SysTemp: A Multi-Agent System for Template-Based Generation of SysML v2

Yasmine Bouamra[1,2], Bruno Yun[2], Alexandre Poisson[1], and Frédéric Armetta[2]

[1] Siemens Digital Industries Software
[2] University Claude Bernard Lyon 1,
Ecole Centrale de Lyon, INSA Lyon,
Université Lumière Lyon 2, LIRIS, UMR5205,
69622 Villeurbanne, France

Abstract. The automatic generation of SysML v2 models represents a major challenge in the engineering of complex systems, particularly due to the scarcity of learning corpora and complex syntax. We present SysTemp, a system aimed at facilitating and improving the creation of SysML v2 models from natural language specifications. It is based on a multi-agent system, including a template generator that structures the generation process. We discuss the advantages and challenges of this system through an evaluation, highlighting its potential to improve the quality of the generations in SysML v2 modeling.

Keywords: SysML v2 · Large Language Models · Model-Based Systems Engenieering · Multi-Agents.

1 Introduction

Model-Based Systems Engenieering (MBSE) [12] is an approach that relies on the use of system models (representations of a system's components, behaviors, and interactions) for analysis, design, and validation in engineering and systems development. MBSE is increasingly adopted as it provides a more rigorous, comprehensive, and collaborative approach to system development compared to the traditional document-based methods. A key driver for MBSE is the fact that 70% of a product's lifecycle cost is defined during the conceptual phase. MBSE enables systems engineers to capture system requirements, design, analysis, and verification in an integrated model, rather than relying on disconnected files and documents. This model-centric approach helps identify issues earlier, improve design quality, and streamline the development process - overcoming the limitations of the traditional file-based approach that was leading to a dead end.

SysML v1[3] is the most widely adopted modeling language within MBSE, allowing for the encoding of a wide variety of diagrams. It provides a rigorous framework for the detailed description of a system model. However, SysML v1 has limitations, including insufficient expressiveness, increasing complexity in

[3] See the documentation at `https://www.omg.org/spec/SysML/1.0/PDF`.

managing large-scale models, and the absence of a standardized textual representation. To address these challenges, the *Object Management Group (OMG)* has undertaken the development of SysML v2[4]. This second version allows for the translation of specifications expressed in natural language into a formal model, described in a textual notation (see Fig. 1 for an illustrative example).

```
part vehicle{
    attribute mass = engine.mass+transmission.mass;
    perform providePower;
    part engine{
        attribute mass;
        port torqueOutPort;
        perform providePower.generateTorque;
    }
    part transmission{
        attribute mass;
        port torqueInPort;
        perform providePower.amplifyTorque;
    }
    connect engine.torqueOutPort to transmission.torqueInPort;
}
action providePower{
    action generateTorque;
    action amplifyTorque;
}
```

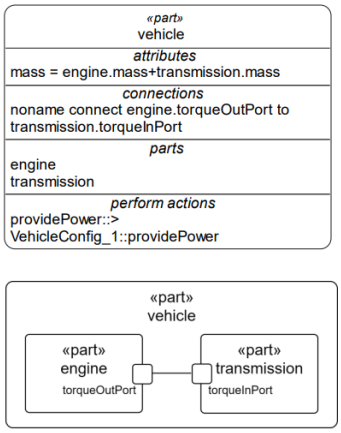

Fig. 1. Example of vehicle modeling in SysML v2: The textual notation (left) and the corresponding graphical notation (right) - Source: `https://tinyurl.com/ycjbexme` - OMG Slide 18.

However, assisting system engineers to write SysML v2 models is complex as (1) the final specification of the language has not been released as its development is still in progress, (2) there are not many available examples (less than 150), and (3) while there is an existing parser[5], it does not describe the errors encountered in the model.

Thus, a critical question emerges: *how can we assist system engineers in writing SysML v2 models*?

Recently, several assistant, based on generative artificial intelligence (GenAI), have been integrated into various software environments (such as Visual Studio Code [13] and Photoshop [1]) with the objective of automating repetitive tasks, providing recommendations, and assisting users to reduce their workload. Following this trend, we plan to assist users by exploring the challenges associated with translating specifications and instructions formulated in natural language into correct SysML v2 models, leveraging the capabilities of large language models (LLMs). This preliminary work shows promises for future integration into a

[4] The improvements of SysML v2 are presented by Sanford Friedenthal: `https://www.omg.org/pdf/SysML-v2-Overview.pdf`.

