Exploring the use of large language models in domain-specific language development: an experience report

Jordi Cabot[1,2,*]

[1]Luxembourg Institute of Science and Technology, Esch-sur-Alzette, Luxembourg
[2]University of Luxembourg, Esch-sur-Alzette, Luxembourg

Abstract

Large language models (LLMs) are increasingly used in many software-related tasks. In this paper, we explore whether LLMs can also be used to create a domain-specific language (DSL) from scratch. This includes both the language abstract syntax and two types of concrete syntaxes: textual and graphical. Results are promising, showing the potential of LLMs to assist in the creation of a DSL even if, so far, they cannot be left alone and need some guidance in order to produce optimal results.

Keywords

LLM, DSL, Language Engineering, Vibe coding, Metamodel, Grammar, Notation, AI

1. Introduction

Domain-Specific Languages (DSLs) facilitate the creation of models and software artifacts for specific domains, helping domain experts express their knowledge and requirements using a vocabulary that is closer to their own.

On the one hand, the creation of a DSL is a complex task often left to language engineers. While their expertise guarantees a high-quality DSL that satisfies user needs, it is still a time-consuming task. Additionally, language engineers may not be available or not be able to take on the task within the time (or budget) constraints.

On the other hand, Large Language Models (LLMs) are increasingly used in many software-related tasks [1]. They are able to generate code, documentation, tests, and so on with an acceptable level of quality. Therefore, it is fair to wonder whether the same is true for DSL-related tasks.

In this sense, this paper presents a concrete experiment in using an LLM to create a DSL from scratch, including both the abstract syntax and two types of notation: textual and graphical. Based on this experiment and the author's experience in other (smaller) LLM-driven DSL tasks, the paper draws some discussion points on the benefits and challenges of using LLMs in language engineering.

Surprisingly enough, this is not a topic that has been thoroughly studied so far. Although the use of DSLs to help in the configuration and management of LLMs is a rather popular topic (e.g., [2, 3, 4, 5, 6]), the use of LLMs to create full DSLs is still a rather unexplored topic. Existing works focus on the use of LLMs to elicit the language semantics [7], to create code-generation artefacts [8] or to derive models from natural language documents [9, 10, 11, 12, 13]. Even if works in this latter group target models and not DSLs, they are still relevant as they could be directly used to create the DSL metamodel, typically expressed as a UML class diagram.

The rest of the paper is structured as follows. The next section details the experiment, describing the use of LLMs to create each of the main DSL components. Based on this, Section 3 discusses the results

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

*Corresponding author.
jordi.cabot@list.lu (J. Cabot)
https://jordicabot.com/ (J. Cabot)
0000-0002-0877-7063 (J. Cabot)

and adds some additional reflections. Finally, Section 4 wraps up the paper with some final conclusions.

2. Creating a DSL with an LLM: an experiment

There are many tools and language workbenches to create a DSL, where the choice often depends on the core notation of the DSL[1] (graphical, textual or projectional) and the type of DSL (external or internal [14]). But even if each tool offers some type of support (from syntax highlighting to rule-based autocompletion), most of the burden still falls on the language engineer.

In this section, we aim to do the opposite, free the language engineer and push the LLM to create a DSL from scratch. Each subsection covers a specific DSL component and the LLM's role in its creation.

Think of this experiment as an attempt to replicate the vibe coding[2] trend at the DSL level and aim to "vibe DSLing" a brand new DSL.

As an example DSL, we will be using the FUNDING specification[3] aimed to specify how to manage sponsorship money in an open source project. While simple, this DSL is representative of many other similar DSLs focused on facilitating the precise specification of structured domain data and is rooted in a real-world use case.

For the experimentation, we have used Claude Sonnet 4[4] as LLM and Cursor[5] as the IDE in charge of interacting with the LLM in agent mode. The initial prompt was rather simple.

