

TTool-AI: a Large Language Model-based Assistant for Model Driven Engineering

Bastien Sultan^{1*} and Ludovic Apvrille¹

^{1*}LTCI, Telecom Paris, Institut Polytechnique de Paris, 450 route des Chappes, Sophia Antipolis, 06410, France.

*Corresponding author(s). E-mail(s): bastien.sultan@telecom-paris.fr;
Contributing authors: ludovic.apvrille@telecom-paris.fr;

Abstract

Throughout the history of engineering, successive innovations have been implemented to assist engineers in their tasks, allowing them to concentrate on high-value activities while minimizing time-consuming and error-prone tasks. Large Language Models (LLMs) represent one of these innovations, with significant potential for developing new kinds of engineering assistants, as demonstrated by a rich body of recent literature. The paper introduces TTool-AI, a model-driven engineering assistant based on LLMs and integrated within the SysML modeling and formal verification toolkit, TTool. TTool-AI enables system architects to generate and incrementally refine various types of SysML diagrams directly from textual specifications with a single click. The core mechanisms of TTool-AI—contextual knowledge injection, automated prompt generation, and iterative feedback—enable it to produce good quality models that can serve as a sound foundation for system architects in MDE processes. Building on our previous work presented at MODELSWARD 2024, this paper provides a comprehensive description of TTool-AI’s MDE assistance features. It introduces new functionalities, including requirement engineering and automated model mutation generation. An evaluation of these features, comparing their performance against Master-level students, demonstrates the tool’s efficacy and suggests a strong potential to significantly enhance engineering productivity by enabling engineers to focus on high-value tasks.

Keywords: Generative AI, Large-Language Models, SysML, Model Driven Engineering, Mutations

1 Introduction

Throughout history, system engineering, architecture, design and development has benefited from successive innovations, transforming the profession of engineers by providing increasingly sophisticated assistance, allowing them to focus on tasks with higher added value. Naval architects, once performing complex hydrodynamic and stability calculations by hand, have benefited from successive introductions of calculation aid—logarithm tables, then computers—, and then CAD software to draw their plans, relieving them

of time-consuming tasks and enabling them to refocus their time on tasks more oriented towards more conceptual thinking. Similarly, a goal towards which much research in systems engineering, and particularly in Model-Driven Engineering (MDE), is directed is the automatic generation of (drafts of) models from a textual specification. While several proposals have been made, notably relying on natural language processing [19], the recent introduction of large language models (LLMs) opens up very promising perspectives for MDE assistance [12]. We are therefore once again on the threshold of a new transformation in system engineering, architecture, design and development,

thanks to a new technological innovation enabling the emergence of a new assistance tool for engineers.

However, the use of LLMs as engineering assistants requires prior adaptation of these tools, as their “raw” use produces (sometimes very) imperfect results [9]. This article contributes to this adaptation by presenting TTool-AI, an MDE assistant based on the SysML modeling and formal verification toolkit TTool¹, integrated with OpenAI’s GPT [22]. TTool-AI offers a variety of MDE features, including the ability to generate different types of SysML diagrams with a single click, based on system specifications written in natural language by the user. Additionally, it can automatically incrementally modify these models according to user-described desired evolutions. This tool is primarily intended to help system architects save time by producing good quality initial models, providing a solid basis for further refinement. As a result, engineers can further focus their efforts on high-value engineering tasks. This article extends a conference paper [4] which introduced a subset of the TTool-AI features. The enhancements include a comprehensive description of the MDE assistance features of TTool-AI—where [4] focused only on the generation of Block Diagrams (BDs) and State Machine Diagrams (SMDs) from textual specifications. In particular, it introduces new features such as requirement engineering assistance and assistance in incremental refinement of models through automated generation of model mutations. In addition, it features an evaluation that uses a broader basis and includes new diagrams and engineering tasks.

The rest of the paper is organized as follows: Section 2 provides an overview of the literature on the automated generation of models from textual specifications, with a focus on the introduction of LLMs as assistants in MDE. Sections 3 and 4 present our contributions. Section 3 provides an overview of TTool-AI, detailing its current assistance capabilities and the three key mechanisms—contextual knowledge injection, automatic prompt generation and automated feedback loop—underlying its functionality and common to all TTool-AI features. Afterwards, Section 4 presents in more concrete terms a subset of five TTool-AI features illustrating how software and system engineers can utilize our framework during the (incremental) design of a system. Section 5 evaluates several TTool-AI features by comparing their performance with that of Master-level students on various

engineering tasks, and discusses the relevance of our contributions. Section 6 finally concludes the paper.

2 Related Works

2.1 Generation of Models from Specifications

The automatic generation of (formal) models from system specifications has long been a significant research challenge. As highlighted in the comprehensive literature review by Landhäuser [19], this area of study has been active since the late 1990s. However, model generation often requires imposing constraints on the syntax of input requirements or necessitates manual preprocessing, as demonstrated by Gelhausen et al. [17]. Recent advancements, such as the ARSENAL framework [18], have introduced model generation approaches that minimize restrictions on the input language. Nonetheless, even with these advanced tools, certain natural language expressions can still pose challenges, preventing their automated transformation into formal models. We believe that recent advancements in generative AI models, such as GPT, offer an opportunity to handle system specifications written in natural language. Relying on these AI models, as the following sections will illustrate, helps reduce the research effort on language processing and redirects it toward tailoring LLMs to suit the requirements of the modeling process.

More broadly, the topic of modeling assistants is not new in research and engineering. In a comprehensive survey conducted by Savary-Leblanc et al. [27], which reviewed papers published between 2010 and 2022, the authors identified 11 notable papers introducing tools aimed at aiding engineers in the model design process. Among these papers, four specifically focused on UML models, with one addressing SysML models by introducing a tool that supports the design of use-case diagrams [5]. Furthermore, recent research has explored the development of AI-based Model-Based Systems Engineering (MBSE) assistants within the growing trend of AI-based methods and tools. In this context, Chami et al. [10] introduced a framework based on natural language processing that autonomously generates SysML use-case and block diagrams from textual requirements. Additionally, Schröder et al. [28] introduced three distinct AI-based MBSE assistants, each tailored to a specific task. The first is a workshop assistant that transforms hand-drawn sketches into formal SysML models. The

¹<https://ttool.telecom-paris.fr>

second is a knowledge-based assistant that generates design suggestions using training data derived from a set of existing models. The third is a chatbot capable of interpreting natural language queries related to modeling and responding in natural language.

2.2 Integrating LLMs into MDE

The introduction of GPT-3.5 in 2022 opened the way to a new kind of assistants. LLMs have shown capabilities far beyond basic question-and-response tasks and they soon have been adapted to tackle various challenges, including MDE support. Notably, they can generate responses tailored to the needs of model designers, such as producing UML diagrams with minimal syntax errors [9]. In this study, Camara et al. evaluated ChatGPT’s ability to create UML class diagrams from natural language specifications. By examining the responses to 40 distinct modeling tasks, they found that ChatGPT often produced syntactically correct models. However, the semantic accuracy, especially regarding class relationships, was inconsistently achieved. Thus, generating an accurate model with ChatGPT often requires iterative refinement. The authors concluded that significant user effort is still necessary. Our contribution addresses this issue through an automated response analysis and feedback loop, resulting in substantial time savings. Netz et al. [21] propose an approach that shares some commonalities with our strategy for improving LLM responses. Specifically, they prompt the LLM with a description of the target modeling language’s syntax. The LLM then generates responses according to this syntax. These responses are subsequently parsed using a grammar-based parser, ensuring that only syntactically correct segments of the models are constructed.

In a more architecture process-oriented study, Ahmad et al. [1] used ChatGPT for software architecture tasks, including requirements generation, UML modeling, and architectural evaluation. Their findings underscore ChatGPT’s utility as an assistant for software architects but also raises concerns about response variability and ethical/intellectual property issues. In both studies, human analysis and iterative questioning remain crucial for arriving at a correct system architecture.