[5] See `https://tinyurl.com/4wauck7j`.

software assistant dedicated to SysML v2 modeling within *Siemens Digital Industries Software*.

While LLMs have shown excellent performance in user assistance tools (and other downstream tasks like text generation [15], translation [24] and code synthesis [17], due to their ability to capture complex relationships within text (and code), challenges persist in their adaptation to specialized domains with limited or no available data [16]. Techniques such as few-shot learning [8] and chain-of-thought [9] were developped to tackle these challenges by leveraging LLMs' remarkable generalization capabilities [6]. However, they do not entirely resolve the issue [18], highlighting the need for further research into improving LLM adaptability in low-data scenarios. In the case of SysML v2, initial tests[6] have reported poor performance of state-of-the-art closed-source (GPT-4 [19]) and open-source (Llama 3.1-70B [2], Mistral 7B [3], Gemma[21]) LLM models for related tasks.

To overcome this generalisation problem, we have used a multi-agent approach which has been proven to be particularly suitable for complex tasks [14], as it facilitates the decomposition of the task into subtasks and enable successive iterations incorporating feedback and compilation signals. Our contributions are two-folds: (1) We propose a multi-agent system called `SysTemp`, dedicated to SysML v2 code generation, and (2) analyze its effectiveness on several scenarios.

The structure of this paper is as follows. In Section 2, we present the background and related work on SysML v2 generation. Section 3 then provides a detailed description of our model and its architecture. The experiments and obtained results are discussed in Sections 4 and 5, where we evaluate our approach's performance on a set of representative case studies. Finally, we conclude by summarizing the key contributions of this research and identifying potential improvements and future applications.

2 Background and Related Work

The automatic generation of SysML v2 models from natural language specifications represents a major challenge in integrating assistance tools for systems engineering. This problem is similar to code generation in that it involves converting a textual description into a formal representation usable by modeling tools. SysML v2 relies on a dual syntax, as shown in Fig. 2: an abstract (and machine-oriented) syntax, expressed in JSON, and a concrete syntax, in code form, which is closer to human language and more readable. It is possible to translate from the abstract to the concrete syntaxe with an existing function. The abstract syntax is structured as dictionaries of values and is not suited for manual manipulation. Automatic handling of SysML v2 is possible through an API[7] specified by the OMG. It is possible, through API calls, to modify an

[6] See benchmark at `https://github.com/yasminebouamra/SysMLv2-Benchmark`.
[7] `https://github.com/Systems-Modeling/SysML-v2-API-Services`.

existing model by adding, and/or removing elements. The concrete syntax[8], resembles traditional programming languages, it is more concise, and it is more understandable by humans. Consequently, we decided to use LLMs to generate the concrete syntax of a model as it is shorter, thus requiring less resources to generate and avoiding problems with large contexts [4].

Fig. 2. Example of abstract syntax (left) and concrete syntax (right) in SysML v2 – The abstract syntax is the translation of the concrete syntax but was cropped for readability purposes.

A major obstacle to applying LLMs for SysML v2 generation lies in the lack of suitable training data. Unlike traditional programming languages, which have extensive annotated corpora and open-source codebases, SysML v2 resources are limited and heterogeneous[9]. From our initial test, even the latest LLMs struggle to generalize effectively for specific modeling tasks in systems engineering, making generation less reliable and more prone to syntactic and semantic errors.

To date, few studies have combined artificial intelligence with SysML modeling, whether in the initial version or in SysML v2. Among the existing contributions, John K. Dehart's work [10] proposes a methodology for the automated generation of SysML v2 API calls to add, remove, or modify model elements. This approach relies on targeted API calls to manipulate specific local properties of the generated model. However, their approach requires the user to have an initial SysML v2 model as input. Thus, a promising research avenue would require the exploration of alternative approaches, such as integrating multi-agent systems [22] capable of combining model generation with iterative verification and correction mechanisms.

The adoption of multi-agent systems for code generation via LLMs has grown significantly in recent years, leading to the development of specialized frame-

[8] The terms *concrete syntax* and *textual notation* refer to the same thing and are used interchangeably in the article.

