A language workbench extension to generate conversational interfaces for domain-specific languages

Luigi Brandolini[1], Massimo Tisi[1] and Jean-Sebastien Sottet[2]

[1]*IMT Atlantique LS2N, UMR CNRS 6004, F-44307 Nantes, France*

[2]*Luxembourg Institute of Science and Technology (LIST), Esch-Sur-Alzette, Luxemburg*

Abstract

The integration of Large Language Models (LLMs) into software engineering has led to significant advancements, particularly in supporting development with well-established programming languages (e.g., Copilot for Java). LLMs face challenges when applied to domain-specific languages (DSLs), since often their training dataset does not include extensive sets of examples for these languages. This is also the case for languages that are either newly developed or under refinement.

In this paper, we introduce `langium-llm`, an extension of the Langium language workbench designed to facilitate interactions with LLMs during development with DSLs. Given a Langium-based DSL, we propose two approaches for the communication with the LLM: one based on the DSL concrete syntax and the other in its abstract syntax. Depending on the solution chosen, our extension generates a conversational interface that is aware of the grammar of the DSL concrete syntax or of the JSON Schema of its abstract syntax, enabling software engineers to collaborate with the LLM while developing in the DSL, using natural language instructions.

We first present an overview of the architecture underlying our proposed solutions. We then evaluate the effectiveness by assessing the LLM's ability to interpret and execute development instructions through the graphical chat interface. We especially focus on the responsiveness and accuracy of our LLM-based approach in editing programs written in different DSLs.

Keywords

Domain-specific languages, Large Language Models, Model-Driven Engineering

1. Introduction

The field of software engineering is increasingly benefiting from the widespread adoption and continuous advancements of Large Language Models (LLMs). These models have demonstrated remarkable potential in automating tasks traditionally performed manually by software engineers, thereby enhancing productivity and reducing error rates. As highlighted in [1], LLMs provide substantial support across all key areas of software engineering, including requirements engineering, design, code generation and completion, testing, maintenance, evolution, and deployment.

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

luigi.brandolini@imt-atlantique.fr (L. Brandolini); massimo.tisi@imt-atlantique.fr (M. Tisi); jean-sebastien.sottet@list.lu (J. Sottet)

0009-0005-4027-2208 (L. Brandolini); 0000-0001-7891-9138 (M. Tisi); 0000-0002-3071-6371 (J. Sottet)

A prominent area where LLMs excel is code generation and completion. By leveraging the predictive capabilities of LLMs, developers can efficiently write code with suggestions related to syntax, functions, and design patterns. As demonstrated in [2], models such as GPT exhibit strong proficiency in solving programming problems of low to medium complexity across established programming languages like Java, Python, and C++. In addition, this approach not only accelerates development cycles, but also minimizes syntax errors and bugs. For instance, in Java development LLMs can assist in generating boilerplate code, handling exceptions, and implementing object-oriented design principles. Furthermore, LLMs serve as invaluable tools for code review and debugging. LLMs also offer extensive support for learning and documentation, in such a way that developers can query LLMs for explanations of complex concepts, best practices, and library usage.

Furthermore, there is growing interest in the application of LLMs to code generation for DSLs, as discussed in [3]. However, despite their promising performance in mainstream programming languages, LLMs face significant challenges when applied to DSLs due to the limited availability of training data for these specialized languages. Consequently, the accuracy and relevance of LLM-generated output in such contexts are often hampered.

To address this challenge, we propose an extension to Langium[1], a language engineering open-source framework, to support the definition of specialized DSLs. This extension is complemented by a Visual Studio Code (VSCode) plugin featuring an LLM-powered chat facility. Once users have defined the grammar for their domain, they can interact with the LLM through natural language instructions to request model generation and modification, while the corresponding DSL grammar is automatically incorporated.

To this end, we propose two different solutions: In the first, the communication with the LLM is performed leveraging the langium concrete syntax, and the DSL langium grammar is directly propagated to the LLM context. In the second solution, the communication is performed leveraging the DSL abstract syntax represented in JSON, and the corresponding JSON Schema is propagated to the LLM context. In both cases the extension manipulates the user prompt and interacts with the LLM under the hood to ensure correct communication.