Petrovic et al. [24] introduce an LLM-based MDE assistant designed to generate code from natural language software system specifications. This assistant operates in two stages: model generation and code

generation, with the former being particularly relevant to our discussion. During the model generation stage, the tool produces an Ecore model of the system by iteratively sending prompts to the LLM. These prompts are automatically generated and incorporate the specification along with additional contextual knowledge, which varies depending on the stage of model generation. This contextual knowledge may include the Ecore metamodel, a description of the hardware on which the software system will be executed, a set of reference requirements, ... The responses from the LLM are then validated for consistency using OCL rules. This assistant relies on two of the mechanisms—contextual knowledge injection and response verification—that are central to our contribution.

Additionally, at earlier design stages, LLMs have been assessed for generating goal models [11, 20], showing promising results when used judiciously, such as incorporating feedback or using multiple prompts. Alarcia et al. [2] introduced a GPT-4-based assistant designed to generate a detailed list of technical specifications from a natural language description provided by the user. Their study highlights the significance of prior knowledge injected into the LLM: the results showed that the assistant produced superior technical specifications when an Object-Process Methodology [15] model of the system was included as part of the input to the modeling assistant.

3 TTool-AI: an Overview

3.1 TTool: Context

TTool is a comprehensive MDE open-source toolkit supporting formal modeling, model verification (simulation and model-checking) and code generation from models. It supports a wide variety of SysML diagrams [23], from requirements and analysis diagrams (such as use case (UCDs), activity and sequence diagrams) to design diagrams (such as BDs and SMDs). Additionally, TTool extends these models with attack-defense graphs [3]. TTool diagrams are underlined by formal semantics: among others, this enables for direct model checking of SysML models without the need of translating them in intermediate formalisms [8]. This is also highly useful when it comes to use LLMs to generate these diagrams, since it provides a basis for defining, verifying and enforcing formal consistency rules [29]. This is also a foundation for AMULET, the TTool integrated mutation language [30] that enables

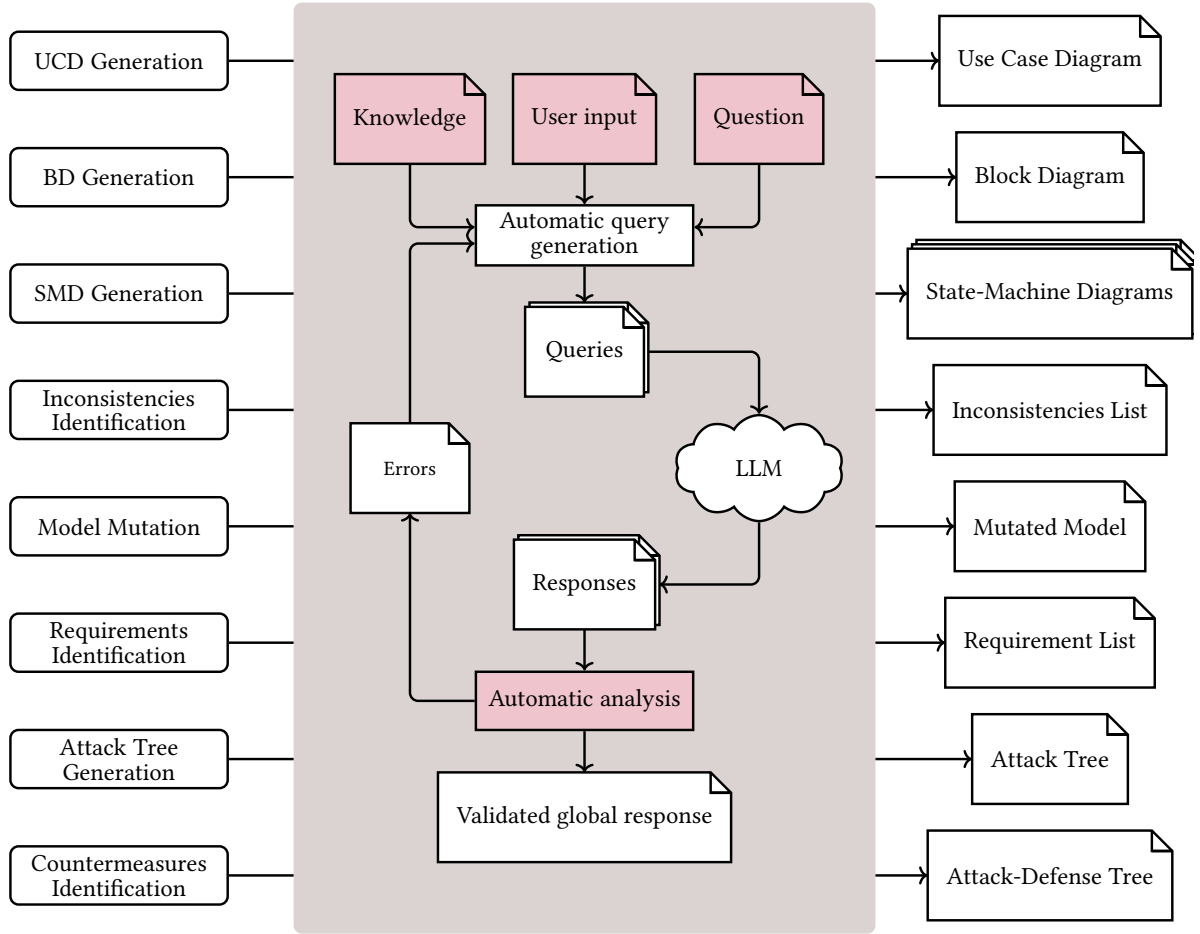


Fig. 1: Overview of TTool-AI. The central section illustrates the generic workflow of our framework. Tasks and data that vary based on the selected TTool-AI feature are depicted in light red boxes. These features are listed on the left side of the figure, while the corresponding models generated by TTool-AI for each feature are shown on the right side, aligned with their respective feature heights.

for scripting the modifications a user can bring to BDs and UCDs. TTool-AI, the contribution we present in this paper, extends TTool with an AI-based assistant that is intended to assist the TTool’s users in producing various kinds of diagrams supported by the tool, including most of the SysML diagrams and attack-defense graphs, or AMULET mutation scripts for supporting incremental MDE processes.

3.2 A Polymorphic Assistant

TTool-AI is designed as a versatile system engineering assistant that can support system architects in a wide range of model-based tasks, including system analysis (e.g., deriving use cases from a textual specification) and system design (e.g., creating SysML

block diagrams). Its main goal is to automatically generate initial models that users can subsequently refine, offering two main benefits:

- **Efficiency:** by handling time consuming and sometimes low-value modeling tasks, TTool-AI enables engineers focusing their time and expertise on refining the more complex aspects of the models.
- **Variability:** TTool-AI can propose radically different architectures for the same specification. Indeed, creating radically new designs from scratch after having designed an initial architecture can be challenging for engineers. The stochastic nature of LLMs’ responses facilitates this variability, aiding in the exploration of diverse architectural solutions.

Currently, TTool-AI offers the following features (see Figure 1):

- UCD generation [29]: generates use case diagrams from a system specification written in natural language.
- BD generation [4]: generates block diagrams from a system specification written in natural language.
- SMDs generation [4]: generates state-machine diagrams—one per block—from a block diagram and a system specification written in natural language.
- Inconsistencies identification [29]: identifies semantic inconsistencies between different SysML views modeling the same system (e.g., determining whether a concept defined in one view and expected in another is missing from that second view). Currently, our implementation supports inconsistencies between UCD, BD, and SMD.
- Model mutation: updates BD and SMDs based on an initial BD+SMDs set and a textual query describing the desired modification (e.g., “add an integer attribute to every block and delete all the links between block B0 and block B1”).
- Requirements identification: generates a list of requirements from a system specification written in natural language.
- Attack Tree generation [6]: generates attack trees given a system specification written in natural language.
- Countermeasures identification [7]: enriches an attack tree with mitigation leaves, given an attack tree.