[9] There are only two available sources: `https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation/tree/master/sysml/src` and `https://github.com/GfSE/SysML-v2-Models`, totalling less than 150 SysML v2 scenarios.

works such as AutoGen [23]. The AutoGen framework allows to orchestrate multiple LLM agents specialized in distinct tasks, such as code generation, syntax validation, and optimization. This methodology is particularly relevant for assisted code generation, where models like PPoCoder [20] or StepCoder [11] adopt an incremental approach. Instead of producing a program in a single phase, these models decompose the task into sub-problems and iteratively refine the proposed solutions based on interaction feedback. This iterative structuring not only enhances the quality of the generated code but also improves alignment with the initial specifications.

This growing interest in multi-agent orchestration for structured generation tasks extends beyond code to include system modeling. Maria Stella de Biase [7], for example, explores state machine generation through a refinement process that allows users to iteratively adjust solutions to their specifications.

In this vein, our contribution introduces a multi-agent system augmented by a SysML v2-specific parser and a template-based generator. By leveraging agent collaboration, conversational programming, and user interaction, we aim to enhance automated SysML v2 model generation, producing more accurate system models.

3 Generative Pipeline Based on Specialist Agents

In this section, we present `SysTemp`, a framework designed to guide an LLM in generating SysML v2 models from natural language descriptions.

`SysTemp` is a multi-agent framework, illustrated in Fig. 3, that relies on multiple specialist agents interacting with each other.

We introduce two key specialist agents that characterize our approach: first, the `TemplateGeneratorAgent`(TG), which structures the model by generating a syntactically correct SysML v2 skeleton, and second, the `ParserAgent`, which ensures compliance with specifications by detecting potential syntactic errors (e.g. missing parenthesis, missing semicolumn...)

The `ParserAgent` and `WriterAgent` work together in an iterative loop, where the generated model is progressively refined and adjusted, ensuring conformity to user requirements while minimizing syntactic errors.

Roughly speaking, the generation process unfolds as follows:

1. First, the user provides a natural language description of the desired model, detailing system requirements and characteristics.
2. This input is processed by the `SpecificationGeneratorAgent`, which extracts key information and reformats it into a structured Python dictionary.
3. The structured Pytthon dictionary is passed to the `TemplateGeneratorAgent`, which generates a syntactically correct SysML v2 skeleton.
4. The `WriterAgent` uses the skeleton as base and completes it to specify various values (attributes, etc.) consistent with the user's needs, producing a completed SysML v2 model.

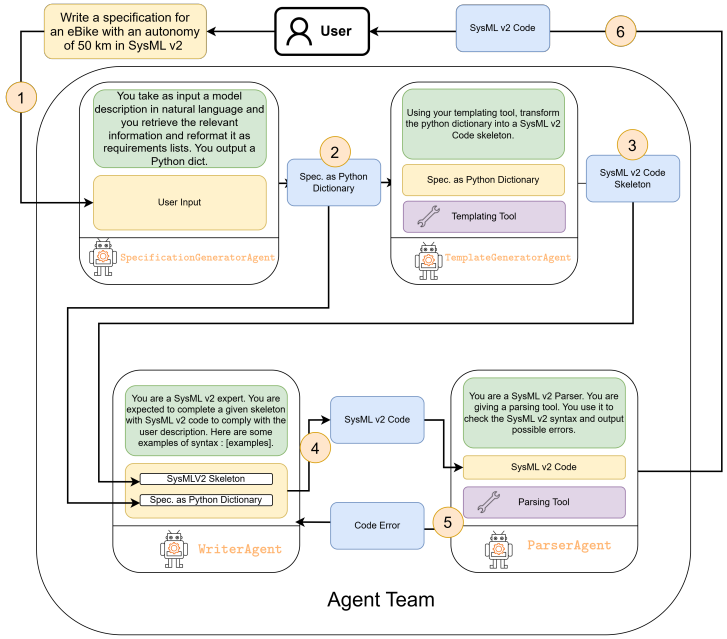

Fig. 3. `SysTemp` pipeline: SysML v2 generation via a multi-agent system.

5. The completed model is then analyzed by the `ParserAgent`, which detects potential syntactic errors or structural inconsistencies and forwards them to the `WriterAgent`.
6. Based on this feedback, the `WriterAgent` iteratively adjusts the SysML v2 model until it meets the initial requirements and is free of errors.