The contributions of the paper are:

- An approach to extend language workbenches to produce conversational interfaces that are tailored to the DSL under development.
- Two versions of the same approach, based respectively on concrete syntax and abstract syntax, and an initial comparison of the two methods.
- `langium-llm`, an extension of langium for conversational editing, publicly available as open source.[2]

To evaluate the effectiveness of the approach we use an original method based on a definition of a complex task, where another LLM is used to simulate a human modeler, using our LLM-based conversational interface.

The remainder of this paper is structured as follows. Section 2 introduces our proposed solution, providing a comprehensive overview of the linguistic processing through a graphical

[1]https://langium.org/
[2]https://github.com/NaoMod/langium-llm/

```
Given the following Langium grammar:
{DSLGrammar}

and this Input program:
{inputProgram}

{userPrompt}

I expect the response directly in the corresponding valid Langium textual syntax according to
↪  the grammar provided.
```

Listing 1: Prompt template (excerpt)

representation which captures its behavior and interactions. Section 3 evaluates the effectiveness and performance of the proposed solution through a series of experiments, with results analyzed using specific evaluation metrics to assess accuracy and efficiency. Section 4 discusses related work, positioning our contribution within the broader research landscape. Finally, Section 5 presents the conclusions, summarizing the findings and outlining potential directions for future research.

2. Proposed solution

We propose two different solutions for our DSL-specific conversational interface. In the first one, we directly include the grammar of the concrete syntax in the LLM context, while, in the second one, we include a JSON Schema representing the abstract syntax of the DSL. The aim is to compare the effectiveness and efficiency of the two proposed solutions in order to determine which one performs best. Specifically, we want to assess the performance of LLMs in managing abstract and concrete syntax.

The diagram in Figure 1a visually represents the different processes involved in grammar-based solution. The diagram is a directed hypergraph, where: nodes are artefacts; continuous arrows are processes that have possibly several inputs (i.e. the input artefacts), and a single output (i.e. the resulting artefact); dotted arrows represent conformance relationships between artefacts and their schema or grammar.

It consists of the following processes:

Prompt Generation. We automatically generate the prompt we will pass to the LLM by composing 4 input artefacts:

- *LLM Prompt Template* – Template used to dynamically generate prompts for language models. It acts as a blueprint that combines static text with placeholders for variables. An excerpt of the template is shown in Listing 1.
- *DSL Grammar* – The DSL grammar in langium format.[3]
- *DSL Input Program: concrete syntax* – A DSL program written by the user, and currently displayed in the open editor.

[3]https://langium.org/docs/reference/grammar-language/

```
The current response:
{LLMResponse}

contains the following errors:
{validationErrors}

So it doesn't represent a correct Langium model according to its grammar.

Please fix this model and return only a correct one for the current request:

{userPrompt}
```

Listing 2: Prompt update template (excerpt)

- *User Prompt* – The natural language message expressed by the user in the conversational interface.

The resulting *LLM prompt* is obtained by replacing the placeholders inside the LLM Prompt Template, with the related artifacts.

LLM Request. In response to the *LLM Prompt* we obtain from the LLM an *LLM Response* message, possibly containing a DSL program in concrete syntax.

Syntax Checking. The Langium parser checks for syntactical correctness of the DSL program possibly included in the response. If the program is syntactically correct, then it is returned to the user (*DSL Output program: concrete syntax* node). If some errors are detected, we gather the parsing error messages returned by the syntactic parser (*Validation Errors* node). Note that in the whole process we only allow a syntactic check to be repeated a finite number of times (5 by default) for the same user prompt. If the LLM won't be able to provide a correct response within this number of trials, an error message will be provided to the user.

