Proceedings of the International Conference on
Formal Methods and Foundations of Artificial Intelligence
Eszterházy Károly Catholic University
Eger, Hungary, June 5–7, 2025
pp. 78–89
DOI: 10.17048/fmfai.2025.78

# LLM-based framework to support the construction of valid formal models

**Gábor Guta**[a], **Gábor Kusper**[b]

[a]Axonmatics Kft.
gabor.guta@axonmatics.com

[b]Faculty at Eszterházy Károly University,
Mathematics and Informatics Institute,
gkusper@aries.ektf.hu

**Abstract.** The use of large language models (LLMs) in software development is becoming increasingly widespread, despite well-known concerns regarding their reliability. A significant risk arises from relying on poorly understood approximate solutions that may subtly introduce errors into the final system. A key barrier to the adoption of formal modeling – beyond the steep learning curve of formal specification languages – is the additional abstraction layer, which can be as difficult to maintain as the source code itself. This complexity persists even when the formal specification can generate code directly. Another challenge is that, while tools for verifying properties of formal models are well-established, the initial translation of a mental model into a formal one often results in invalid or imprecise representations.

We propose a tool which facilitates the validation of formal models generated by LLMs from natural language specifications. The validation process involves two steps: first, the formal model is translated back into natural language using a deterministic, easily verifiable rule-based method; second, the author of the original specification validates this reformulated version. This human-in-the-loop method mitigates the risks associated with LLM black-box generation by enabling explicit semantic verification of the model.

*Keywords:* LLM, formal specification, human-in-the-loop

## 1. Introduction

Recent advances in large language models (LLMs) have democratized access to code and specification generation. Their natural language interfaces make them

immediately useful to developers, designers, and even non-technical stakeholders. However, their logical consistency and reliability remain problematic, especially when applied in a business-critical environment.

Formal specification and modeling techniques, such as UML, TLA+, and Alloy, provide solid foundations for correctness, verification, and maintainability, but still have very limited adaptation. This is probably due to various challenges like:

- adoption is hindered by steep learning curves for language syntax and semantics;
- maintaining a separate formal abstraction layer alongside code can be costly and error prone;
- LLM-generated formal models can be both syntactically and semantically wrong; and
- current verification tools can prove properties of given formal models, but the critical failure point is in the translation of the mental model to the formal model, where subtle intent mismatches are introduced.

It would be straightforward to attempt to solve these challenges with LLM. This can be done by supporting the translation from natural language representation to formal notation and helping the examination of the generated models.

## 1.1. Motivational example

We start by writing a natural language specification for a simple web-shop application. This application development task adequately represents the typical real-world challenges of implementing a software product. There are many similar software products, and it offers a relatively simple domain model with several decision points. Then we have used this specification to test our tool and its ability to construct formal specifications and back-translation to natural language representation.

The specification is formulated according to the IEEE 830 standard. It contains 32 functional and 15 nonfunctional requirements.

**Listing 1.** Example requirement markdown.

```
### 3.2 Cart Management
**FR-5** Add item to cart with quantity
**FR-6** Update quantity / remove items
**FR-7** Recalculate totals including tax / shipping
**FR-8** Validate stock on each cart update
```

## 1.2. Objective

Our aim is to build a tool prototype to enable easy creation and maintenance of formal models from a natural language specification. Additionally, the tool must

help the users to gain understanding of the specified models' properties on how the ambiguity of their original natural language specification is interpreted.

The tool also must support the exploration of the challenges of a real-world system (not only support an over-simplified example). Under that we mean that user experience matters and supports the usual industrial software development practices.

# 2. Related work

In this section, we review the related developments and focus on the most recent literature. We focus on four main areas: whether similar approaches taken or not and with what result; how formal semantics of UML models looks like; how LLMs used for diagramming (this is basically the extension of the first topic from a different direction); finally what is the state-of-the-art on validating LLM results by human.

## 2.1. LLMs in formal specification

Recent studies [2, 6, 10] have shown that LLMs can produce formal artifacts such as pre / post conditions or Alloy / TLA + models directly from natural language, but suffer from

- hallucinations and incomplete constraints;
- lack of traceability to original intent.

Approaches like nl2spec and SpecGen introduce prompt engineering and structured intermediate formats to improve correctness.

## 2.2. Formal semantics of UML diagrams

Multiple formalisms exist for interpreting UML class diagrams. Mathematical and denotational semantics [14] for associations, generalization, and constraints. Description Logic mappings [3] that enable formal reasoning. Another popular UML diagram type that is formalized is the sequence diagram [12].

