARL program, grant agreement 16544475 and the CLIMABOROUGH project, funded by the European Union under the grant agreement 101096464.

[6] https://github.com/BESSER-PEARL/BESSER-examples
[7] https://github.com/BESSER-PEARL/IMG2UML-Examples

References

[1] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, et al., A survey of large language models, 2023. arXiv:2303.18223.

[2] S. Barke, M. B. James, N. Polikarpova, Grounded copilot: How programmers interact with code-generating models, 2022. arXiv:2206.15000.

[3] H.-G. Fill, P. Fettke, J. Köpke, Conceptual modeling and large language models: Impressions from first experiments with chatgpt, Enterprise Modelling and Information Systems Architectures 18 (2023) 1–15. doi:10.18417/emisa.18.3.

[4] J. Cámara, J. Troya, L. Burgueño, A. Vallecillo, On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml, Softw. Syst. Model. 22 (2023) 781–793. doi:10.1007/s10270-023-01105-5.

[5] W. Wang, Q. Lv, W. Yu, W. Hong, J. Qi, Y. Wang, et al., Cogvlm: Visual expert for pretrained language models, 2023. arXiv:2311.03079.

[6] C. Si, Y. Zhang, Z. Yang, R. Liu, D. Yang, Design2code: How far are we from automating front-end engineering?, 2024. arXiv:2403.03163.

[7] E. Planas, J. Cabot, How are uml class diagrams built in practice? a usability study of two uml tools: Magicdraw and papyrus, Computer Standards & Interfaces 67 (2019) 103363. doi:10.1016/j.csi.2019.103363.

[8] J. Cabot, Low-modeling of software systems, 2024. arXiv:2402.18375.

[9] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, et al., Large language models for software engineering: Survey and open problems, 2023. arXiv:2310.03533.

[10] B. Karasneh, M. R. Chaudron, Img2uml: A system for extracting uml models from images, in: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications, 2013, pp. 134–137. doi:10.1109/SEAA.2013.45.

[11] F. Chen, L. Zhang, L. Xiaoli, N. Niu, Automatically recognizing the semantic elements from uml class diagram images, Journal of Systems and Software 193 (2022) 111431. doi:10.1016/j.jss.2022.111431.

[12] N. Best, J. Ott, E. Linstead, Exploring the efficacy of transfer learning in mining image-based software artifacts, 2020. arXiv:2003.01627.

[13] S. Shcherban, P. Liang, Z. Li, C. Yang, Multiclass classification of uml diagrams from images using deep learning, International Journal of Software Engineering 31 (2021). doi:10.1142/S0218194021400179.

[14] E. Lank, J. Thorley, S. Chen, D. Blostein, On-line recognition of uml diagrams, in: Proceedings of Sixth International Conference on Document Analysis and Recognition, 2001, pp. 356–360. doi:10.1109/ICDAR.2001.953813.

[15] T. Hammond, R. Davis, Tahuti: A geometrical sketch recognition system for uml class diagrams, AAAI Press (2002). doi:10.1145/1185657.1185786.

[16] H. Koç, A. M. Erdoğan, Y. Barjakly, S. Peker, Uml diagrams in software engineering research: A systematic literature review, Proceedings 74 (2021). doi:10.3390/proceedings2021074013.

[17] I. Alfonso, A. Conrardy, A. Sulejmani, A. Nirumand, F. Ul Haq, et al., Building BESSER: An open-source low-code platform, in: Enterprise, Business-Process and Information Systems Modeling, 2024, pp. 203–212. doi:doi.org/10.1007/978-3-031-61007-3_16.