Preliminary Study of DSL Code Generation for Robotics with LLMs

Alberto Tagliaferro(✉), Livia Lestingi, and Matteo Rossi

Politecnico di Milano, Milan, Italy
{alberto.tagliaferro,livia.lestingi,matteo.rossi}@polimi.it

Abstract. Code generation through Large Language Models (LLMs) has made significant progress in recent years. However, when the code generation involves a lesser-known Domain-Specific Language (DSL), a standard tool in software development for cyber-physical systems, LLMs' performance significantly decreases. We present an exploratory study assessing LLMs' performance in generating LIrAs (Language for Interactive Agents) code, a DSL for robotic and multi-agent tasks specification. This work is a stepping stone towards improving LLM-generated DSL code through iterative specification repair techniques driven by formal verification results.

Keywords: Domain-Specific Languages (DSLs) · DSL Code Generation · Multi-Agent Systems · Large Language Models

1 Introduction

Software-intensive systems—e.g., autonomous driving, assistive robotics, and smart manufacturing—are increasingly integrated into our lives. These systems rely on intelligent cyber agents that gather and process environmental data, and act according to their tasks and analyzed inputs. Human interaction in Multi-Agent Systems (MASs) further complicates these complex systems, requiring stakeholders to mitigate various risks, including financial and, crucially, physical harm. Consequently, specifying interactive tasks in MASs is a significant challenge for software engineers [6], demanding specialized expertise, time, and effort, and it is error-prone.

Domain-Specific Languages (DSLs) have become essential tools in robotics to specify tasks [8]. Their high-level nature makes them effective for expressing complex behaviors and configurations in a concise, human-readable form. They can also be translated into formal models, enabling formal verification of task specifications. However, users must learn both the syntax and semantics of each DSL and each application often requires a custom DSL, as shown by the large number of such languages developed for different domains [13].

The advancements in Large Language Models (LLMs) have capabilities in understanding and generating human language, as well as code in widely used

A. Cavalcanti et al. (Eds.): TAROS 2025, LNAI 16045, pp. 273–280, 2026.
https://doi.org/10.1007/978-3-032-01486-3_22

languages like Python, C, and C++ [21]. This opens up opportunities to automate software development from natural language specifications to implementation. However, when tasked with generating code in lesser-known DSLs, LLMs performance drops noticeably [3,11]. Hallucinations and syntax errors increase, and the generated DSL code often fails to capture the intended specification.

LLMs have shown strong performance in complex code generation tasks [10], including those traditionally considered challenging [7,21,22], thanks to the large amount of training data. Beyond general-purpose languages, recent research investigates their potential in generating code with DSL. For example, LLMs have been used to generate textual modeling languages like PlantUML [16] for UML diagrams [4,18]. Bassamzadeh and Methani [3] compare fine-tuning and Retrieval Augmented Generation (RAG) for DSL code generation; fine-tuning reduces hallucinations, while RAG improves syntax correctness. LLMs have also been applied in robotics, particularly for task planning and execution. Singh et al. [17] propose ProgPrompt, which uses LLMs to generate executable task plans from program-like specifications, showcasing the potential to bridge human-readable task descriptions and robot-executable plans.

This work investigates the use of LLMs for DSL code generation in robotics, starting from real-world scenario descriptions. As a testbed, we use the LIrAs DSL, a language designed for specifying MAS tasks via agents' atomic abilities [19]. Listing 1.1 [20] illustrates a basic example of LIrAs, introducing some of the language's key elements to aid in understanding the experimental campaign described in Sect. 2. In this example, a `Human` and a `Robot` must reach the first Point of Interest (PoI), namely `poi1`, but the `Robot` begins its movement only once the `Human` has reached the destination. After both agents arrive at the destination, the `Human` proceeds towards `poi2`, followed by the `Robot`.