## 2.3. Diagramming with the help of LLMs

LLMs can generate UML diagrams directly from plain English descriptions or even interpret images to create formal models, dramatically reducing manual effort in model-driven engineering. Recent frameworks and tools, like UMLAI and DiagrammerGPT< [8], leverage LLMs to automate diagram synthesis, support various types of UML, and allow fast iteration from specification to visualization. Meanwhile, empirical studies and surveys highlight both the capabilities and limitations of

LLMs in diagram generation, highlighting their growing role in requirements engineering and software design tasks [5, 9]. Generating a meaningful natural language description of UML diagrams is also not a trivial task [4].

### 2.4. Human-in-the-loop (HITL) validation

Recent work by Qi et al. (2025) explores the use of ChatGPT for conducting System-Theoretic Process Analysis (STPA) in safety-critical domains, benchmarking its effectiveness against human experts and highlighting the necessity of human validation for trustworthy analysis. Their findings demonstrate both the promise and the current limitations of LLM-based safety assurance, underscoring the challenges related to reliability, prompt engineering, and the need for future standardization and regulation in this field. [13].

### 2.5. Low-code, no-code and AI-coding approaches

Low-code and no-code approaches have a long tradition [1]. Some notations and tool-chains have been highly successful in domain-specific contexts, such as LabView [15] and its derivatives for measurement automation or BPMN tools for enterprise workflow automation [11]. In general, such tools often require either custom-developed extensions or embedded scripts written in standard programming languages. More recently, these tools have been extended with AI-based automatic code-generation capabilities, as seen in platforms like Claude [7]. These systems can be viewed as modern successors to the earlier practice of copying and pasting code from online development forums (e.g., Stack Overflow). While the graphical representations of no-code/low-code tools often limit the expressiveness and generality of the tool-chain, AI-coding approaches, in turn, suffer from the "almost-works" problem of partially understood code fragments.

## 3. Our solution

In software development practice, formal or semi-formal modeling notation can be used additionally to support development. We are prototyping a tool that is capable of managing informal natural language specification and formal UML specifications jointly. UML is chosen as the formal specification notation widely used in industrial practice, and various groups described its unofficial formal semantics. In this paper, we start with informal natural language specification structured into requirement items and this turned into UML various types and abstraction level of UML models. UML models are translated back to natural language and displayed along the original specification item to allow the user to check whether the diagram reflects the original intention. The workflow is described in Section 4.

# 4. Our approach

We present a two-step LLM-assisted process with human-in-the-loop validation that ensures semantic correctness when converting informal requirements into formalized models. The idea is that we use a generic LLM with prompt-based configuration to derive the specification. In the next step, the models are converted back to the representation of natural language 1.

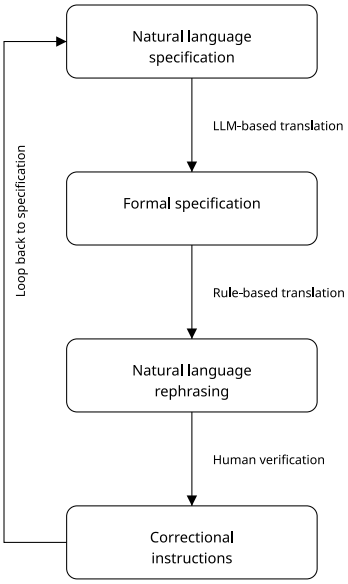


**Figure 1.** The main workflow of the system.

The tool is also designed to store the specifications, keep the traceability information, and a change history.

Use case diagrams can describe the high-level structure of the specification; class diagrams can be used to describe the static structure of the concepts of the specification; and finally, sequence diagrams can describe the dynamic behavior. These three types of diagrams provide a rich set of examples for translating specifications into high-level models. Additionally, they provide cross-checking between use-case diagram, class diagrams, and sequence diagrams (e.g., whether an existing method is called or not). The relation between the models is shown in Figure 2. These diagram types can be extended further in the future with additional diagram types and additional notation such as deployment diagrams or wireframes, respectively. Detailed class diagrams and sequence diagrams can be used to generate source code without human intervention. During the prototyping it became evident that the tool must maintain relationships between the models and must be able to update source models when the generated models changed, or to modify related models to provide plasticity.