Listing 1: Initial prompt explaining to the LLM the tasks required to create the DSL

```
We are going to create a new Domain-Specific Language: In
    particular we will create a "Funding" DSL to help open source
    maintainers specify how to manage the sponsorships they get
    from github. You can read more about this concept here @Web
    @https://docs.github.com/en/repositories/managing-your-
    repositorys-settings-and-features/customizing-your-repository/
    displaying-a-sponsor-button-in-your-repository

Your task will be to create:

1 - A set of Python classes to represent the metamodel (also known
     as Abstract Syntax) of the FUNDING dsl. These concepts are
    inspired by the explanation of the language in the above page.
    Identify the main concepts and their relationships. Represent
    the concepts as python classes and the relationships as
    attributes linking objects of the related topics. To help me
    verify the metamodel you propose, create a graphviz
    representation I can visualize.
2 - A concrete textual syntax users can use to define the models
    corresponding to the Funding DSL. Files using this concrete
    syntax will be parsed (and if successful) the corresponding
    Python objects representing the textual model as instances of
    the Funding metamodel will be instantiated. Use ANTLR for this
    part
```

[1] A DSL can have several notations, each one targeting a different user profile. Nevertheless, most DSLs have just one, or a clearly dominant one, with others being just a view over that one

[2] Vibe coding is the practice of creating apps, by just telling an AI program what you want, and letting it create the product for you. The coder does not need to understand how or why the code works, and has to accept that a number of bugs and glitches will be present.

[3] https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/displaying-a-sponsor-button-in-your-repository#about-funding-files

[4] https://www.anthropic.com/claude/sonnet

[5] https://www.cursor.com/

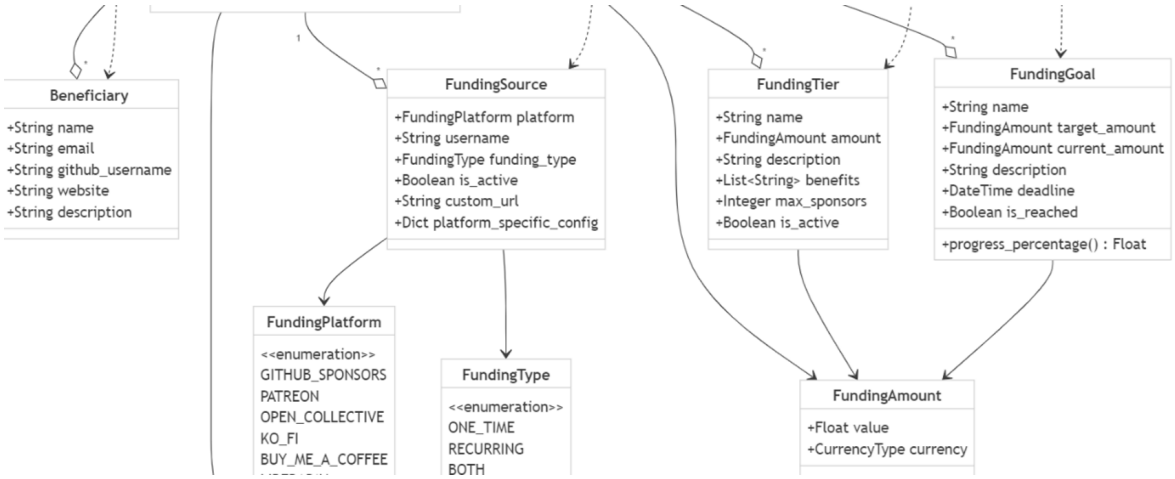

Figure 1: Metamodel for the FUNDING DSL, visualized using a (partial) Graphivz rendering created by the LLM itself.

3 – A graphical notation to enable users to define the models corresponding to the Funding DSL in a graphical modeling environment using the icons of the graphical notation. The environment will then create the corresponding Python objects representing the textual model as instances of the metamodel will be instantiated
4 – An export functionality from a model (expressed as the set of Python objects instantiating the model) that generates funding.yml files from a Funding DSL model.

Let's complete the tasks step by step and make sure one step is correct before moving to the next one

The full results of the experiment are available in a GitHub repository[6]. Next subsections go over its elements in more detail.