In the first line of Listing 1.1 (as in any LIrAs specification), we define the name of the pattern, followed by the required input parameters—agents, locations, and any necessary resources (Lines 2, 3, 4, 5). Next, separate sequences are defined (Line 6 and Line 9), one for each agent involved in the pattern. These sequences specify the actions that the corresponding agent must perform, e.g., the actions listed on Line 7 and Line 8 for the `Human` agent. The concurrent execution of these sequences, synchronizes according to specific rules, the most important of which are described here.

Each sequence may contain an arbitrary number of sub-sequences, labeled using the notation `i:`, where `i` is a natural number. Within a sequence, sub-sequences must have unique numbers and must be in ascending order. When the pattern starts, all sub-sequences labeled `1:` begin simultaneously across all agents. Execution of sub-sequences labeled `2:` starts only after all agents have completed their first sub-sequence. Notably, a sub-sequence may contain multiple actions, and its duration is not predetermined.

In LIrAs, it is also possible to handle empty or missing sub-sequences, loops, and a property known as continuation, which allows an agent to skip certain synchronizations with others. Finally, as shown in Line 10, conditional statements can be used to control actions. The conditional constructs available in LIrAs

Listing 1.1. LIrAs specification of the motivating example.

```
RobotFollower
agents:
    humans: Human
    robots: Robot
locations: poi1, poi2
Human:
  1: moveTo poi1
  2: moveTo poi2
Robot:
  1: moveTo poi1 if position(Human,poi1) else stop
  2: follow Human (poi2)
```

include `if/else`, `until`, and their combinations. Each conditional statement depends on one or more atomic predicates, combined using unary or Boolean operators (e.g., `and`, `or`, `not`).

In this work, we explore various system prompt configurations—including the DSL grammar, textual descriptions, and in-context learning. This last technique means that we provide one or more user-assistant examples, more specifically we provide a specification in natural language and the expected associated LIrAs code that the LLM should generate. Experiments were conducted using a locally deployed LLM, and the generated LIrAs code was manually evaluated. Our goal is to leverage LLMs to lower the cost and complexity of writing DSL specifications, making them more accessible to the robotics community.

The experimental setup and the results are presented in Sect. 2. While Sect. 3 concludes and outlines the envisioned future developments.

2 Probing LLMs for LIrAs Code Generation

This section presents the experimental campaign supporting our exploratory study, which addresses the following research questions:

RQ1. How accurate is the selected LLM in generating LIrAs specifications?
RQ2. What is the time overhead of using the LLM for DSL code generation?

To address these questions, we developed the pipeline shown in Fig. 1[1]. We tested the generation of LIrAs code for four scenarios (SC_k where $k \in \{1, 2, 3, 4\}$ in Fig. 1) drawn from the literature on real-world robotic applications [1,2,15]. Each natural language specification was pre-processed to remove superfluous

[1] Experimental details, results, and evaluations are available at https://github.com/albiferro99/TAROS25-Paper74. All experiments were performed on a machine with an Apple M3 chip and 16 GB RAM, using Meta-Llama-2-7B with Q4 quantization [14].

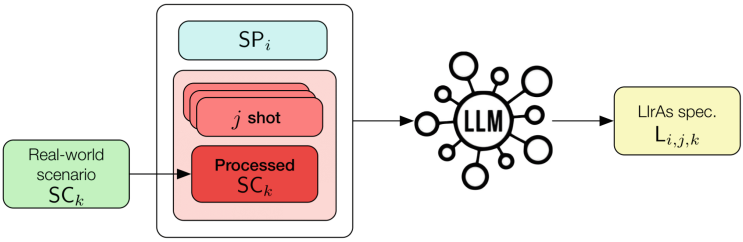

Fig. 1. Pipeline overview, from a real-world scenario up to its LIrAs specification.

Table 1. Questions used to evaluate LLM-generated specifications for **RQ1**.