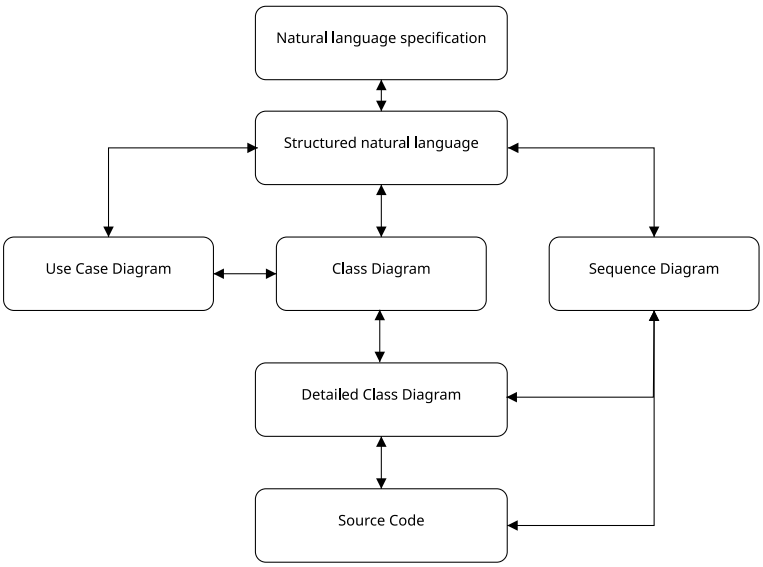


**Figure 2.** Hierarchy of the models.

In addition to high-level functional requirements, we must emphasize the importance of usability aspects of the tool. The primary goal of our approach is to make formal modeling easier to use. This implies that the tool must also have an easy-to-use interface. In addition, it must be implemented with the latest technologies/libraries as this also eases a deployment and installation of the tool, i.e. the user does not have to hunt for exotic compilers or libraries.

## 4.1. Implementation

Python programming language was chosen for implementation and LangChain was chosen as the core library to implement our system. The system uses PlantUML format to store UML diagrams and its tools to generate visual representation. Currently, we used Streamlit as the web application framework. To demonstrate what model validation looks like, we show a fragment of the generated UML diagram and its description displayed along the original requirements. The fragment of the PlantUML representation of the class diagram can be seen in Listing 2. The JSON data structure fragment demonstrates how the UML diagram template-based description is mapped back to the original requirements for validation, can be seen in Listing 3.

**Listing 2.** Example PlantUML class diagram.

```
@startuml
' Diagram type: class
left to right direction
```

```
package "Catalog package" {
  class Item {
    +id: UUID
    +sku: String
    +name: String
    +description: Text
    +price: Money
    +taxClass: String
    +stockQty: int
    +isActive: boolean
    +createdAt: DateTime
    +updatedAt: DateTime
    --
    ' FR-1,FR-2,FR-4,FR-5,FR-20,FR-21,FR-23,NFR-9
  }
}

package "Cart package" {
  class Cart {
    +id: UUID
    +sessionId: String
    +customerId: UUID
    +currency: String
    +subtotal: Money
    +taxTotal: Money
    +shippingTotal: Money
    +grandTotal: Money
    +updatedAt: DateTime
    +recalculateTotals(): void
    +validateStock(): boolean
    --
    ' FR-7,FR-8,NFR-4
  }

  class CartItem {
    +id: UUID
    +itemId: UUID
    +nameSnapshot: String
    +unitPrice: Money
    +taxClassSnapshot: String
    +qty: int
    --
    ' FR-5,FR-6,FR-8
  }
}

Cart "1" o-- "0..*" CartItem
```

```
CartItem "*" --> "1" Item


@enduml
```