2.1. Creating the abstract syntax

As stated in the prompt, the first task is the creation of the DSL abstract syntax, i.e. its metamodel, as a set of Python classes following the internal DSL paradigm.

Overall, the results of this task were quite satisfactory. The LLM was able to create a metamodel that was close to the one I had in mind. The This satisfactory result was to be expected as this task (i.e. the creation of a structural model) is the most popular one and therefore the one LLMs have seen the most in the training data.

For the same reason, limitations are also well-known and are related to the tendency of the LLM to add to the models additional concepts (based on its own semantic knowledge) beyond those mentioned in the input document.

Indeed, in this case, the main differences between my ideal metamodel and the one generated by the LLM were these additions of elements (attributes in this case) that were not necessary (but the LLM felt that "should be there") and the inclusion of technical concepts, such as the visitors pattern elements, which were not required at this stage, though they will be useful in the next phases.

[6] https://github.com/jcabot/funding-dsl

2.2. Creating a textual concrete syntax

As a next step, the LLM was asked to create a textual concrete syntax for the DSL. In fact, two[7], one based on ANTLR[8] and another one based on TextX[9]. Beyond the grammar and parser, the LLM was also asked to write a transformation to express the parsed ASTs as instances of the classes of the DSL metamodel.

While the LLM was quickly successful with the ANTLR grammar (as there are plenty of example ANTLR grammars), it took several attempts to create a TextX one that was comparable to the ANTLR one. Here the agent mode of Cursor was particularly useful as it allowed to quickly iterate on the grammar and the transformation based on a couple of simple examples. The results were automatically compared with the ones obtained with the ANTLR version until the examples were giving as result the exact same model in both cases.

The LLM, in fact, declared that it was not familiar at all with TextX so we had to point to the official TextX page in the prompt and let the LLM explore and get familiar with the library before being able to tackle this part.

Listing 2: Example FUNDING file expressed using the created ANTLR grammar. The grammar, parsers and transformations are available in the GitHub repository

```
funding "AwesomeLib" {
  description "A comprehensive library for awesome functionality"
  currency USD
  min_amount 1.00
  max_amount 1000.00

  // Define who receives the funding
  beneficiaries {
      beneficiary "Alice Johnson" {
          email "alice@example.com"
          github "alicej"
          website "https://alicej.dev"
          description "Lead maintainer and project founder"
      }

      beneficiary "Bob Smith" {
          github "bobsmith"
          description "Core contributor and documentation
              maintainer"
      }
  }

  // Define funding sources/platforms
  sources {
      github_sponsors "alicej" {
          type both
          active true
      }
```

[7]Technically speaking, you could even say three, as we also asked the LLM to create an export functionality that would transform the funding model into a funding.yaml file with the syntax expected by GitHub