Update Prompt Generation. If some errors are found, we build an *LLM Update Prompt* asking the LLM to update its previous response. The update prompt is generated expanding an *LLM Update Prompt Template*. It includes the original *LLM Prompt* and the indication of the *Validation Errors* found in the previous response. An excerpt of the template is shown in Listing 2.

The diagram in the Figure 1b visually represents, instead, the solution based on the abstract syntax in JSON. We describe the new processes w.r.t. the solution based on concrete syntax:

JSON Schema Generation. A *JSON schema* of the abstract syntax is generated from the Langium *DSL grammar*. Indeed, a Langium grammar contains also information about the structure of the DSL abstract syntax. By default, langium will create an abstract syntax class per grammar rule. The assignments (e.g. `power=KWH`) define the attributes of these classes. Cross-references (e.g., `fromStop=[Stop:ID]`) define the associations between these classes. A syntax is also available to describe in the grammar file a different structure for the abstract syntax.[4] Using this information, Langium generates an abstract

[4]For more information see https://langium.org/docs/reference/semantic-model/

syntax as TypeScript classes. A JSON Schema, then, can be generated automatically by analyzing these classes.[5]

Parsing. Langium parses the input program (*DSL Input program: concrete syntax*) and produces an abstract syntax tree that is serialized as an instance of the JSON Schema (*DSL Input program: JSON format*). From here on, we communicate with the LLM exclusively about this JSON data structure, possibly asking the LLM to modify it.

JSON Validation. We use a JSON Schema validator[6] to check if the JSON returned in the *LLM Response* represents a correct abstract syntax tree, conforming with the DSL *JSON Schema*. If there are *Validation Errors*, we gather them in textual format. In the following steps these errors will be passed to the LLM, requesting to correct them, exactly as we did for the grammar-based version of the tool. Again, in the whole process we only allow validation to be repeated a finite number of times (5 by default) for the same user prompt. If the LLM won't be able to provide a correct response within this number of trials, an error message will be provided to the user.

Serialization. In order to be displayed in the user's editor, the LLM response in JSON format is serialized in the concrete syntax, by means of the DSL formatter.

3. Experimentation

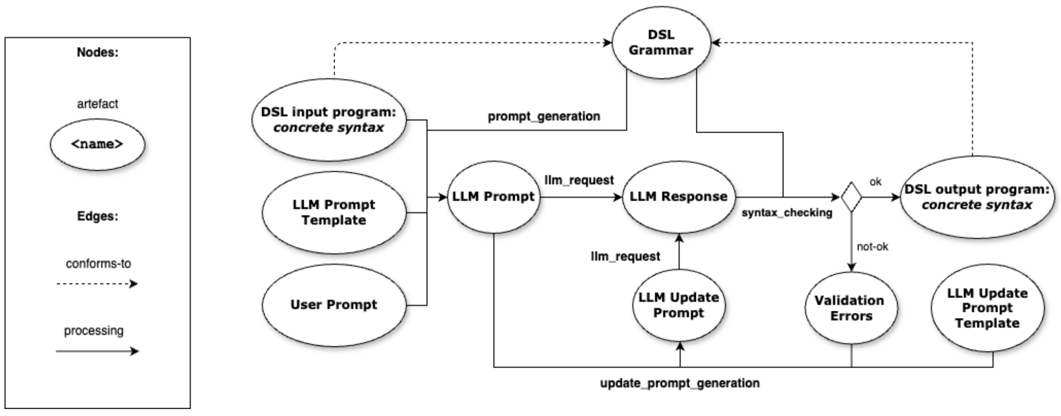
We conduct an experimentation to assess the support that LLMs can provide to software engineers and developers in the definition of software models. To this end, the `langium-llm` framework was employed to generate editors and conversational interfaces for multiple DSLs. The selected LLM for these experiments was Google Gemini, using the `gemini-flash-2.0` model.