Q1	Is the name of the Scenario consistent with the specification?
Q2	Are the inputs syntactically well structured?
Q3	Are the input agents consistent with the specification?
Q4	Are the input locations consistent with the specification?
Q5	Are the input resources consistent with the specification?
Q6	Is there a sequence for each Agent?
Q7	Is there any additional Sequence?
Q8	Are the used actions coherent with the specification?
Q9	Are the used atomic predicates coherent with the specification?
Q10	Are sub-sequences numbered increasingly?
Q11	Do sub-sequences contain continuations?
Q12	Are actions that must be executed simultaneously in the same sub-sequence?
Q13	Is the specification syntactically correct?
Q14	Are there no hallucinations? (Extra information, sub-sequence for locations...)

context and ambiguous details, ensuring that inputs matched what a user might realistically provide. Statements like "Infections caught during hospital stays account for around 37,000 deaths per year in Europe and almost 100,000 in the U.S." were removed, as they do not contribute to robotic specifications.

To define the LLMs' behavior across all interactions [9] we initialized it with five different system prompts (SP_i, $i \in \{1,2,3,4,5\}$). SP_1 and SP_2 have textual descriptions of LIrAs's features, e.g., agent synchronization and conditional statements. SP_3 provides the grammar of LIrAs, while SP_4 includes the grammar and explanatory comments. Lastly, SP_5 is a pseudo-code version of LIrAs. We also adopted in-context learning [5] with j-shot configurations ($j \in 0,1,2$). Each shot included (i) a user prompt describing a scenario from real-world robotics applications [12] and (ii) the corresponding expected DSL code. This approach is especially useful when the model is not fine-tuned for the task, as it helps reduce hallucinations, syntax errors, and unnecessary textual elaboration.

In summary, we combined 5 system prompts, 4 scenarios, and 3 shots configurations. This resulted in 60 generated LIrAs instances, denoted as $\mathsf{L}_{i,j,k}$.

To address **RQ1**, we evaluated the accuracy of each generated LIrAs code against the original scenario description using 14 quantitative metrics (Table 1) tailored to LIrAs. Each metric was scored by a human expert using a simplified 0–2 Likert scale: 0 (missing or incorrect), 1 (partially correct), and 2 (fully correct and aligned with the expectations). These metrics provide a general overview of

Table 2. RQ1 average results among the four different tested scenarios.

	SP1			SP2			SP3			SP4			SP5		
	0-shot	1-shot	2-shot	0-shot	1-shot	2-shot	0-shot	1-shot	2-shot	0-shot	1-shot	2-shot	0-shot	1-shot	2-shot
Q1	1.5	1	2	0.5	0.5	2	0	0.5	2	0	0.5	1	1.75	0.75	2
Q2	0	2	2	0	1.5	2	0	1.75	2	0	2	2	0	1.75	2
Q3	0.5	1	1.5	0	1.25	1.25	0	0.75	1.25	0	0.75	1.25	0	1	1.5
Q4	0.5	0.75	1.75	0	0.75	1.5	0	0.5	1.5	0	0.5	1.75	0	0.75	1.5
Q5	0.25	0.25	1	0	0.25	1.25	0	0.75	1	0	0.25	1	0	0.75	1.5
Q6	0	2	2	0.75	1.5	2	0	1.75	2	0	1.5	1.5	0.5	2	2
Q7	0.5	0.5	0.5	0.5	1.5	0.5	0.5	0.5	0	1	0	0.5	0.5	0.5	1
Q8	1.25	1	1.5	1.25	0.75	1.75	0	0.75	2	0.25	0.75	0.75	0.75	1.5	1
Q9	0.25	0	0.75	0	0	1	0	0	0	0	0	1	0	0	0.25
Q10	0.75	2	1.75	1.25	2	2	0	2	2	0	1.75	2	0	2	2
Q11	0	2	1.75	0.25	1.5	2	0	1.5	1.25	0	1	2	0	1.75	2
Q12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Q13	0	0.5	1.5	0	1	1	0	0.75	1.25	0	0	0.75	0	0.25	1.75
Q14	0	0	0.5	0.5	0.5	0.75	0.25	0	0.5	0	0	0.5	0	0.5	1.5
Average	0.393	0.929	1.321	0.357	0.929	1.357	0.054	0.821	1.196	0.089	0.643	1.143	0.25	0.964	1.429
Overall	0.25	0.5	1.25	0	0.25	1	0	0	1.75	0	0.25	0.75	0	0.5	1.75

the quality of the generated LIrAs code, focussing on syntax correctness and alignment with the specification. An exception is the *Overall evaluation*, a more subjective metric, which uses a 0–5 scale to capture broader qualitative judgments. For **RQ2**, we track the time required to generate each $\mathsf{L}_{i,j,k}$.