[8]https://antlr.org/

[9]https://textx.github.io/textX/

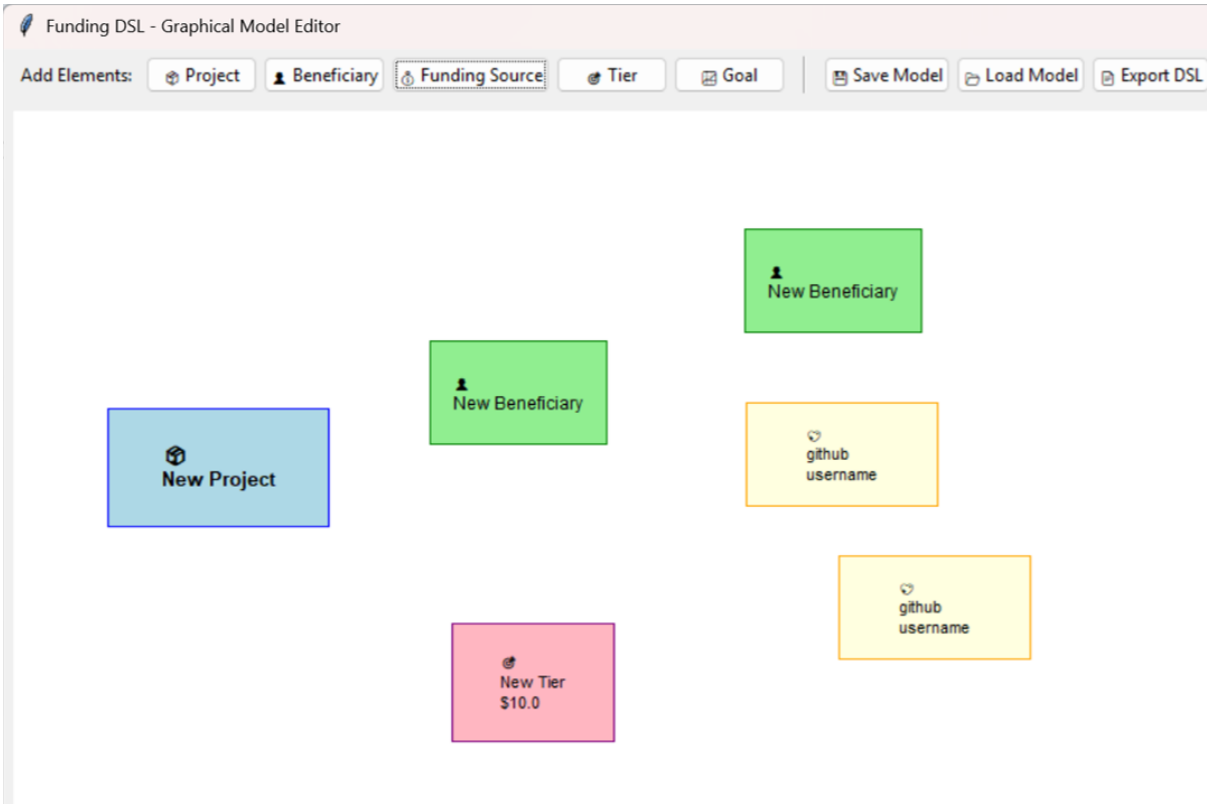

Figure 2: Example of a graphical funding model created on a Tkinter-based editor.

2.3. Creating a graphical concrete syntax

The last step involved the creation of a graphical notation, together with an actual modeling editor, that could be used to create graphical funding models and store them as instances of the DSL metamodel. There was no indication of the framework to use to create the editor itself. Nor any indication of the icons to use to represent the DSL concepts.

The LLM was able to comply with the request even if the quality of the editor and the choice of icons was rather questionable.

I did a second attempt, asking the LLM to improve the icons following Moody's recommendations [15]. The results were not much better. You can judge by yourself the evolution between Fig. 2 and 3.

It is worth highlighting that the editor was functional. Which in itself is a major accomplishment considering the LLM had no indication of the framework to use to create it. But clearly, the quality of the editor is not good enough to be used by the end-users. And the choice of the icons does not really resemble the concepts behind the DSL or any other type of decision that could be justified based on the DSL semantics.

Overall, this task was too creative and advanced for the current capabilities of the LLM. At least unless you provide many more precise instructions on how to carry it, narrowing down its options.

3. Discussion

Overall, the LLM managed to create a fully-fledged and functional DSL from scratch. Though the quality of each component was very diverse, depending on the freedom the LLM had to achieve the task. This section complements the experiment with some additional reflections and discussion points.