The complete dataset, scripts and results of this experimentation are available in the `langium-llm` repository.[7]

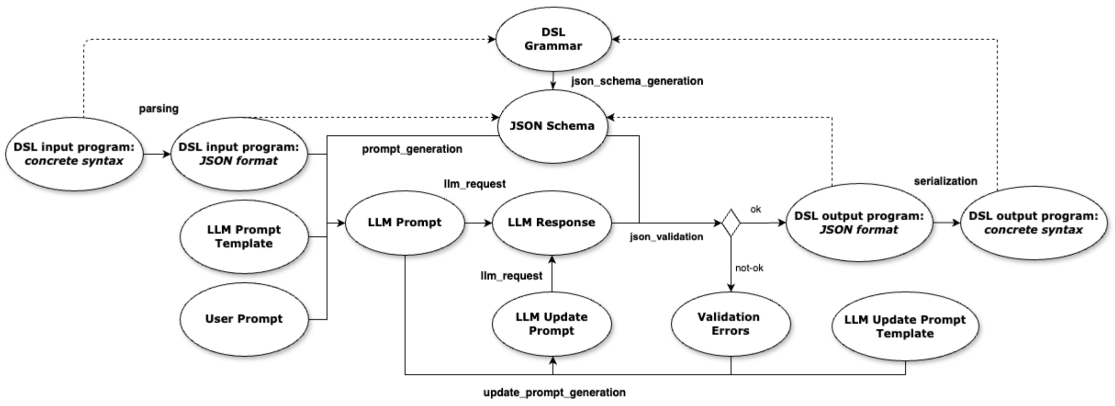
3.1. Dataset

Regarding the selected languages, we covered different application domains. We began with the `"State Machine"` DSL, sourced from the official Langium project GitHub repository. To ensure a consistent and diversified benchmark, we also manually developed three additional custom DSLs: `"Urban transportation"`, `"Cultural Heritage"` and `"Human Body"`. The Urban Transportation DSL, allows users to define electric buses operating along predefined routes, between bus stops. Buses are associated with their residual charge, routes with an expected consumption and reloaders with their maximum reloading power. The `Cultural Heritage` DSL, models heritage assets through properties such as typology, material, conservation status, and monitoring parameters. Once an asset is defined, it is also possible to configure a restoration intervention. The `Human Body` DSL, on the other hand, allows the definition of a set of organs, interconnected and annotated with a specific health status.

[5]For instance, with the tool https://github.com/vega/ts-json-schema-generator
[6]https://ajv.js.org/
[7]https://github.com/NaoMod/langium-llm/blob/main/examples-llm



(a) Approach based on concrete syntax



(b) Approach based on abstract syntax

Figure 1: Processes in the proposed approach

For each DSL we generated 32 sample models by prompting OpenAI GPT-4o, each one with a size between 90 and 200 tokens.[8]

The first columns of Table 1 describe quantitatively the dataset. The DSLs have between 5 and 13 production rules, and the generated models have, depending of the DSL an average size betwen 129 and 155 tokens.

3.2. Experimentation process

By our approach we generate conversational interfaces that allow users to perform edits on a given DSL. To evaluate the effectiveness of the conversational interfaces we measure, as customary for LLM applications, the success rate in a given task. We devise an original task that has the following characteristics: it is complex, can be automatically executed, and can be easily replicated for several DSLs.

[8]Number of tokens is the most common size measure for LLM inputs https://en.wikipedia.org/wiki/Large_language_model#Tokenization

Language	Rules	Tokens	Context	Convergence rate	Error rate	Rounds	Rounds St.Dev.	Duration (ms)
Urban transportation	7	129.55	-	68.75%	25.00%	11.91	2.98	15,847
			Grammar	87.50%	12.50%	11.57	5.21	15,986
			JSON Schema	100.00%	0.00%	14.56	3.48	26,088
Cultural heritage	13	155.59	-	50.00%	50.00%	14.38	3.81	24,053
			Grammar	56.25%	43.75%	13.44	6.27	22,640
Human body	9	139.26	-	56.25%	43.75%	11.00	2.55	22,476
			Grammar	81.25%	12.50%	12.08	3.25	28,139
State machine	5	127.56	-	68.75%	25.00%	9.73	7.14	12,401
			Grammar	75.00%	18.75%	11.75	7.41	16,123

Table 1
Experiments outcomes. All measures are averages over the 16 repetitions of the experiment.