Table 2 reports the results for **RQ1**, specifically the average score across the four input scenarios and all the possible configurations previously described.

The most important trend observed in the figures reported in Table 2 is the always growing score both on the *Avarege value* and the *Overall evaluation* for each *SP* from 0-shot up to 2-shots. This shows the importance of in-context learning when dealing with applications the LLMs are not trained for.

Several trends emerge from the results. On the positive side, pattern names are usually consistent, and input sections are generally captured well, both syntactically and semantically. Sub-sequences are almost always numbered in ascending order, aligning with LIrAs conventions. However, a key feature—agent synchronization through sub-sequence labels—was never accurately captured, as shown by the zeros in Q12 of Table 2. Similarly, the property called "continuation" was frequently used in a syntactically correct manner but never meaningfully deployed, as it appears almost randomly rather than being driven by the scenario's requirements. Actions were mostly well-aligned and ordered correctly within each agent's sequence. Conditional statements and atomic predicates were less frequent than expected and often failed to match scenario details. Hallucinations (Q14 in Table 2) also surface in the generated code. However, these are mitigated in two-shot learning, where the model shows fewer extraneous additions. Table 2 shows that in-context learning (1- or 2-shot) consistently improved syntax, scenario alignment, and reduced hallucinations. Finally, in 0-shot cases with grammar-based prompts (SP_3 and SP_4), the model often misunderstood the task, sometimes outputting only the grammar or irrelevant natural language sentences.

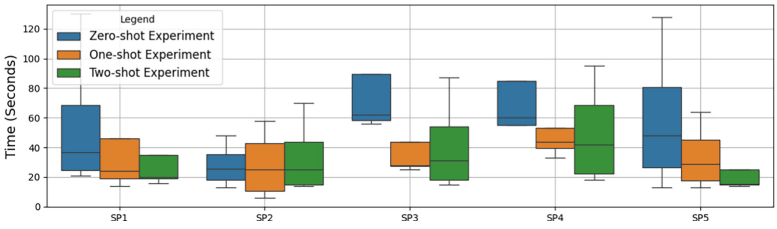

Fig. 2. RQ2 results: time [s] to generate the DSL code, grouped by SP and number of shots.

Figure 2 shows a grouped box plot of generation times. On average, generating a result takes about 45 seconds, with times ranging from 6 to 164 seconds. Notably, 0-shot examples took longer and were less accurate, indicating this approach is less effective than 1- or 2-shot learning.

Because this is a preliminary study, it is not exhaustive. Rather, it lays the groundwork for future research and more extensive empirical evaluations. For instance, future work could explore different models (including those not deployed locally) with more parameters to enhance LLMs performance, consider a wider array of scenarios and more-shot examples, employ metrics to capture semantic correctness more precisely, and automate the evaluation process to increase objectivity and reduce time overhead.

3 Future Research Outlook

This preliminary work is a baseline for future research. The envisioned framework begins with a user providing a natural language prompt to the Generative AI, describing the agents' goals and the operational environment. The Generative AI, potentially using the foundational DSL syntax and semantics (in this case, LIrAs), generates a textual DSL specification for the mission. Next, the AI-generated DSL specification is translated into a formal model (or programming language) and undergoes formal verification (or program verification). The resulting specification is then subjected to a data analysis step, which generates a repair action specification to be sent to a *Prompt Refiner*. The Prompt Refiner, starting with the initial natural language prompt, implements the repair action and updates the prompt for the Generative AI (LLM) to produce a revised specification. This iterative process can be repeated multiple times to ensure the final specification aligns with the user's requirements.