These different generations are performed thanks to a standardized automated workflow. The base principle of this workflow involves: (i) automatically forging queries (prompts) based on the feature selected by the user, (ii) analyzing the LLM’s responses for syntactic and semantic accuracy, and automatically reiterating the questions to the LLM, including the detected errors, until they converge to correct responses, and (iii) when applicable, parsing the correct responses to automatically generate new graphical models. This workflow relies on three key mechanisms, which are detailed below.

3.3 Three Key Mechanisms

3.3.1 Preliminary considerations: TTool/OpenAI interfacing

TTool-AI fundamentally relies on automated interactions between TTool and a LLM. To comprehend the framework’s functionality, it is essential to first understand this interaction mechanism. Therefore, we will initially explain how TTool interacts with the LLM. Following this, we will be able to introduce the three core mechanisms central to our contribution. Although the details provided here are specific to OpenAI’s GPT API, the overall interaction mechanism is applicable to any other LLM that can be queried through such an API.

GPT can be easily accessed via simple HTTP requests using its API. This involves sending a structured request that includes the API key, the desired GPT model, and a JSON message with the question. For instance, the LLM can be queried using `curl` in this way:

#1: Querying GPT through the API

```
curl https://api.openai.com/v1/chat/
completions -H "content-type:
application/json" -H "
Authorization: Bearer OPENAI-API
-KEY" -d '{ "model": "gpt-4o", "
messages": [{"role": "user", "
content": "Hello! Do you like
reading SN Computer Science?"}]
}'
```

We then receive the following response in JSON format, which includes the answer provided by the LLM:

Getting an answer

```
{ "id": "chatcmpl-9
ofN0oXU81393MUj5UisOv9il8DSy",
"object": "chat.completion",
"created": 1721862454,
"model": "gpt-4o-2024-05-13",
"choices": [
{
"index": 0,
"message": {
"role": "assistant",
```

```

        "content": "Hello! As an AI,
                    I don't have personal
                    likes or dislikes, but I
                    can certainly help you
                    with information about SN
                    Computer Science. Are
                    you looking for specific
                    articles, topics, or
                    general information about
                    this journal?"
    },
    "logprobs": null,
    "finish_reason": "stop"
  }
],
"usage": {
  "prompt_tokens": 17,
  "completion_tokens": 43,
  "total_tokens": 60
}
}

```

We can see from this example that automating the querying of the LLM and parsing its responses to extract the answer is very simple. Our integration of TTool with GPT functions similarly; instead of using `curl`, we directly connect with OpenAI servers and utilize the API to query the LLM via HTTP requests. To enhance transparency and support user understanding of the automated exchanges between TTool and the LLM, the full sequence of queries and responses is accessible within the TTool GUI. These interactions are displayed in real time and remain available for review afterwards.

3.3.2 Mechanism 1: knowledge injection

The interfacing between TTool and GPT opens the way for the three mechanisms we need for building TTool-AI. The first mechanism, we call “knowledge injection” and that is closely related to retrieval-augmentation techniques [14], consists in sending contextual knowledge to the LLM. This knowledge forms a part of the query forged by TTool-AI (see Figure 1), and is based on the feature selected by the user. For instance, if the user selects the UCD generation feature, contextual knowledge about use case diagrams is automatically included in the query sent to the LLM. This contextual knowledge is necessary for providing the LLM with the information it needs

to produce an initial exploitable response. TTool relies indeed on concepts that are likely not included in the LLM’s training data, such as, for instance, details related to the syntax and semantics of the SysML profiles used in TTool or the syntax and semantics of its SysML mutation language. Below is an example demonstrating the importance of this knowledge injection. First, we start a chat and ask GPT to generate a command in AMULET, the SysML mutation language used in TTool [30]:

#2: Generating a mutation command

Please write an AMULET command that deletes block B0.

GPT then answers by inventing a syntax and referencing an AMULET tool which is not relevant to our context:

Response to #2

Certainly! AMULET is a tool commonly used in computer systems for configuration and management purposes. To delete block B0 using an AMULET command, you need to refer to the specific syntax and procedures of the platform you’re working on. However, a generic command structure might look something like this:

```

```shell
amulet delete block B0
```
[...]

```

We then initiate a new chat and start by sending some prior knowledge on AMULET:

#3: Adding some context on AMULET

AMULET is a SysML mutation language. To delete a block myBlock from a model, the AMULET syntax is:

```
rm block myBlock.
```


When you are asked to provide an AMULET command, please provide only the command, without any comments nor explanations.

Afterwards, we reiterate question #2:

Response to #3 then #2

```
rm block B0
```

However, providing the LLM with knowledge about fundamental aspects alone is not sufficient to obtain exploitable responses. We also aim for the LLM to deliver structured responses to ease their parsing and presentation to the TTool user in textual or graphical model formats. Therefore, it is also necessary to inject constraints regarding the form of the responses through this knowledge injection mechanism. Therefore, depending on the feature selected by the user, specific form constraints are included in the knowledge sent to the LLM. Concretely, for most features, we have defined specific JSON formats that the LLM is asked to follow when providing its answers. For instance, for a use case diagram, we inject the following knowledge about form:

#4: Knowledge about use case diagrams form

1. When you are asked to identify actors, return them as a JSON specification formatted as follows: {actors: [{ "name": "Name of actor"...}]}
2. When you are asked to identify the use cases from a specification, return them as a JSON specification formatted as follows: {usecases: [{ "name": "Name of use case"...}]}

3. When you are asked to identify connections between use cases and actors, return them as a JSON specification formatted as follows: {connections: [{ "element1": "usecase or actor name", "element2": "usecase or actor name"...}]}

3.3.3 Mechanism 2: automatic query generation

As previously mentioned, the injected knowledge is part of a query sent to the LLM. As depicted in Figure 1, this query also includes two other elements: a **user input** and a **question**.

The **user input** refers to the information provided by the user, which varies depending on the selected feature. For instance, for the UCD generation feature, the user input consists of the textual specification of the system for which the diagram is being created.

The **question** is a standard sentence included by TTool-AI in the query. It also depends on the feature selected by the user. For example, in the case of UCD generation, the question is:

#5: Questions related to use case diagrams generation

1. From the provided system specification, using the specified JSON format, identify the typical actors, that is elements not part of the system but directly interacting with it. Do respect the JSON format, and provide only JSON (no explanation before or after)
2. From the provided system specification, using the specified JSON format, identify the typical use cases of the system. Do respect the JSON format, and provide only JSON (no explanation before or after).

3. From the previous JSON and system specification, find the typical connections between actors and use case, and between use cases. Do give them respecting the specified JSON format.

At the end, the query includes: (i) contextual knowledge, (ii) the question, and (iii) the user input. It is important to note that, to prevent overloading the LLM with a large amount of information at once (which could (1) exceed its input context limit in number of tokens and (2) cause it to “lose focus” or ignore parts of the input), for most TTool-AI features, the query is sliced and sent in several successive parts. For instance, for the UCD generation feature, knowledge #4 and question #5 are divided and sent in three successive queries, which mirror the logical progression of the UCD generation process (first identifying the actors, then the use cases, and finally the relationships between them). This slicing strategy is similar to the one employed in *least-to-most* prompting techniques [31].