**Listing 3.** Example mapping of the UML natural language represenation to the original requirements.

```
[
    ...
    {
        "id": "FR-5",
        "original_specification": "Add item to cart with
            quantity",
        "specification_to_compare": "Item carries 10 fields -
            id (UUID), sku (String), name (String),
            description (Text), price (Money), taxClass
            (String) and 4 more. Operationally, it offers a
            minimal API surface: no public operations. In
            terms of relationships, Item the model sets clear
            expectations about connected entities Item
            references zero or more CartItem.
        CartItem carries a concise set of fields - id (UUID),
            itemId (UUID), nameSnapshot (String), unitPrice
            (Money), taxClassSnapshot (String) and qty (int).
            Operationally, it offers a minimal API surface: no
            public operations. In terms of relationships,
            CartItem the model sets clear expectations about
            connected entities CartItem references exactly one
            Item. CartItem has exactly one Cart. Overall, the
            object stays cohesive: the fields align with its
            intended role and the relations define its
            lifecycle within the shopping flow."
    },
    {
        "id": "FR-6",
        "original_specification": "Update quantity / remove
            items",
        "specification_to_compare": "CartItem carries a
            concise set of fields - id (UUID), itemId (UUID),
            nameSnapshot (String), unitPrice (Money),
            taxClassSnapshot (String) and qty (int).
            Operationally, it offers a minimal API surface: no
            public operations. In terms of relationships,
            CartItem the model sets clear expectations about
            connected entities CartItem references exactly one
            Item. CartItem has exactly one Cart."
    },
    {
```

```
        "id": "FR-7",
        "original_specification": "Recalculate totals
            including tax / shipping",
        "specification_to_compare": "Cart carries 9 fields -
            id (UUID), sessionId (String), customerId (UUID),
            currency (String), subtotal (Money), taxTotal
            (Money) and and 3 more. Operationally, it offers a
            minimal API surface: recalculateTotals returning
            void and validateStock returning boolean. In terms
            of relationships, Cart the model sets clear
            expectations about connected entities Cart has
            zero or more CartItem. Cart has exactly one
            Customer."
    },
    {
        "id": "FR-8",
        "original_specification": "Validate stock on each cart
            update",
        "specification_to_compare": "Cart carries 9 fields  -
            id (UUID), sessionId (String), customerId (UUID),
            currency (String), subtotal (Money), taxTotal
            (Money) and and 3 more. Operationally, it offers a
            minimal API surface: recalculateTotals returning
            void and validateStock returning boolean. In terms
            of relationships, Cart the model sets clear
            expectations about connected entities Cart has
            zero or more CartItem. Cart has exactly one
            Customer."
    },
    ...
]
```

# 5. Discussion

Although the key parts of our idea are straightforward, its implementation reveals fundamental challenges which are not widely addressed by the research community. To demonstrate these challenges, we performed three experiments with the example problem described in Section 1.1.

## 5.1. Testing existing tools

The first is to use available LLM tools to test their capabilities to generate specification and applications from our requirement. This is a relatively well-researched area (as mentioned in Subsection 2.1) and although the tools are very capable of generating formal artifacts. The quality of these artifacts is often questionable: typically fragments of learning samples are recognizable in these artifacts, and the

internal structure is not consistent. Most of these problems can be addressed with extensive prompting.

### 5.2. Formal to NL representation

The second key experiment was developing a rewriting-based algorithm to translate the diagrams back into natural language. This problem is typically solved by using LLM as they have the capability to incorporate domain knowledge. This is a no-go option for us due to our explainability requirement. In this experiment our key finding was that just mechanistically translating back the models to natural language is not sufficient even if the textual representation is nicely formulated.

### 5.3. Testing our tool

Managing requirements and generating models (UML diagrams) from natural-language specifications integrated into a tool-chain operate in a manner similar to boxed chat products. The chat-based model and the requirement management functionality also perform well. The challenge lies in whether the relevant part of the natural-language representation operates at the same level of abstraction as the original specification. If this is not the case, it can be confusing for users for two reasons: first, compact concepts are described as mechanical enumerations, and second, only a single model-specific aspect is presented. To improve this, our tool needs further fine-tuning in model extraction and model-to-NL template generation. We must invest further effort in refining LLM prompts to generate formal models and extract additional information (e.g., identifying which models can best represent the content of a given requirement) that is needed for more accurate mapping. The natural-language requirements also demand more sophisticated templates and potentially conceptual definitions from formal reference ontologies (e.g., those defined by the OMG). Currently, we support only the validation of the forward-engineering approach - i.e., the tool is not yet capable of propagating model changes back to the requirements. Managing such bidirectional consistency would require significant additional development of the underlying system.

## 6. Conclusion and future work

Based on our initial experiments described in the previous section, it is evident that our approach works, but we have identified two main areas of functionality that require further research. The first area is model-to-natural-language (model-to-NL) translation, which requires a more deterministic mapping between low-level structural patterns and high-level conceptual constructs to make it easier to align the original requirements with the resulting model. For model-to-NL translation, we are eager to experiment with traditional description-logic-based knowledge-engineering methods and ontologies to ensure that the model translated back into natural-language representation remains at a comparable level of abstraction. The

second area concerns the management and visualization of updates to interrelated model elements, for which we plan to explore and adapt state-of-the-art model-matching approaches.

We introduced a framework combining LLM power and human validation to enable the usage of formal models derived from natural language requirements. Although building such a tool was a promising experiment, it currently lacks the capability to be used in practice.