Summarizing, this framework allows users to generate DSL specifications for robotic missions via LLM, laying the groundwork for future research in automated mission specification through an iterative refinement process.

References

1. Askarpour, M., et al.: Robomax: robotic mission adaptation exemplars. In: 2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 245–251. IEEE (2021)
2. Baraka, K., Veloso, M.M.: Mobile service robot state revealing through expressive lights: formalism, design, and evaluation. Int. J. Soc. Robot. **10**, 65–92 (2018)
3. Bassamzadeh, N., Methani, C.: A comparative study of dsl code generation: fine-tuning vs. optimized retrieval augmentation. arXiv preprint arXiv:2407.02742 (2024)
4. De Bari, D., Garaccione, G., Coppola, R., Torchiano, M., Ardito, L.: Evaluating large language models in exercises of UML class diagram modeling. In: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '24, pp. 393–399. Association for Computing Machinery (2024)
5. Dong, Q., et al.: A survey on in-context learning (2024). arXiv:2301.00234 [cs]
6. Dragule, S., Gonzalo, S.G., Berger, T., Pelliccione, P.: Languages for specifying missions of robotic applications. In: Software Engineering for Robotics, pp. 377–411 (2021)
7. Feng, Z., et al.: Codebert: a pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
8. Fowler, M.: Domain-Specific Languages. Pearson Education, Boston (2010)
9. Giray, L.: Prompt engineering with ChatGPT: a guide for academic writers. Ann. Biomed. Eng. **51**(12), 2629–2633 (2023)
10. Jana, S., Biswas, R., Pal, K., Biswas, S., Roy, K.: The evolution and impact of large language model systems: a comprehensive analysis. Alochana J. **13** (2024)
11. Joel, S., Wu, J.J., Fard, F.H.: A survey on llm-based code generation for low-resource and domain-specific programming languages. arXiv preprint arXiv:2410.03981 (2024)
12. Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T.: Specification patterns for robotic missions. IEEE Trans. Softw. Eng. (2019)
13. Nordmann, A., Hochgeschwender, N., Wrede, S.: A survey on domain-specific languages in robotics. In: Brugali, D., Broenink, J.F., Kroeger, T., MacDonald, B.A. (eds.) SIMPAR 2014. LNCS (LNAI), vol. 8810, pp. 195–206. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11900-7_17
14. Ollama: Llama 2 (2025). https://ollama.com/library/llama2
15. of Robotics IFR, I.F.: Case studies - service robots (2025). https://ifr.org/case-studies/service-robots-case-studies
16. Roques, A., Contributors, P.: PlantUML Software (2009). https://github.com/plantuml/plantuml
17. Singh, I., et al.: Progprompt: generating situated robot task plans using large language models. In: 2023 IEEE International Conference on Robotics and Automation (ICRA), pp. 11523–11530 (2023)
18. Tagliaferro, A., Corbo, S., Guindani, B.: Leveraging llms to automate software architecture design from informal specifications. In: 9th International Workshop on Formal Approaches for Advanced Computing Systems (FAACS), pp. 291–299. IEEE (2025)
19. Tagliaferro, A., Lestingi, L., Rossi, M.: Towards verifiable multi-agent interaction pattern specification. In: Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE), pp. 122–126 (2024)

20. Tagliaferro, A., Lestingi, L., Rossi, M.: Verification-oriented specification of multi-agent interaction patterns. In: Workshop on Agents and Robots for reliable Engineered Autonomy, pp. 38–53. Springer, Heidelberg (2024). https://doi.org/10.1007/978-3-031-73180-8_3
21. Wang, J., Chen, Y.: A review on code generation with llms: application and evaluation. In: IEEE International Conference on Medical Artificial Intelligence (MedAI), pp. 284–289. IEEE (2023)
22. Xu, F.F., Vasilescu, B., Neubig, G.: In-ide code generation from natural language: promise and challenges. ACM Trans. Softw. Eng. Methodol. (TOSEM) **31**(2), 1–47 (2022)