3.3.4 Mechanism 3: automated feedback loop

The last core mechanism of TTool-AI is its automated feedback loop. This feedback loop is based on the fact that LLMs rarely produce a perfect response to an initial query on the first attempt. Typically, querying LLMs involves a series of successive questions to refine the initial response. The feedback loop mimics this iterative process by algorithmically analyzing the responses provided by the LLM and iteratively reformulating the query to address the potential detected errors. The algorithmic analysis identifies two categories of errors in the responses:

- **Response structure errors:** for features that require a JSON-structured response (i.e., all features except the model mutation feature), the response is checked for syntactic errors in the structure of the JSON array.
- **Constraints errors:** the response is also checked for violations of a set of constraints depending on the selected feature [29]. These rules cover syntactic aspects (e.g., for model mutation, does the AMULET script comply with AMULET syntax?) or semantic aspects (e.g., in the AMULET script

line `rm block B0`, does B0 exist in the targeted diagram?).

This analysis/re-querying loop continues until the responses converge to a point where either (i) the algorithmic analysis detects no errors or (ii) a maximum number of loop iterations is reached, to prevent infinite loops. If the loop terminates due to reaching this iteration limit, users can examine the interaction history (including all intermediate queries and responses) provided in the TTool-AI interface to determine how best to proceed.

Algorithm 1: Automated feedback loop

Data: String r (initial LLM response), int $maxIterations$

Result: String r_i (improved LLM response)

```

1 String query  $\leftarrow \epsilon$ 
2  $err_{json} \leftarrow listOfJSONFormatErrorsIn(r)$ 
3  $err_{const} \leftarrow listOfConstraintsErrorsIn(r)$ 
4 while  $card(err_{json}) + card(err_{const}) >$ 
    $0 \vee maxIterations > 0$  do
5   query  $\leftarrow$  “In the previous response” +  $r$  +
     “the following errors were found, do
     correct them:”
6   foreach element  $e$  in  $err_{json}$  do
7     | query  $\leftarrow$  query +  $e$ 
8   end
9   foreach element  $e$  in  $err_{const}$  do
10    | query  $\leftarrow$  query +  $e$ 
11  end
12   $sendToLLM(query)$ 
13   $r \leftarrow NewLLMResponse$ 
14   $err_{json} \leftarrow listOfJSONFormatErrorsIn(r)$ 
15   $err_{const} \leftarrow listOfViolatedConstraintsIn(r)$ 
16   $maxIterations \leftarrow maxIterations - 1$ 
17 end
18  $r_i \leftarrow r$ 
```

Algorithm 1² provides a structured description of the feedback loop. Currently, the feedback loop is implemented for the following features:

- UCD generation
- BD generation
- SMDs generation
- Model mutation
- Attack-tree generation

² ϵ designates the empty string.

- Countermeasure identification

3.3.5 Graphical model generation

Once valid JSON arrays are provided by the LLM at the output of the feedback loop, TTool-AI parses them to generate both textual and graphical models, with the graphical models automatically rendered in the TTool GUI.

4 TTool-AI: a Practical Insight

This section now offers a practical overview of several TTool-AI features based on the mechanisms discussed hereinabove. To illustrate these features effectively, we will use a running example that demonstrates their utility and outputs in a concrete way.

4.1 Running Example



Fig. 2: One of the fins used for active stabilization, deployed from the hull of a passenger ship [16]

Our running example is an active fin stabilizer system, a stabilization system deployed on modern vessels. A ship at sea is indeed subject to strong environmental stresses. Among these, rolling is the most detrimental, as it can cause difficulties for the ship to maintain its course, create discomfort for the crew and passengers, damage the cargo due to violent roll/righting movements, and even in extreme cases lead to capsizing. To tackle these issues, active fin stabilizers are often used for roll compensation. This kind of stabilizer relies on deployable fins located in pairs on both sides of the hull. In the system we consider as a running example in the paper and inspired by the systems presented in [16], the movement of these fins is controlled by actuators, managed by a controller. This controller develops commands based on data received

from a gyrometer and an inclinometer, which enable it to detect the ship's rolling movements and heeling. It also generates commands for the retraction and deployment of the fins. For this purpose, the system is equipped with a control console that enables the crew for commanding the retraction or deployment of the fins.

To illustrate the various TTool-AI features involved in an MDE process, we will walk through a toy MDE process using this system. For this purpose, we have developed a textual specification for the fin stabilizer, which closely mirrors the previous paragraph (see Figure 5). This specification will serve as the input for the TTool-AI features presented.

4.2 Assisting SysML Analysis and Design

4.2.1 Analysis stage: requirements identification

We start with the analysis of the specification. An essential initial step for an engineer at the analysis stage is to derive requirements from the specification. For this purpose, we utilize the requirement identification feature of TTool-AI. This feature requires the user to input the textual specification, and forges a query including this specification, a question tasking the LLM with the generation of a list of requirements, and knowledge defining the expected JSON format. The JSON contains an array of requirements, each requirement containing the following fields: name, unique identifier, description (the requirement itself), and relationships with other requirements (composition, derivation, refinement). Below is an excerpt from the array returned by TTool-AI for our example:

Requirement JSON array

```
{
  "name": "
    Deployable_Fins_Installation
  ",
  "id": "R01",
  "doc": "The system shall have
    deployable fins located in
    pairs on both sides of the
    hull.",
  "compose": [],
  "derive": ["Fin Deployment
    Control"],
```

```

"refine": ["Dynamic Angle
Adjustment"]
},
[...]
```

Overall, the generated list—which accurately mirrors the system specification—contains the following requirements:

Requirement List

- R01 The system shall have deployable fins located in pairs on both sides of the hull.
- R02 The deployable fins shall dynamically change their angle of attack to compensate roll and heeling.
- R03 The movement of the fins shall be controlled by actuators.
- R04 The controller shall manage the actuators.
- R05 The controller shall compute angle of attack commands based on data received from a gyrometer and an inclinometer.
- R06 The system shall use a gyrometer to detect the ship's rolling movements.
- R07 The system shall use an inclinometer to detect the ship's heeling.
- R08 The controller shall generate commands for the retraction and deployment of the fins.
- R09 Each time a command is computed by the controller, it shall be sent to the actuators.
- R10 The system shall be equipped with a control console that allows the crew to command the retraction or deployment of the fins.