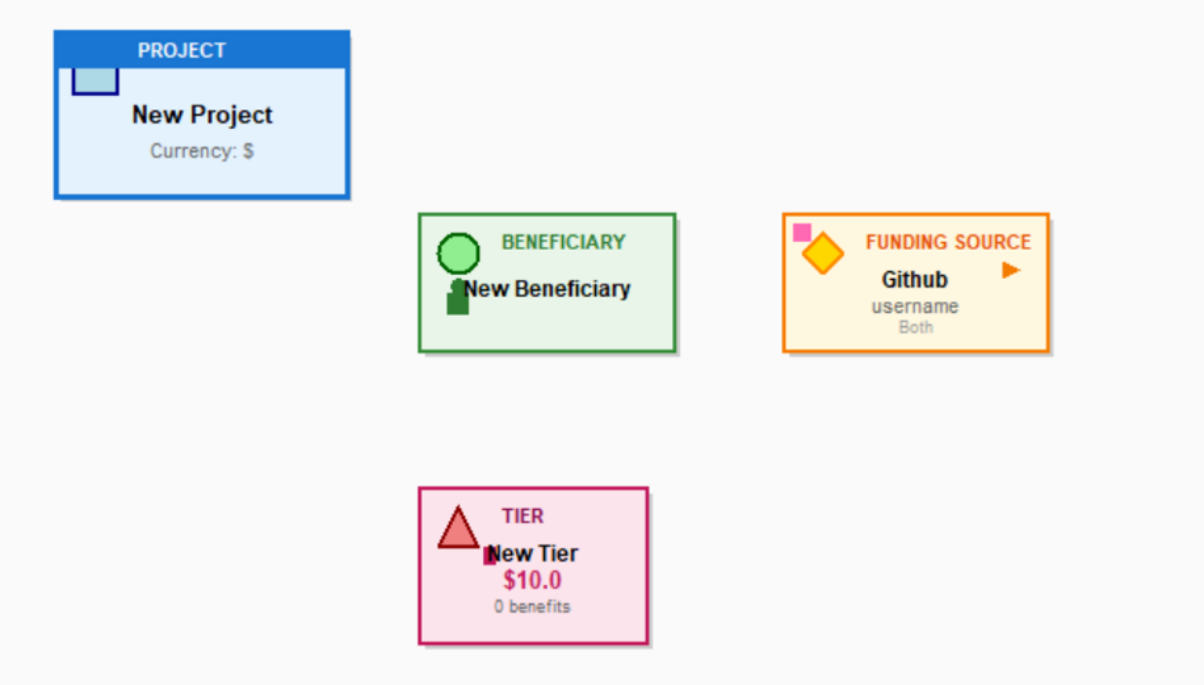

Figure 3: Example of a graphical funding model with the icons slightly improved following Moody's recommendations.

3.1. Uncertainty and traceability

When agents and LLMs are part of the language engineering process, it is more important than ever to keep full traceability of the language evolution. You should be able to explain who proposed and approved the changes. Both for the abstract and concrete syntaxes. Similar to what we proposed in Collaboro [16], you can link to the language concepts the list of change proposals for that element, the user (human or agent) behind the change and the full list of users that voted in favor of the change. When such traceability support is not natively available, we can still try to recover some of it by analyzing the logs of the conversation between the LLM and the user, kind of a process mining exercise applied to the conversation.

Traceability is obviously linked to the inherent uncertainty of the LLM. Each agent suggestion will come with a certain level of confidence and need to keep this confidence / certainty score together with the suggestion. This confidence level could be provided by the agent itself as part of its self-reporting but it could also be estimated based on benchmarks of LLMs on specific DSL tasks.

3.2. Look out for biases

LLMs are full of biases (racism, sexism, ageism, etc.) and this can affect any task where an LLM is involved. This is the case even for code generation tasks [17]. And the same can happen when using LLMs in language creation tasks. For instance, examples of biases that we should be checking for are:

- **Choice of the icons**: The visual representation of concepts may reflect cultural or gender biases.
- **Names of the DSL primitives**: Naming conventions might favor certain languages, cultures, or stereotypes.
- **Well-formedness rules of the DSL**: Constraints may unintentionally exclude valid use cases or encode biased assumptions in the set of models that can be created.
- **Example scenarios and documentation**: Example scenarios for the DSL should show diversity of profiles.
- **Default values and options**: The default values suggested by the LLM may not be neutral.