We choose a development task where a human user needs to edit a program in a given DSL from an initial state to a final state. For experimentation, we use an LLM (again `gemini-flash-2.0`, in a different session) to simulate the human user, that wants to exploit the conversational interface to perform the development task. In the following we refer to this LLM as `LLM2`. Given a DSL, we automatically generate two DSL programs and consider them the initial state and final state of this development task. The human user, simulated by `LLM2`, produces a sequence of prompts to edit the file from the initial state, with the objective of arriving to the final state. These prompts are given to our tool that actually performs the edits on the file. The task is achieved if the simulated human user is able to arrive to the final state. In the following, we call the success rate in this task as *convergence rate*.

This task can fail for two reasons: 1) our tool is not able to produce a syntactically correct output in one of the steps (as a reminder, at each step our tool asks the LLM to correct its output for a maximum of 5 times) 2) the maximum number of iterations is not enough for the pair of LLMs to reach the final state. We call the first failure an *error*, and we calculate an *error rate* as the percentage of task executions that end in an error.

The implemented approach follows an iterative process: a couple of programs are selected as `m0` and `mFinal`. `LLM2` analyzes the current state of `m0` and detects a single difference relative to `mFinal`. This difference is formulated as an instruction in natural language (e.g. `Add a Reloader`) and communicated to the conversation interface (indicated as `LLM1`), which applies the identified change to the current program. This process repeats until no further discrepancies remain between the two programs, ultimately achieving convergence. A limit of 50 rounds (`MAX_LIMIT`) has been fixed, and if it is passed the task is considered as failed. To determine this limit we used an IQR method to compute the upper bound for outliers in all the tasks, and we chose a value that exceeds all the upper bounds.

Depending on the case we perform the same experiment with different versions of the conversational interface: the one aware on the grammar of the concrete syntax, the one aware of the JSON Schema of the abstract syntax, and a baseline version where neither the grammar or the JSON Schema has been added to the context of the LLM.

3.3. Results and Discussion

The outcomes of the experiment are summarized in Table 1. In the `Context` column we indicate if the experiment was performed with the grammar-based version (`Grammar`), the json-based version (`JSON Schema`), or the baseline version (`-`). The last columns reports on the evaluation metrics adopted: the convergence rate (`Convergence rate`), the error rate (`Error rate`), the average number of rounds in successful cases (`Rounds`), the standard deviation of such number of rounds (`Rounds St.Dev.`), and the average task duration expressed in milliseconds (`Duration (ms)`).

The baseline version consistently demonstrates lower convergence rates and varying degrees of efficiency across domains. For example, in the ”Urban transportation (no grammar)” DSL, the convergence rate drops to 68.75%, with an average of 11.91 rounds and 15,847 ms in execution time. Similarly, for the ”Cultural heritage (no grammar)” DSL, convergence is further reduced to 50.00%, with 14.38 rounds and a duration of 24,053 ms. The ”Human body (no grammar)” DSL yields 56.25% convergence in 11.00 rounds, while ”State machine (no grammar)” achieves 68.75% convergence in 9.73 rounds and 12,401 ms.

The inclusion of the grammar significantly improves both convergence rates and, in some cases, execution efficiency. For instance, comparing the ”Urban transportation” results with and without grammar reveals a substantial increase in convergence from 68.75% to 87.5%, with a comparable number of rounds (from 11.91 to 11.57) and similar execution time. In the ”Cultural heritage” domain, grammar improves convergence from 50.00% to 56.25%, with a slight reduction in the number of rounds (from 14.38 to 13.44) and duration (from 24,053 ms to 22,640 ms). The ”Human body” domain experiences a notable increase in convergence from 56.25% to 81.25%, despite a slight increase in the number of rounds and duration. Similarly, in the ”State machine” domain, convergence improves from 68.75% to 75.00%, with modest increase in rounds and duration. These improvements suggest that the presence of grammatical structure contributes to higher task success rates while maintaining competitive efficiency.