4.2.2 Analysis stage: UCD generation

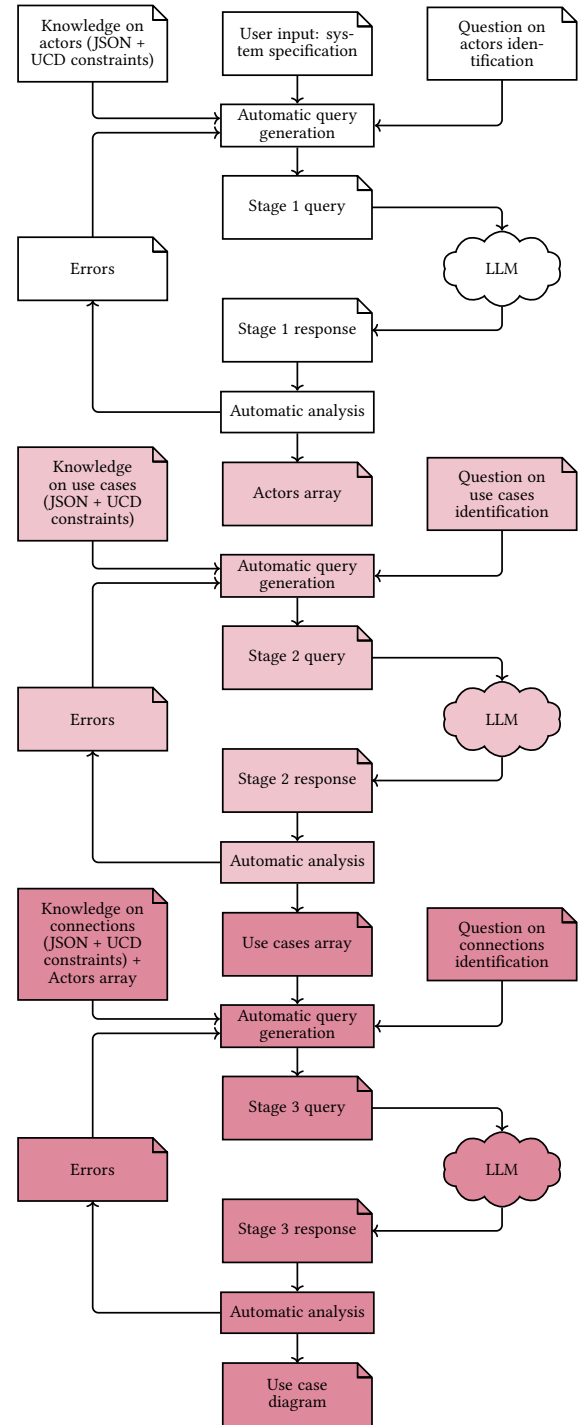


Fig. 3: UCD generation process

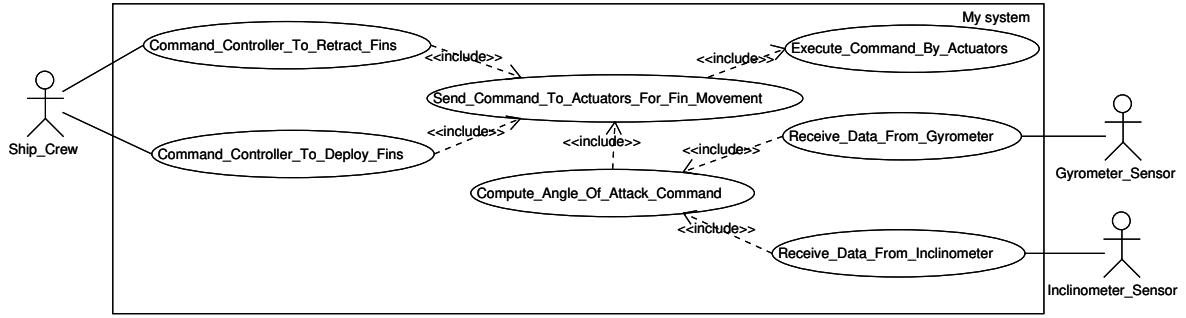


Fig. 4: Generated UCD

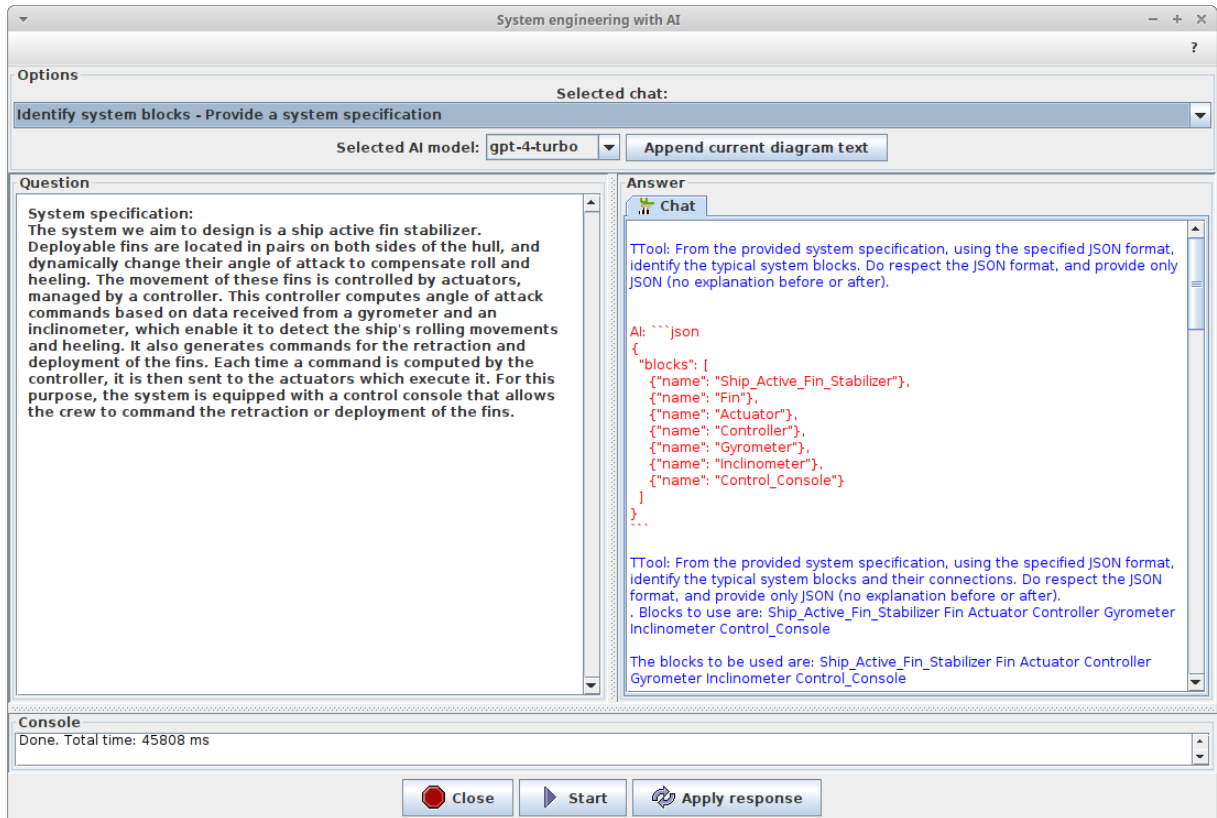


Fig. 5: TTool-AI Window. The top section contains the options menu, allowing users to select a TTool-AI feature (here, “Identify system blocks”) and the desired LLM (here, “gpt-4-turbo”). The central section is designated for interactions with the LLM. The bottom section includes command buttons, with “Apply response” initiating the process of parsing and drawing diagrams based on the LLM’s responses.

A second analysis task that TTool-AI can assist with is the definition of use cases from the system specification. The framework includes a UCD generation feature, initially introduced in [29], which operates in three consecutive stages as depicted in Figure 3.

- **Stage 1:** the user provides TTool-AI with the system specification. TTool-AI then constructs a query that includes this specification, a question asking the LLM to identify the actors in the use case diagram, and contextual knowledge about the expected

JSON format and naming rules³ for actors. Upon receiving the LLM’s response, TTool-AI initiates the analysis and feedback loop. The loop continues iteratively until either (i) the LLM produces an actor list that complies with a set of actors-related rules, or (ii) 20 iterations are completed. At the end of this stage, a validated JSON array containing the actors is built and TTool-AI automatically moves to Stage 2.

- **Stage 2:** TTool-AI follows a similar process to generate an array of use cases—except no user input is provided. The initial query includes a question that tasks the LLM with identifying use cases, and contextual knowledge about the expected JSON format and naming rules for use cases (the system specification has been provided at Stage 1 and is already included in the context of the LLM). By the end of this second stage, the framework produces a validated JSON array that includes the identified use cases.
- **Stage 3:** equivalent steps are then undertaken to generate an array containing the relationships between the use cases, and between the use cases and the actors. At the end of this stage, a JSON structure containing the full UCD is generated. The user can then choose to either generate a graphical UCD with a single click in the TTool-AI interface or ask the framework to refine the UCD by adding new questions in the interface (e.g., “add an actor representing the sensors”).

The UCD produced by TTool-AI for our running example (without human intervention) is depicted in Figure 4. Overall, it accurately mirrors the specification, although it would be beneficial to include an actor representing the actuators. Additionally, the *include* relationships involve the correct use cases, but they are drawn in the wrong direction.