In the next section, we formally define the different agents that constitute this pipeline.

3.1 Pipeline's Agents

The pipeline consists of four agents, described in the following subsections. We consider a single LLM as a non-deterministic function f which outputs a string given a string input. This LLM is re-used for all our agents. In our agents, the LLM f may be used to either (1) generate the agent's output directly or (2) provide the input to a tool $t \in \{\mathcal{P}, \mathcal{T}\}$ that generates the agent's output (described below).

SpecificationGeneratorAgent The `SpecificationGeneratorAgent` automatically extracts requirements from a natural language specification. It analyzes the user-provided description to identify key system attributes, structuring them into

a Python dictionary for downstream agents. The SpecificationGeneratorAgent employs few-shot prompting. Namely, For a description or query in natural language for the desired model D^q_{NL}, the procedure SpecificationGeneratorAgent outputs a Python dictionary $Dict^q$ (as a string). An illustrative example of the output is displayed in Fig. 4. Formally, we have:

- **Input:** A natural language specification D^q_{NL},
- **Output:** $Dict^q = f(\texttt{concat}(P_{RGA}, Ex, D^q_{NL}))$,

where P_{RGA} is the agent's system prompt[10], q is the given scenario, k is the number of examples, $Ex = \texttt{concat}(\{(D^{(i)}_{NL}, Dict^{(i)}) \mid i \in \{1, \ldots, k\})$ are few-shot examples, $D^{(i)}_{NL}$ is the i-th natural language specification, and $Dict^{(i)}$ is the i-th corresponding structured Python dictionary.

```
{
    "Package": "BikeFork",
    "attributes": {
        "power": "Rated power in watts (W) or
            horsepower (HP)",
        ...
    },
    "constraints": {
        ...
    },
    "requirements": {
        "Material": "The bike fork should be made of
            aluminum.",
        ...
    }
}
```

Fig. 4. Specification as Python dictionary obtained from SpecificationGeneratorAgent.

TemplateGeneratorAgent The TemplateGeneratorAgent is a core component of our approach, responsible for constructing a syntactically correct SysML v2 skeleton (see Fig. 5) from the structured dictionary generated by the SpecificationGeneratorAgent. This model serves as a foundation for integrating detailed specifications, yielding a format-compliant representation of system requirements. Formally, we have:

- **Input:** $Dict^q$ obtained from SpecificationGeneratorAgent (see previous sub-section),

[10] See Appendix A.1.

```
package BikeFork {

    requirement Material
    {
        doc /* The bike fork should be
        made of aluminum. */
    }

    requirement PivotType
    {
        doc /* The bike fork should have
        a 1" 1/8 Aheadset pivot. */
    }
}
```

Fig. 5. Skeleton (in textual notation) for a BikeFork SysML v2 Specification.

– **Output:** $M_{SysML}^{skeleton} = \mathcal{T}(f(\texttt{concat}(P_{TGA}, Dict^q)))$. Here the LLM is used to provide the input to the tool.

In the above, P_{TGA} is the agent's system prompt[11], $\mathcal{T}$ is a tool that applies a sets of rules[12] to $Dict^q$ by substituting placeholders in a template with corresponding values from $Dict^q$. The rules specify the document's fundamental structure and syntax. The template tool $\mathcal{T}$ follows an expert system approach leveraging the Jinja2 library[13]. In this agent, the LLM's role is mainly to adapt the input to the tool's signature.

WriterAgent The `WriterAgent` plays a crucial role in the co-construction process by completing the skeleton generated by the `TemplateGeneratorAgent` with information from the user's specifications to obtain a fully specified and SysML v2-compliant model. By combining the skeleton with user-provided specific data, this agent ensures model adaptation while maintaining compliance with SysML v2 syntax and principles. Formally, we have:

– **Input:** $M_{SysML}^{skeleton}$, $Dict^q$ (see previous sub-sections), and a possible reply r from the `ParserAgent` (see next sub-section).
– **Output:** $M_{SysML}^{completed} = f(\texttt{concat}(P_{WA}, Dict^q, M_{SysML}^{skeleton}, r))$,

where P_{WA} is the agent's system prompt[14] and r is the empty string initially. Note that P_{WA} includes examples illustrating the structure of specific elements such as parts, requirements, and other relevant components. The information

[11] See Appendix A.2.
[12] See appendix B for full description.
[13] See `https://jinja.palletsprojects.com/en/stable/templates/`.
[14] See Appendix, Table 3.

provided by the ParserAgent is then used to iteratively correct syntax until it converges to a specification-compliant version.