3.3. Help the LLM

We have seen that the LLM has limitations when it comes to the most creative aspects of the task, such as coming up with a possible set of graphical icons. Therefore, to optimize the use of the LLM, we need to help it to be more helpful. For instance, by suggesting libraries or frameworks to use. Or by pointing to sets of icons to be used as inspiration. The more specific we can be, the better. The LLM excels at implementing the technical low-level details, so if we can take care of the conceptual aspects, including the architecture of the DSL itself, the results will improve noticeably.

One aspect that could make a significant difference is to describe your DSL as a Product Requirement Document (PRD). PRDs are template documents often used in LLM tasks and, thus, well-known to LLMs. A PRD for a DSL could guide the LLM through the DSL creation tasks and help to create a DSL more aligned with the user's requirements. For instance, the PRD could include the purpose of the DSL, the list of domain concepts to support, the profile of the target users, and so on.

If necessary, explicitly point to the parts of the DSL that have already been created by pointing to the online docs or the DSL repository. You can even provide the DSL itself as part of the prompt, e.g. when you want to generate a concrete syntax for an existing metamodel.

3.4. An LLM can be useful beyond assisting in the language creation process

So far, we have focused on the use of LLMs to assist in the creation of a DSL. But LLMs can be useful beyond that. Some other language tasks where LLMs can be useful are:

- **Simulation**. The LLM can be used to create examples of diverse DSL instances to simulate and exercise different components of the DSL.
- **Role Playing**. The LLM can play the role of a user trying to use the DSL. This could help to detect parts of the DSL that are not intuitive to use [18] Sure, a human tester would be even better but with the LLM you can fully automate the process.
- **Tool infrastructure**. Even if you want to keep the core parts of the DSL "human-based", you could still use the DSL to generate a lot of the tooling infrastructure and auxiliary components (such as the code-generators, docs, tests, tutorials...) that are less critical.
- **Language evolution**. The LLM can be very effective when extending an existing DSL as in this case, all the critical decisions are already taken and the LLM just needs to imitate and slightly extend what is already there. As an example, for the BESSER language[19][10], I was able to quickly extend the metamodel with one new attribute and the visual editor with a new visual field to let users enter the value of the new attribute. In both cases, I could just point to an existing attribute and ask the LLM to copy the existing code and slightly modify it to add the new attribute.
- **Alternative types of concrete syntaxes**. Beyond graphical and textual syntaxes, we see more and more the use of chatbots as an alternative syntax for a DSL [20]. In this scenario, a conversational syntax instantiates a chatbot with whom users talk to create DSL instances. Another example is form-based syntaxes, where the DSL instance is created based on the users' answers to the form questions, similar to the concept of CRUD-like scaffoldings for web applications, where the database is in fact the DSL schema. These two types of syntaxes are well suited to the capabilities of LLMs.

To sum up, even if the LLM may fail to create a brand new complex DSL, there are plenty of tasks the LLM could chime in. At the very least, you could use it to criticize your work and suggest improvements.

3.5. LLMs are not perfect, humans neither

After reading the results of our experiment, an engaged reader could think that LLMs are still not up to the task (even if, as I just described, there are a variety of noncritical tasks where they can safely help).