4.2.3 Design stage: BD generation

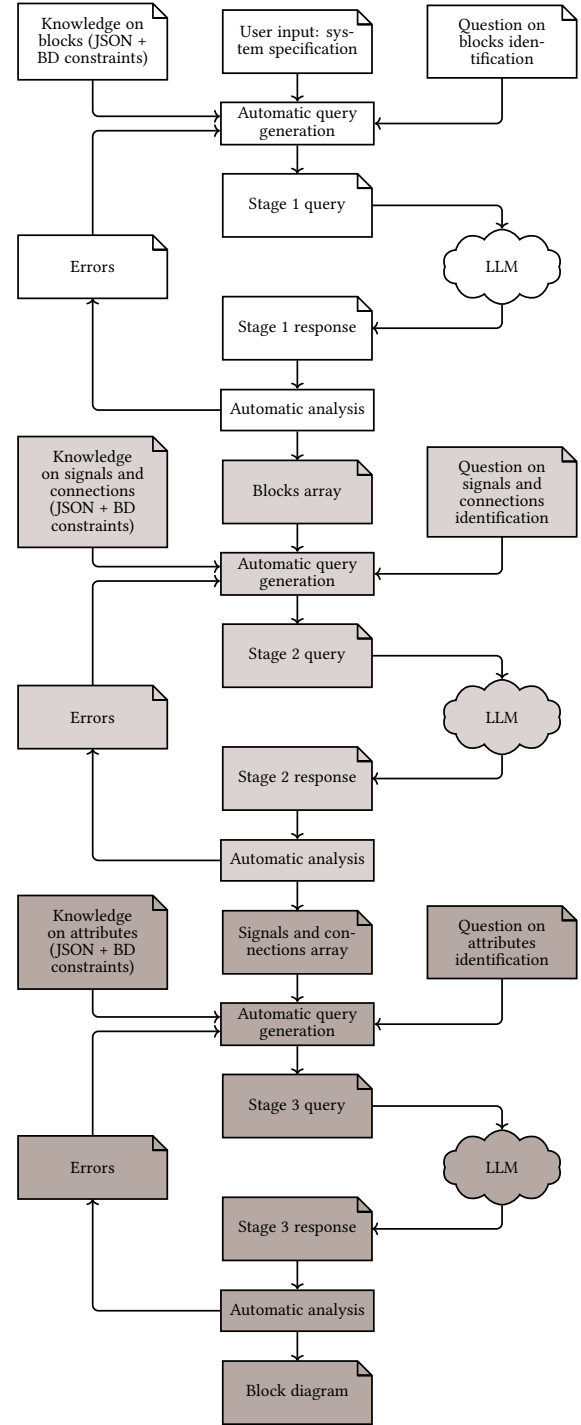


Fig. 6: BD generation process

³A comprehensive list of rules included in the UCD generation process, including these naming rules, is available in [29].

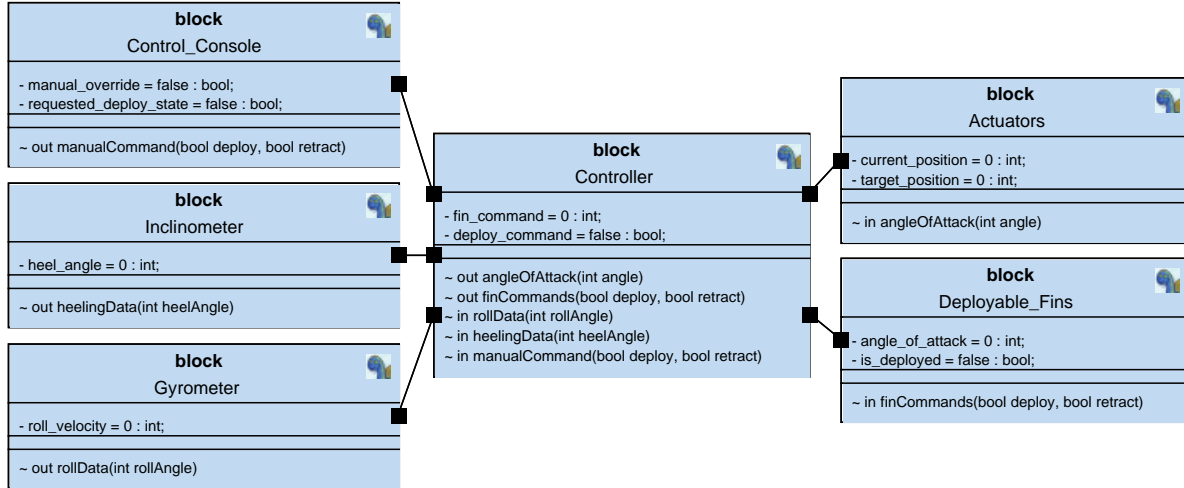


Fig. 7: Generated BD

TTool-AI can also assist system architects during the design stage. In SysML-based MDE methods, block diagrams are central to this stage. As TTool relies on a SysML profile that merges internal block diagram and block definition diagram into a single block diagram, TTool-AI can assist this design step by generating a block diagram from a textual specification. Similarly to the UCD generation feature, this function employs a sliced generation process operating in three consecutive stages (see Figure 6).

Figure 5 illustrates the interactions between TTool-AI and the LLM during the BD generation process. On the left, the user input (the system specification) is displayed. On the right, the automated interactions between TTool-AI and the LLM are shown. The question tasking the LLM with the block identification is displayed on the top. Below this, the LLM’s response—the JSON containing the identified blocks—is shown. Following this, our framework generates a question to task the LLM with identifying signals and connections. For the sake of user-friendliness, contextual knowledge injection occurs in the background and therefore is not displayed in the window.

Figure 7 displays the block diagram generated by the BD generation feature. A minor anomaly is that TTool-AI has included two separate blocks, “Actuators” and “Deployable.Fins”, which could be combined into a single block. However, TTool-AI includes a consistency feature that detects and corrects inconsistencies within and between diagrams. This feature,

which can be applied at this stage to improve and align the BD with the previously generated UCD, leverages both LLMs and formal consistency rules, as introduced in [29].

4.2.4 Design stage: SMDs generation

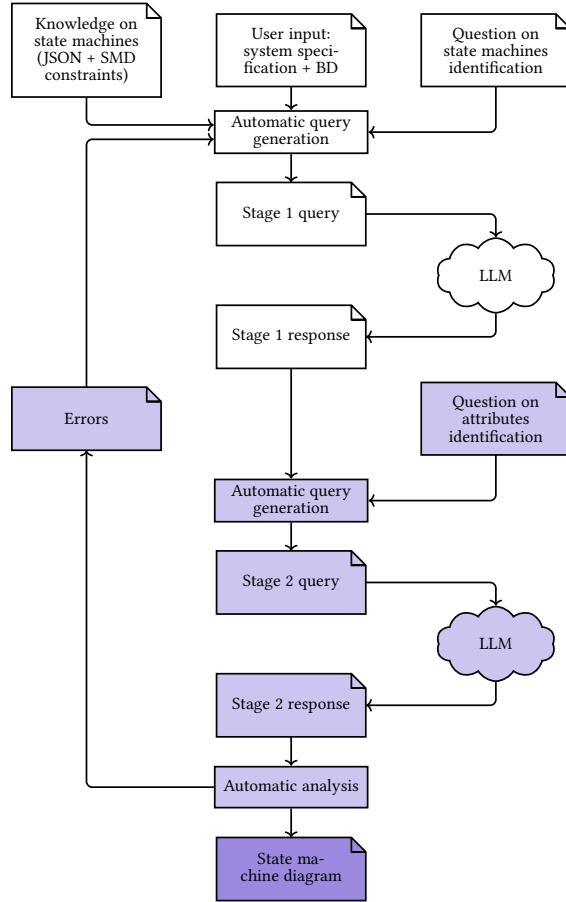


Fig. 8: SMD generation process

To complement the BD generation feature, TTool-AI also has an SMD generation feature that associates an SMD with each block of the generated BD. This feature requires the user to provide the system specification and select a block diagram in TTool’s GUI. For each block of the selected BD, a specific query is generated. This query includes the system specification, the block diagram in textual format, contextual knowledge on state machines, and a question formatted as follows:



```

From the system specification, and
from the definition of blocks
and their connections, identify
the state machine of block:
Control_Console
This block has the following
attributes:
bool manual_override = false
bool requested_deploy_state = false

This block has the following signals
:
out manualCommand(bool deploy,bool
retract)

```

The generation feature (see Fig. 8) operates in two consecutive stages: first, it generates the state machine, and second, it assigns types (integer or boolean) to all attributes involved in the generated state machine’s guards and actions. For each generated SMD, similar to the UCD and BD generation features, the response undergoes an analysis/feedback loop to help the LLM converge on a better answer. To illustrate this process, Figure 9 displays the SMD generated by TTool-AI for the block “Control_Console”. A syntactic error can be noted, as the signal “manualCommand()” does not include the two boolean attributes defined in the BD. Moreover, the states Start and Idle could be merged.