# References


[1] M. Ajimati, V. Mohan, J. Wang, M. Ojo: *Low-code/no-code (LCNC): A systematic literature review and future research agenda*, Journal of Information Security and Applications 79 (2024), p. 103738, doi: 10.1016/j.jisa.2024.103738, url: https://www.sciencedirect.com/science/article/abs/pii/S0164121224003443.

[2] A. Beg, D. O'Donoghue, R. Monahan: *Formalising Software Requirements using Large Language Models*, arXiv preprint 2506.10704 (2025), ADAPT Annual Conference (AACS2025), poster presentation, doi: 10.48550/arxiv.2506.10704.

[3] D. Berardi, D. Calvanese, G. D. Giacomo: *Reasoning on UML Class Diagrams*, Artificial Intelligence 168.1-2 (2005), pp. 70–118, doi: 10.1016/j.artint.2005.05.001, url: https://www.sciencedirect.com/science/article/pii/S0004370205000792.

[4] H. Burden, R. Heldal: *Natural Language Generation from Class Diagrams*, in: Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa), 2011, pp. 70–76, doi: 10.1145/2095654.2095665.

[5] A. Conrardy, J. Cabot: *From Image to UML: First Results of Image Based UML Diagram Generation Using LLMs*, arXiv preprint (2024), doi: 10.48550/arXiv.2404.11376, url: https://arxiv.org/abs/2404.11376.

[6] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, C. Trippel: *nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models*, arXiv preprint 2303.04864 (2023), Logic in Computer Science (cs.LO); Artificial Intelligence (cs.AI); Machine Learning (cs.LG), doi: 10.48550/arXiv.2303.04864.

[7] Y. Dong, X. Jiang, J. Qian, T. Wang, K. Zhang, Z. Jin, G. Li: *A Survey on Code Generation with LLM-based Agents*, arXiv preprint arXiv:2508.00083 (2025), url: https://arxiv.org/abs/2508.00083.

[8] E. Inc.: *DiagramGPT – AI diagram generator created by Eraser*, https://www.eraser.io/diagramgpt, 2025.

[9] H. Islam: *Introducing UMLAI: Generate UML Diagrams from Natural Language Descriptions*, https://dev.to/hamidul_islam_ca3e9d4201e/introducing-umlai-generate-uml-diagrams-from-natural-language-descriptions-42bo, 2025.

[10] S. K. Jha, S. Jha, P. Lincoln, N. D. Bastian, A. Velasquez, R. Ewetz, S. Neema: *Counterexample Guided Inductive Synthesis using Large Language Models and Satisfiability Solving*, arXiv preprint 2309.16436 (2023), arXiv version; presented at MILCOM 2023, doi: 10.48550/arxiv.2309.16436.

[11] T. Lopes, S. Guerreiro: *A literature review on techniques for BPMN testing and formal verification*, Business Process Management Journal 29.8 (2023), pp. 133–162, doi: 10.1108/BPMJ-11-2022-0557, url: https://linkconsulting.com/eprocess/wp-content/uploads/sites/9/2025/04/Assessing-business-process-a-literature-review.pdf.

[12] Z. Micskei, C. Waeselynck: *The many meanings of UML 2 Sequence Diagrams: a survey*, Software and Systems Modeling 10.4 (2011), pp. 489–514, doi: 10.1007/s10270-011-0213-4, url: https://mit.bme.hu/~micskeiz/papers/micskei-waeselynck-semantics-2011.pdf.

[13] Y. Qi, X. Zhao, S. Khastgir, X. Huang: *Safety analysis in the era of large language models: A case study of STPA using ChatGPT*, Machine Learning with Applications 19 (2025), p. 100622, ISSN: 2666-8270, DOI: 10.1016/j.mlwa.2025.100622, URL: https://www.sciencedirect.com/science/article/pii/S2666827025000052.

[14] M. Szlenk: *Formal Semantics and Reasoning about UML Class Diagram*, in: Proceedings of the International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX), 2006, pp. 51–59, DOI: 10.1109/DEPCOS-RELCOMEX.2006.27.

[15] X. Zhao, J. Sun, S. Goh, S. Pu: *An empirical study on modeling challenges from LabVIEW community: A systems modeler perspective*, Journal of Systems Architecture 148 (2024), p. 103567, DOI: 10.1016/j.sysarc.2024.103567, URL: https://www.sciencedirect.com/science/article/abs/pii/S2590118424000273.