[10] https://github.com/besser-pearl

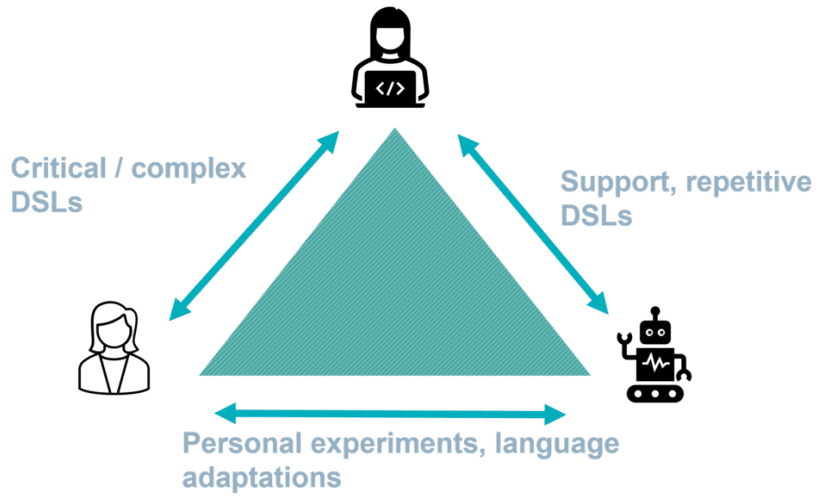

Figure 4: Collaboration paths between language engineers, domain experts and agents to build DSLs. For complex and critical domains, we may completely skip the LLM. On the contrary, for non-risky projects, the domain expert could fully create the DSL with the help of the LLM. In other scenarios, the language engineer could delegate some work to the LLM while still taking the lead on in the interaction with the user.

Maybe others will think the opposite and get the "LLM fever". As usual, the truth lies somewhere in the middle. LLMs make mistakes, but language engineers do as well. So *when* and *how much* to use the LLM will depend on the complexity of the domain, its criticality (how risky is it to make a mistake), the time we have available, the budget we have (good LLMs are not free but still cheaper than humans) or even the tool support as a good language workbench could provide some advanced features that could mitigate some of the limitations of the LLM, e.g., by guiding non-expert users in the creation of the DSL without the need for a heavy use of the LLM.

I feel the results of *vibe DSLing* are good enough in many situations. But experiment and decide for yourself what style of LLM-driven DSL generation suits you best. And repeat your experiments often. Every new major LLM release may bring new capabilities that could significantly change your perception. For instance, Claude 4 has convinced me to adopt an LLM-based approach in scenarios where when using Claude 3.7 I was still hesitant.

4. Conclusions

This paper has explored the use of LLMs for the creation of a full DSL, including both abstract and concrete syntaxes. The results show that LLMs can be indeed a powerful assistant helping DSL experts to refine their DSLs and a reasonable language engineers for non-experts that require a new DSL or an evolution of an existing one. In the latter case, the results are, of course, of limited quality, especially when it comes to the most creative aspects of the task. Note that both options are not exclusive. LLMs can also play a complementary role in the discussion and prototyping of new languages, accelerating the feedback loop between domain experts and language engineers. This would bring down the cost of creating a new DSL, improving the ROI of the language engineering process.

Moreover, given that the quality of LLMs is still improving at a rapid pace, we can foresee that their value as assistant language engineers will only improve and will become one more valuable tool in the arsenal of tools available to professional language engineers. No, they will not make us irrelevant. But they can definitely make our lives easier if you learn how to properly leverage them. Therefore, as a community, I strongly believe that we need to embrace this new tool and learn and experiment how best to use it to create better languages faster.

This reflection is based on anecdotical evidence. And as such, conclusions drawn from it have limited generalization. Even more in a context where LLMs quickly evolve. Nevertheless, I hope this paper

serves as a starting point for more experiments and discussions on this topic, e.g. properly benchmarking the quality of the LLM-generated DSL components against manually created ones.

Acknowledgements.

This project is supported by the Luxembourg National Research Fund (FNR) PEARL program, grant agreement 16544475.

I would also like to thank OOPSLE attendees and many other colleagues (especially Juha-Pekka Tolvanen and Aaron Conrardy) for their feedback on the keynote presentation at the origin of this work.

Declaration on Generative AI

The authors have not employed any Generative AI tools to create, change or rephrase the content of this document.

References

[1] J. He, C. Treude, D. Lo, Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead, ACM Transactions on Software Engineering and Methodology 34 (2025) 1–30.

[2] P. Kourouklidis, D. S. Kolovos, J. Noppen, N. Matragkas, A domain-specific language for monitoring ML model performance, in: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023 Companion, Västerås, Sweden, October 1-6, 2023, IEEE, 2023, pp. 266–275. doi:10.1109/MODELS-C59198.2023.00056.

[3] S. Morales, R. Clarisó, J. Cabot, Impromptu: a framework for model-driven prompt engineering, Software and Systems Modeling (2025) 1–19.