4.3 Supporting Incremental Design

Incrementality is a fundamental aspect of MDE processes, particularly in agile methodologies [25]. TTool supports such incremental methodologies, including SysML-Sec [26], which depend on continuous updates to SysML models. In recent years, TTool has been enhanced with features to support these incremental processes. One such enhancement is AMULET, a mutation language that enables for scripting all possible modifications of SysML models that were previously only possible via TTool’s GUI [30]. Additionally, a new model checking algorithm has been introduced to reduce the complexity of iterative formal verification [13]. TTool-AI model mutation feature is part of this effort.

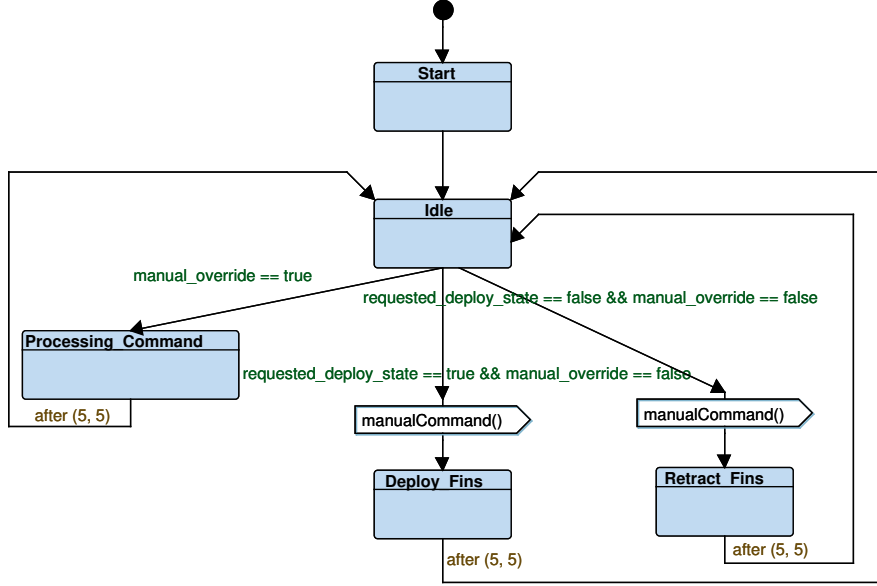


Fig. 9: Generated SMD

Table 1: Excerpt of AMULET syntax (provided in EBNF). From [30]

| Mutation | AMULET syntax | Examples |
|--------------------|--|---|
| Attribute addition | add attribute type attribute-Name [= intVal] in blockName
NB: type = int bool | add attribute bool myBool in myBlock
or
add attribute myInt x = 42 in myBlock |
| Attribute deletion | remove rm attribute attribute-Name in blockName | remove attribute myBool in myBlock |

4.3.1 Tailoring contextual knowledge on AMULET

AMULET enables for describing any model mutation using a syntax that closely resembles natural language. Table 1 provides a brief overview of this syntax. Contextual knowledge has been developed to enable GPT to “learn” AMULET. For instance, the two operators described in Table 1 are “taught” to GPT through the injection of the following knowledge:

Contextual knowledge on AMULET

Adding an integer attribute *i* in a block *b* is written "add attribute int *i* in *b*".
Adding a boolean attribute *x* in a block *b* is written "add attribute bool *x* in *b*".

Removing an attribute *a* from a block *b* is written "remove attribute *a* in *b*".

If we want to set an existing attribute *n* of a block *myBlock* to a value *x*, we will write "modify attribute *n* in *myBlock* to *x*".

Note that in the previous knowledge, we do not instruct GPT with the syntax enabling for adding an attribute and setting its value in a single line of code. Since AMULET already has an operator dedicated to setting an existing attribute to a specific value, composing these two operations (first adding the attribute, then setting its value) is semantically equivalent. To

avoid overloading the LLM with excessive knowledge to process, we have chosen to rely solely on the “value setting” operator for initializing newly created attributes.

4.3.2 Model mutation in practice

Overall, the model mutation feature follows the process presented in Figure 1. More in detail:

- **Query generation:** TTool-AI slices the query into several sub-queries. The first sub-query contains contextual knowledge about the syntax and semantics of AMULET. After sending this sub-query, another sub-query is generated containing the textual format of the diagram selected by the user in TTool’s GUI. For each user input (e.g., “remove all the integer attributes in blockName”), a sub-query containing only this input is sent to the LLM.
- **Response analysis:** the mutation feature also involves the automated analysis/feedback loop. Each time the LLM provides an answer, TTool-AI parses the response and extracts the AMULET code lines. It then attempts to compile these code lines using the AMULET compiler integrated into TTool. Potential error messages from the compiler are appended to iterative queries within the feedback loop until the LLM produces a correct AMULET script.
- **Response application:** once the validated script is received, if the user clicks on “Apply response”, the AMULET compiler compiles the script, and the modified model is displayed in TTool’s GUI.

We now provide an example of the model mutation feature in the context of our running example. Suppose we need to modify the generated Block Diagram (BD) shown in Figure 7. The goal is to align the attribute names in the Inclinator, Gyrometer, and ControlConsole blocks with their corresponding names in the signals. To achieve this, we input the following request into the TTool-AI window:

#6: Example user input for model mutation

In Inclinator, replace the attribute heel_angle with an attribute (int) heelAngle. Do the same in Gyrometer for roll_velocity (with rollAngle), and add two boolean attributes deploy and retract in ControlConsole.

We then obtain the following mutation script:

Response to #6

```
remove attribute heel_angle in
    Inclinator
add attribute int heelAngle in
    Inclinator
remove attribute roll_velocity in
    Gyrometer
add attribute int rollAngle in
    Gyrometer
add attribute bool deploy in
    ControlConsole
add attribute bool retract in
    ControlConsole
```

After clicking on “Apply response”, we finally obtain the updated block diagram displayed in Figure 10.

5 Evaluation

5.1 Evaluation Setup

The principle of our evaluation is to compare the performance of TTool-AI against Master-level students on identical engineering tasks. The experiment involved using TTool-AI (without any human modification of the generated diagrams) to answer the same final exam questions given to five cohorts of about 15 to 25 students, after they completed a 21-hour course on TTool and modeling and verification of embedded systems with SysML.