The use of a structured JSON format further boosts performance, especially in terms of convergence. In the only domain tested with ”Urban transportation (JSON)”, the convergence rate reaches 100%, surpassing both the grammar-only (87.5%) and no-grammar (68.75%) conditions. However, this increase comes at the cost of longer convergence times: the number of rounds increases to 14.56, and execution duration rises to 26,088 ms. This result indicates that, while JSON introduces additional overhead, it substantially enhances the reliability and determinism of the output, fully eliminating convergence failures in this setting. Therefore, the structured format serves as a robust method to ensure completeness at the expense of time.

The `Error rate` column shows that in almost all cases, the reason for not reaching convergence is in the LLM not being able to produce a syntactically correct output in one of the steps. Asking the LLM to correct its output for 5 times seems to fix the problem in some cases (hence the increase in convergence rate), but to be still insufficient. Notice that this problem seems to be much better addressed in the JSON-based version (reaching 100% convergence). This denotes a very good capability of Gemini to manipulate JSON.

4. Related Work

Some surveys are analysing the LLM in software engineering context. Amongst them, we consider [4] which offers insights into the synergic use of LLMs in model-driven engineering. It outlines various strategies for enhancing prompts and highlights related work on using natural language prompts for model querying and creation, constraint generation, and DSL handling.

None of these tools use grammars as input as we do, but some rely on UML input models instead.

Our approach leverages a technique similar to the grammar masking strategy presented in [5], which steers the LLM towards outputs that adhere to the specified grammar rules. The primary goal of our contribution is to embed a similar technique into a standard language workbench, to automatically apply it to new DSLs. We also start experimenting with JSON masking for abstract syntax, that seems to give promising results.

In [5], the authors present a method that integrates different approaches (few-shot learning, guidance-ai tool, grammar masking) to evaluate the syntactic validity of a target domain-specific language (DSL). This approach is close to our solution, but requires relying on the guidance framework[9] to help interpreting the initial grammar, as well as using few shot learning (i.e., providing a small set of examples). On the contrary, we provided a simpler approach either only providing the grammar or using a standard intermediary representation (i.e., JSON).

In [6], LLMs can assist in defining the semantics of DSMLs, potentially streamlining the development process within specific application domains, notably facilitating the discussion with domain expert. The authors highlighted some challenges, notably ensuring the quality of the LLM produced artefacts and the need to enhance LLM with other source knowledge like RAG approaches.

[7] introduces a method called "grammar prompting" to enhance LLMs ability to generate outputs in the targeted DSL. This approach involves augmenting each prompt with a specialized grammar (i.e., specialised according to the prompt), that is minimally sufficient for generating the desired output. During inference, it generates the output according to the rules of the specialised grammar ensuring limited number of error. It provides an interesting approach, by generating a dedicated grammar, to enhance the result. Nevertheless it requires more complex user prompting than simple text as we provided.

[8] provides a web application that allows users to interact with the LLM via a web-based chat interface. While this solution includes a similar LLM facility, our work focuses on analyzing LLM outputs and conducting effectiveness tests with performance benchmarks, with additional attention to JSON as a potential format for LLM message exchange.

Beyond the work specifically targetting DSL as output, we can review [9] that is dedicated to a transpilation approach. It lifts existing code based on specialized DSL (dedicated to optimization, parallelization, etc.). The main focus of their work is the validation of the proposed code transformation: they propose a proof for correctness of the produced code. The authors propose to remove the "understanding" of the DSL for the machine, using an internal representation of the DSL concepts in general purpose language: Python. This is what we did with the JSON version of our experimentations. We also implemented some validation of the proposition made by the LLM ensuring a dialog, but with the aim to support any Langium made DSL we also could not rely on clearly defined semantic to ensure the full validity of the LLM response.