ParserAgent The ParserAgent is responsible for the syntactic and structural validation of the SysML v2 model. Its primary function is to analyze the model generated by the WriterAgent, identify potential syntax or structural errors, and return a detailed report E specifying the errors and their corresponding locations within the model. Formally, we have:

- **Input:** A SysML v2 model $M_{SysML}^{completed}$ generated by the WriterAgent,
- **Ouput:** $E = \mathcal{P}(f(\texttt{concat}(P_{PA}, M_{SysML}^{completed})))$. Here the LLM is used to provide the input to the tool.

In the above, $P_{PA}$is the agent's system prompt[15], $\mathcal{P}$ is a validation tool that checks whether $M_{SysML}^{completed}$ conforms to the formal grammar defining the SysML v2 syntax, as specified by the OMG SysML v2 standard. The parsing mechanism is implemented in Java. The identified errors E are returned to the WriterAgent enabling iterative correction until the model fully complies with the SysML v2 specification. In this agent, the LLM's role is mainly to adapt the input to the tool's signature.

4 Evaluation Protocol for the proposed pipeline

4.1 Evaluating the Structural Role of the Proposed Model

The proposed system is evaluated through an ablation study. We aim to measure the effectiveness of the process with and without the TemplateGeneratorAgent. In the ablation setup, the system processes a natural language description without relying on a pre-constructed skeleton.

Formally, without TemplateGeneratorAgent, the WriterAgent is modified as follows:

- **Input:** $Dict^q$ (see previous sub-sections), and a possible reply r from the ParserAgent (see next sub-section).
- **Output:** $M_{SysML}^{completed} = f(\texttt{concat}(P'_{WA}, Dict, r))$,

where P'_{WA} is the agent's system prompt[16].

The errors are quantified using a counting function that records the number of syntactic errors detected by the ParserAgent.

[15] See Appendix, Table 5.
[16] See Appendix, Table 4.

4.2 Data & Models

The study presented focused on five use cases[17] described in Appendix. Our examples cover different types of bicycles (mountain bikes, electric bikes, tires, forks, etc.). For each scenario, an initial request is written in natural language, and the `TemplateGeneratorAgent` generates the corresponding skeleton. The scenarios primarily differ in terms of component types.

For the ablation study, we selected two state-of-the-art LLMs, GPT-4 Turbo and Claude 3.5 Sonnet, due to their strong performance across various downstream tasks. Additionally, preliminary experiments indicated that open-source LLMs significantly underperform in generating SysML v2 models, making their inclusion less relevant for this study. Moreover, we used $k = 3$ examples for `WriterAgent`. Literature showed that a good number of examples is needed to infer in few-shot prompting.

5 Results

The results in Fig. 6 show the evolution of syntax errors generated by the LLM during the `WriterAgent` - `ParserAgent` loop. A significant difference is observed between the approaches with and without the `TemplateGeneratorAgent`. With the `TemplateGeneratorAgent`, near-systematic convergence is achieved, with syntax being correct and validated by the `ParserAgent` in 80% of the scenarios (4 out of 5). In contrast, without a model-skeleton, convergence (or error-free results at step 5) occurs in only 1 out of 5 scenarios, emphasizing the importance of the initial structuring provided by the skeleton.

Both models (Claude Sonnet 3.5 [5] and GPT-4 [19]) yield similar results. A slight reduction in the number of errors is observed with GPT-4. On average, three fewer errors but this difference is not substantial enough to warrant a significant distinction.

6 Conclusion

In this paper, we explore the application of LLMs to the automatic generation of SysML v2 models, a modeling language used in systems engineering. Our approach, based on a structured and iterative multi-agent system, investigates the feasibility of using GenAI to address the lack of suitable training data.

We propose `SysTemp`, a multi-agent system where we introduce two key agents `TemplateGeneratorAgent` and `ParserAgent`, which respectively create a SysML v2 skeleton and provide feedback on possible syntactical errors in the generated SysML v2 code.