[4] S. Morales, R. Clarisó, J. Cabot, A DSL for testing LLMs for fairness and bias, in: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 203–213. doi:10.1145/3640310.3674093.

[5] G. d'Aloisio, C. Di Sipio, A. Di Marco, D. Di Ruscio, How fair are we? from conceptualization to automated assessment of fairness definitions, Software and Systems Modeling (2025) 1–27.

[6] R. K. Sharma, V. Gupta, D. Grossman, Spml: A DSL for defending language models against prompt attacks, arXiv preprint arXiv:2402.11755 (2024).

[7] 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, 2024, pp. 596–601.

[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, MODELS Companion '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 16–20. doi:10.1145/3652620.3687782.

[9] J. Silva, Q. Ma, J. Cabot, P. Kelsen, H. A. Proper, Application of the tree-of-thoughts framework to llm-enabled domain modeling, in: W. Maass, H. Han, H. Yasar, N. J. Multari (Eds.), Conceptual Modeling - 43rd International Conference, ER 2024, Pittsburgh, PA, USA, October 28-31, 2024, Proceedings, volume 15238 of *Lecture Notes in Computer Science*, Springer, 2024, pp. 94–111. doi:10.1007/978-3-031-75872-0_6.

[10] M. Bragilovski, A. T. van Can, F. Dalpiaz, A. Sturm, Leveraging machines to derive domain models from user stories, Requirements Engineering (2025) 1–23.

[11] M. B. Chaaben, L. Burgueño, H. Sahraoui, Towards using few-shot prompt learning for automating model completion, in: 2023 IEEE/ACM 45th international conference on software engineering: New ideas and emerging results (ICSE-NIER), IEEE, 2023, pp. 7–12.

[12] J. Cámara, J. Troya, L. Burgueño, A. Vallecillo, On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml, Software and Systems Modeling 22 (2023) 781–793.

[13] R. Saini, G. Mussbacher, J. L. C. Guo, J. Kienzle, Automated, interactive, and traceable domain modelling empowered by artificial intelligence, Softw. Syst. Model. 21 (2022) 1015–1045. doi:10.1007/S10270-021-00942-6.

[14] J. S. Cuadrado, J. L. C. Izquierdo, J. G. Molina, Comparison between internal and external dsls via rubytl and gra2mol, in: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global Scientific Publishing, 2013, pp. 109–131.

[15] D. Moody, The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering, IEEE Transactions on software engineering 35 (2009) 756–779.

[16] J. L. C. Izquierdo, J. Cabot, Collaboro: a collaborative (meta) modeling tool, PeerJ Comput. Sci. 2 (2016) e84. doi:10.7717/PEERJ-CS.84.

[17] D. Huang, J. M. Zhang, Q. Bu, X. Xie, J. Chen, H. Cui, Bias testing and mitigation in llm-based code generation, ACM Transactions on Software Engineering and Methodology (2024).

[18] R. Tairas, J. Cabot, Corpus-based analysis of domain-specific languages, Softw. Syst. Model. 14 (2015) 889–904. doi:10.1007/S10270-013-0352-6.

[19] I. Alfonso, A. D. Conrardy, A. Sulejmani, A. Nirumand, F. U. Haq, M. Gomez-Vazquez, J. Sottet, J. Cabot, Building BESSER: an open-source low-code platform, in: H. van der Aa, D. Bork, R. Schmidt, A. Sturm (Eds.), Enterprise, Business-Process and Information Systems Modeling - 25th International Conference, BPMDS 2024, and 29th International Conference, EMMSAD 2024, Limassol, Cyprus, June 3-4, 2024, Proceedings, volume 511 of *Lecture Notes in Business Information Processing*, Springer, 2024, pp. 203–212. doi:10.1007/978-3-031-61007-3_16.

[20] S. Pérez-Soler, M. González-Jiménez, E. Guerra, J. de Lara, Towards conversational syntax for domain-specific languages using chatbots., J. Object Technol. 18 (2019) 5–1.