W.r.t. all these approaches, we propose a direct integration into an existing language workbench (i.e., Langium), to generate tailored conversational interfaces. Moreover, our experimentation, in Section 3, shows a slightly different focus: we are interested in performing more semantics prompts (e.g., `add a Reloader`) that are applied more naturally to the abstract syntax

[9]https://github.com/guidance-ai/guidance

of the program.

5. Conclusion and Future Work

In this work, we extended a language workbench, to generate along with domain-specific editors also conversational interfaces with enhanced syntactic checks. Ultimately, our research has highlighted the role of DSL grammars in structuring interactions with LLMs to ensure the generation of syntactically correct outputs. Moreover, we demonstrated that the adoption of standardized, machine-readable formats like JSON, despite introducing a processing overhead, substantially enhances the effectiveness and reliability of the proposed approach.

Our experience in building the tool highlighted the importance of prompt template design in guiding the LLM's attention. In one case, for instance, it was observed that explicitly emphasizing the differences between the input models within the prompt was essential for the LLM to accurately detect discrepancies. Without such clarification, the model often failed to recognize subtle variations, particularly those involving minor textual differences in properties such as descriptions.

In future work, we will first extend this initial experimentation, considering a bigger set of DSLs and models and applying the JSON Schema approach to all of them.

We will address the limitations of the current approach. Especially, when users write a syntactically incorrect program in the editor and query the LLM about it, our current JSON-based tool behaves erratically. In fact the langium parser silently translate incorrect programs to JSON syntactic trees that have significant differences with the program. In such cases the LLM is only aware of the JSON version, and may apply erroneously some user-requested updates.

Finally we consider this work as a first step for producing LLM-based tools that perform full program transformations described in conversational style. While we experiment here only on atomic changes, full transformations would not be effectively applied in one shot, and may require prompt engineering techniques to decompose them in manageable steps. We will investigate such techniques in the near future.

Acknowledgments

This project has received co-funding from the European Union's Horizon Europe Research and Innovation Programme under Grant Agreement No. 101126644. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT for grammar and spell checking. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. H. M. L. S. S. S. Y. J. M. Z. Angela Fan, Beliz Gokkaya, Large language models for software engineering: Survey and open problems (2023).

[2] V. H. P. T. N. Alessio Bucaioni, Hampus Ekedahla, Programming with chatgpt: How far can we go? (2024).

[3] F. F. Sathvik Joel, Jie Jw Wu, A survey on llm-based code generation for low-resource and domain-specific programming languages, arXiv preprint arXiv:2410.03981 (2024).

[4] J. Di Rocco, D. Di Ruscio, C. Di Sipio, P. T. Nguyen, R. Rubei, On the use of large language models in model-driven engineering, Software and Systems Modeling (2025) 1–26.

[5] L. Netz, J. Reimer, B. Rumpe, Using grammar masking to ensure syntactic validity in llm-based modeling tasks, in: 27th International Conference on Model Driven Engineering Languages and Systems, MODELS Companion '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 115–122. URL: https://doi.org/10.1145/3652620.3687805. doi:10.1145/3652620.3687805.

[6] C. Di Sipio, R. Rubei, J. Di Rocco, D. Di Ruscio, L. Iovino, On the use of llms to support the development of domain-specific modeling languages, in: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS Companion '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 596–601. URL: https://doi.org/10.1145/3652620.3687808. doi:10.1145/3652620.3687808.

[7] B. Wang, Z. Wang, X. Wang, Y. Cao, R. A Saurous, Y. Kim, Grammar prompting for domain-specific language generation with large language models, Advances in Neural Information Processing Systems 36 (2023) 65030–65055.

[8] V. Lamas, M. R. Luaces, D. Garcia-Gonzalez, Dsl-xpert: Llm-driven generic dsl code generation, in: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, 2024, pp. 16–20.

[9] S. Bhatia, J. Qiu, N. Hasabnis, S. A. Seshia, A. Cheung, Verified code transpilation with LLMs, in: The Thirty-eighth Annual Conference on Neural Information Processing Systems, 2024. URL: https://openreview.net/forum?id=spwE9sLrfg.