Our approach primarily focuses on correcting syntax errors in the model, establishing a foundation upon which further refinements can be built. For instance, future research could explore the semantic quality of generated values.

[17] See appendix C for full description.

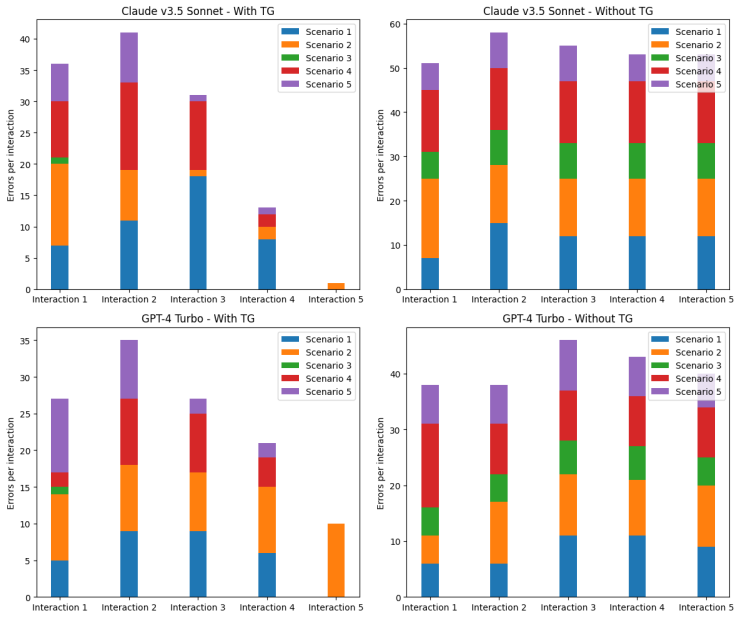

Fig. 6. Evolution of the number of errors with and without the `TemplateGeneratorAgent` for five scenarios.

This could be achieved through an iterative generative process where the user specifies the desired characteristics of their design.

However, several challenges remain. On the one hand, the semantics of the generated models require further investigation to ensure the correct interpretation of concepts and optimal alignment with user needs. On the other hand, assessing the quality of generated SysML v2 models remains constrained by the absence of standard benchmarks, highlighting the need to develop specific evaluation metrics tailored to this task. We propose investigating the integration of an LLM-based jury, where multiple instances of the same LLMs are prompted and the most reccurent output is chosen, to remedy these problems. Additionally, incorporating expert knowledge in the form of ontologies or knowledge graphs could enhance the coherence and relevance of the generated models. Our pipeline could also be used for synthetic data generation. Finally, extending this approach to other under-documented modeling languages could further broaden the applicability of LLMs in systems engineering.

Thus, our work lays the foundation for a new methodology for assisted SysML v2 generation and paves the way for future developments aimed at strengthening collaboration between artificial intelligence and systems engineers in advanced modeling contexts.

References

1. Adobe: Adobe photoshop with ai (2025), https://www.adobe.com/products/photoshop/ai.html
2. AI, M.: Introducing llama 2: The next generation of our open source large language model (2023), https://ai.facebook.com/blog/llama-2/
3. AI, M.: Announcing mistral 7b (2023), https://mistral.ai/blog/mistral-7b
4. An, C., Zhang, J., Zhong, M., Li, L., Gong, S., Luo, Y., Xu, J., Kong, L.: Why does the effective context length of llms fall short? (2024), https://arxiv.org/abs/2410.18745
5. Anthropic: Introducing claude 2.1 (2023), https://www.anthropic.com/claude
6. Bender, E.M., et al., G.: On the dangers of stochastic parrots: Can language models be too big? In: Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency. pp. 610–623 (2021)
7. de Biase, M.S., Marrone, S., Palladino, A.: Towards automatic model completion: from requirements to sysml state machines (2022), https://arxiv.org/abs/2210.03388
8. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., et al., J.K.: Language models are few-shot learners (2020), https://arxiv.org/abs/2005.14165
9. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto et al., H.P.: Evaluating large language models trained on code (2021), https://arxiv.org/abs/2107.03374
10. Dehart, J.K.: Leveraging large language models for direct interaction with sysml v2. ResearchGate (2023), https://www.researchgate.net/publication/383858648_Leveraging_Large_Language_Models_for_Direct_Interaction_with_SysML_v2
11. Dou, S., Liu, Y., Jia, H., Xiong, L., Zhou, E., et al., S.: Stepcoder: Amélioration de la génération de code avec l'apprentissage par renforcement à partir des retours du compilateur. arXiv préprint arXiv:2402.01391 (2024), https://arxiv.org/abs/2402.01391
12. Estefan, J.A.: Survey of Model-Based Systems Engineering (MBSE) Methodologies (2008)
13. GitHub, I.: Github copilot (2025), https://github.com/features/copilot, accessed: 2025-03-12
14. Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N.V., Wiest, O., Zhang, X.: Large language model based multi-agents: A survey of progress and challenges (2024), https://arxiv.org/abs/2402.01680
15. Hao, H., Zhou, L., Liu, S., Li, J., Hu, S., Wang, R., Wei, F.: Boosting large language model for speech synthesis: An empirical study (2023), https://arxiv.org/abs/2401.00246
16. Huang, L., Yu, e.a.: A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. ACM Transactions on Information Systems **43**(2), 1–55 (Jan 2025). https://doi.org/10.1145/3703155, http://dx.doi.org/10.1145/3703155
17. Jiang, J., Wang, F., Shen, J., Kim, S., Kim, S.: A survey on large language models for code generation (2024), https://arxiv.org/abs/2406.00515
18. Li, C., Flanigan, J.: Task contamination: Language models may not be few-shot anymore (2023), https://arxiv.org/abs/2312.16337
19. OpenAI: Gpt-4 technical report (2023), https://openai.com/research/gpt-4
20. Shojaee, P., Jain, A., Tipirneni, S., Reddy, C.K.: Execution-based code generation using deep reinforcement learning. arXiv preprint arXiv:2301.13816 (2023), https://arxiv.org/abs/2301.13816

21. Team, G., Mesnard, T., et al., C.H.: Gemma: Open models based on gemini research and technology (2024), https://arxiv.org/abs/2403.08295
22. Vaswani, A., Shazeer, e.a.: Multi-agent language models for low-resource language generation. In: Proceedings of the 38th International Conference on Machine Learning (ICML 2021). pp. 1234–1245 (2021), https://arxiv.org/abs/2104.04625
23. Wu, Q., Bansal, G., Zhang, J., Wu, e.a.: Autogen: Enabling next-gen llm applications via multi-agent conversation. arXiv preprint arXiv:2308.08155 (2023), https://arxiv.org/abs/2308.08155
24. Zhang, R., Zhao, W., Eger, S.: How good are llms for literary translation, really? literary translation evaluation with humans and llms (2025), https://arxiv.org/abs/2410.18697

A Agents prompts

A.1 `SpecificationGeneratorAgent`

System Prompt

You are an extractor-generator agent. You take a description in natural language. You return a python dictionary of packages. The format should be :

```
{
  "Package" : "PackageName",
  "attributes" : {
    "attribute1" : "attributeName", ...
  },
  "constraints" : {
    "constraint1" : "constraintName", ...
  },
  "requirements" : {
    "requirement1": "requirement", ...
  }
}
```

Each requirement should belong to a package. Each package should have a relevant name. Each requirement should have a description.

###{Example Description 1 : Example Dict 1} ###

⋮

###{Example Description k : Example Dict k} ###

Table 1. `SpecificationGeneratorAgent` system prompt and few-shot examples.

A.2 TemplateGeneratorAgent

System Prompt

You are a SysML V2 template generator. You take a list of requirements and generate a SysML V2 template containing the requirements. You never return a JSON string. You write only the textual SysML V2 code that you encapsulate between ''' and '''. Example of SysML V2 code is:

```
package package_name { part part_name { attribute attribute_name; } }
```

Table 2. `TemplateGeneratorAgent` system prompt.

A.3 WriterAgent

System Prompt

You are a SysML V2 code generator. You use your knowledge of SysML V2 to complete a template with a list of requirements. You take a list of requirements and a SysML V2 template as input. You should never add or remove requirements. You should never change the template's structure, only complete it with parts, attributes, constraints, actions, and other relevant elements. You write only the textual SysML V2 code that you encapsulate between ''' and '''.
Example of SysML V2 code:
`package package_name { part part_name { attribute attribute_name; } }`
Another example of SysML V2 code:
`package package_name { part part_1; part part_2 { attribute attribute_name; } }`
You ask the `syntax_checker_agent` to check the syntax using only the function provided.

Table 3. `WriterAgent` system prompt (for use with `TemplateGeneratorAgent`).

System Prompt

You are a SysML V2 code generator. You use your knowledge of SysML V2 to generate a SysML v2 code from a list of requirements. You take a list of requirements as input. You should never add or remove requirements.
You write only the textual SysML V2 code that you encapsulate between ''' and '''.
Example of SysML V2 code:
`package package_name { part part_name { attribute attribute_name; } }`
Another example of SysML V2 code:
`package package_name { part part_1; part part_2 { attribute attribute_name; } }`
You ask the `syntax_checker_agent` to check the syntax using only the function provided.

Table 4. `WriterAgent` system prompt (for use without `TemplateGeneratorAgent`).

A.4 ParserAgent

System Prompt
You are a SysML V2 code parser. You use your knowledge of SysML V2 to check the syntax of the textual code provided by the user. You never return a JSON string. You write only the textual output that you encapsulate between ''' and '''. Example of output: ''' the SysML V2 code contains no error ''' Another example of output: ''' Your code contains error: Error: Unexpected token 'alias' ''' You use the provided function to check the syntax of the code. You do not use any other function.

Table 5. `ParserAgent` system prompt.

B Templating Rules

The template organizes requirements into **packages**, where each package groups related requirements. The general structure is:

```
package <package_name> {
    doc /* This is the package containing the requirements */
    requirement <requirement_name> {
        doc /* <requirement_description> */

        attribute <attribute_name> = <value> <units>;

        require constraint {
            <constraint_formula>
        }
    }
}
```

The specific rules are defined as follows:

- **Package Definition:** Each package is defined using:

```
    package {{ package_name }} {
        doc /* This is the package containing the
           requirements */
```

- **Requirement Definition:** Each requirement inside a package is declared using:

```
    requirement {{ req.name }} {
        doc /* {{ req.Description }} */
```

– **Attributes (Optional):** If a requirement contains attributes, they are included as:

```
{%- for attribute in req.Attributes %}
attribute {{ attribute.Attribute }} = {{ attribute.
    Value }} {{ attribute.Units }};
{%- endfor %}
```

– **Looping Through Packages and Requirements:** The template iterates over all packages and their corresponding requirements using:

```
{%- for package_name, reqs in packages.items() %}
{%- for req in reqs %}
```

The final output is stripped of trailing whitespace to ensure a clean SysML v2 code format.

C Scenarios

Each scenario follows a structured format:

– **Type:** Indicates that the entry is an input request.
– **Content:** Specifies the requirements for a bicycle or its components in natural language.

1. **Mountain Bike Specification**

```
{
    "type": "input",
    "content": "Write me a specification for a
        mountain bike that has:

    - An aluminum frame that weighs less than 3 kg.

    - A frame with a pronounced sloping design.

    - A cassette with 9 cogs ranging from 11 to 42
        teeth.

    - An aluminum handlebar with a width of 720 mm."
}
```

2. **Electric Bike Specification**

```
{
    "type": "input",
    "content": "Write me a specification for an
        electric bike that has:
```

```
        - An aluminum hardtail frame suitable for light
            off-road use.

        - 27.5-inch wheels.

        - A 380Wh lithium battery."
    }
```

3. **Tire Specification**

```
    {
        "type": "input",
        "content": "Write a specification for tires that
            must be knobby, sized 24x1.95,
        and support pressures between 2 and 3.5 bars."
    }
```

4. **Mountain Bike with Specific Drivetrain**

```
    {
        "type": "input",
        "content": "Write me a specification for a
            mountain bike that meets the following
            requirements:

        - The front suspension must have 50 mm of travel
            to absorb terrain irregularities.

        - The drivetrain must have 7 speeds with an easy
            trigger shifter.

        - The rear derailleur must support a 14/34
            freewheel.

        - The crankset must have 34 teeth and 152 mm crank
             arms."
    }
```

5. **Bicycle Fork Specification**

```
    {
        "type": "input",
        "content": """Write me a specification for a bike
            fork made of aluminum with a 1\" 1/8 Aheadset
            pivot."
    }
```