We conducted two series of evaluations of TTool-AI. The first series, carried out for the study reported in [4], involved using TTool-AI to generate diagrams based on three system specifications from three final

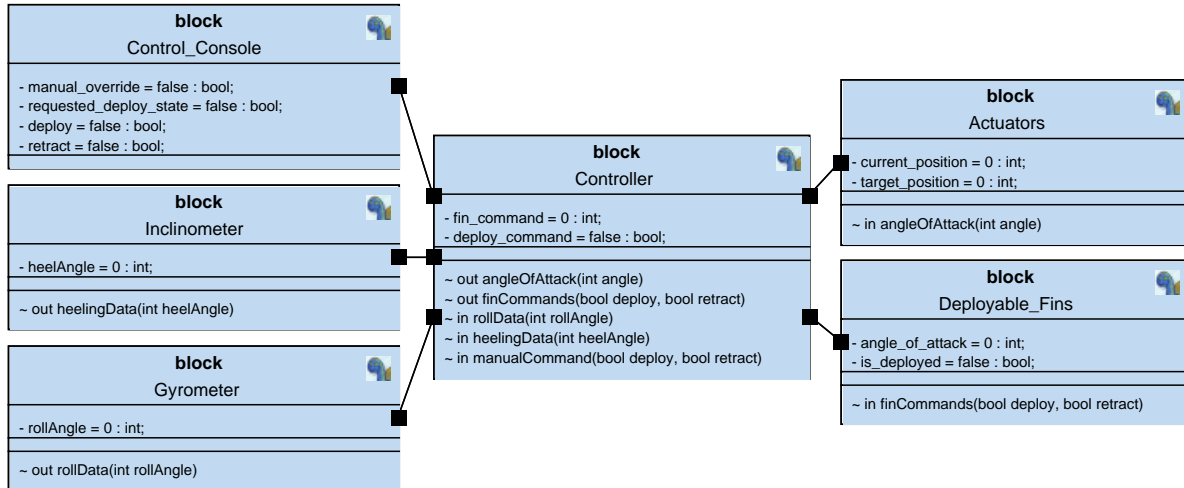


Fig. 10: Mutated BD

exam subjects (a platoon of rovers, an automated braking system, a spatial communication system). Specifically, TTool-AI generated 5 BDs and 5 SMDs for each of the 5 BDs using GPT-3.5-turbo as the underlying LLM. The second series, conducted for this paper, involved the following tasks:

- For the first specification (a packaging chain): generating 5 UCDs.
- For the second specification (a connected device): generating 5 UCDs, 5 BDs, and a set of SMDs for each BD.

In this second series, GPT-4o was used as the underlying LLM. Additionally, for each of the two exam subjects, we tasked the LLM with generating two sets of safety properties to verify based on the specifications and explaining the strategies to verify them (either by checking a CTL property or by adding an observer block to the BD). This question was also posed to the students. The prompt used for the properties identification is as follows:

#7: Tasking GPT-4o with properties identification

From the following system specification, and assuming you have a SysML block diagram modeling the system, list three properties to verify on the system. How would you proceed to verify them? (using a CTL property, an observer block, etc .)

The answers were subsequently graded. The grading criteria remained consistent for both TTool-AI and the students. These criteria adhere to the principles of software engineering quality criteria. They encompass:

- **Completeness:** the adequacy of the diagrams to the specification (does the proposed architecture meet the specification? Is the behavior of the state machines in line with this specification?)
- **Quality of abstraction:** how well the model captures the system specification at an appropriate level of abstraction? This relates to the number of blocks and states and the consistency of their naming, the number of connectors between blocks, the absence of attributes declared in blocks but not utilized in state-machine diagrams
- **Effective use of the modeling language constructs** to accurately represent system features, and the syntactic correctness of the models (i.e., the number

of errors and warnings detected by TTool’s syntax checker).

5.2 Comparative Evaluation and Discussion

Tables 2 and 3 summarize the results of the two evaluations. Table 2 corresponds to the evaluation initially presented in our previous article, using GPT-3.5-turbo as the underlying LLM, and Table 3 corresponds to the supplementary evaluation for which TTool-AI used GPT-4o. The complete dataset used for the evaluation, including the input specifications and the different models generated by TTool-AI, is available on a public Git repository ⁴.

Use case diagrams

TTool-AI produces complex UCDs (ranging from 5 to 15 use cases and 3 to 8 actors for the packaging line, and 7 to 13 use cases with 2 to 4 actors for the connected device) in about half a minute each time, with very low standard deviation. It is obviously impossible for a human engineer to diagram such a model in this time. Moreover, depending on the specification provided, the grades of the students and of TTool-AI are equivalent or higher for TTool-AI (8.7/10 compared to 6.2/10), showing that the produced diagrams are sound, as evidenced by the diagram generated for our running example in Section 4 (see Figure 4).

Block diagrams

Across both experiments, TTool-AI produced block diagrams for the following specifications: the platoon (2-6 blocks, 1-6 connectors), spatial system (10-16 blocks, 9-13 connectors), automated braking system (6-12 blocks, 0-11 connectors), and connected device (8 blocks, 2-6 connectors). The quality of generated diagrams is close to or slightly better than those generated by students. In the second experiment, with GPT-4o, syntax verification detects no errors. This is partly expected due to the contextual knowledge injection step that embeds consistency rules, and the diagram generation stage from the validated response that enforces some of these rules [29]. However, a recurring mistake is that TTool-AI uses attribute names in the signal signatures that often do not match the names of the attributes of the block. Currently, our consistency feature [29] often detects attributes that

are declared in a block but not used. Adding a consistency rule specifically tackling this issue to the rule set injected to the LLM and enforced at the diagram generation would eliminate these naming discrepancies. In addition, it should be noticed that generation times are consistently low (about one minute).

State-machine diagrams

The time needed to generate SMDs is higher than for UCDs and BDs, as TTool-AI generates as many SMDs as there are blocks in the provided BD. It is still satisfactory, with an average of about 3 minutes in both experiments for generating up to 8 SMDs. However, the quality of the diagrams produced is lower than that of the UCDs and BDs (this trend is also observed in student-generated diagrams). This may be due to the more complex semantics of these diagrams, particularly regarding transitions that involve various elements such as guards (logical tests), different types of actions (affectation of variables, emission or reception of signals), and timers. Additionally, TTool-AI with GPT-4o does not seem to perform better at this SMD generation task than TTool-AI with GPT-3.5-turbo, although it produced higher quality BDs. For the connected device, generated SMDs include 4 to 8 states per diagram, for the automated braking system, 2 to 7 states, for the platoon, 2 to 10 states, and for the spatial system, 3 to 11 states.

Safety properties

Another feature evaluated is property identification. TTool, which is primarily a verification toolkit, utilizes formally defined SysML profiles that enable for verifying the models against a set of CTL* properties. This verification stage is crucial when using an LLM-based assistant to eliminate responses that might potentially violate requirements. In this task often challenging for students, GPT-4o excelled, generating and explaining three properties in about 30 seconds. It consistently identified relevant properties and provided detailed explanations on how to verify them, offering sound CTL formulas or consistent strategies to add observer blocks and state machines to the diagrams. This promising experiment opens the way for a new TTool-AI feature dedicated to generating a set of CTL* properties based on input BD/SMDs and a list of requirements. However, it should be noted that the observed grading gap between GPT-4o and students on this specific task is somewhat biased, as some students did not attempt this question during the exam.

⁴<https://github.com/zebradile/ttool-ai>

Table 2: Comparative evaluation of TTool-AI (GPT-3.5-turbo) vs. Master students

| Metrics | | BD | | SMDs | |
|-------------|-----------|----------|------------|----------|------------|
| | | Students | TTool-AI | Students | TTool-AI |
| Grade (/10) | Average | 7 | 8.1 | 5.8 | 6.3 |
| | Std. Dev. | 2.6 | 1.6 | 3.2 | 1.5 |
| | Min | 0 | 4.5 | 0 | 3 |
| | Max | 10 | 10 | 10 | 9 |
| Time (s) | Average | <2700 | 40 | <2700 | 178 |
| | Std. Dev. | — | 10 | — | 97 |

Table 3: Comparative evaluation of TTool-AI (GPT-4o) vs. Master students

| Specification | Metrics | UCD | | BD | | SMDs | | Properties | | | |
|-----------------|------------------|-------------|-----------|----------|----------|----------|----------|------------|--------|------|-----|
| | | Students | TTool-AI | Students | TTool-AI | Students | TTool-AI | Students | GPT-4o | | |
| Packaging chain | Grade (/10) | Average | 7.9 | 7.7 | — | — | — | — | 5.3 | 8.4 | |
| | | Std. Dev. | 1.7 | 1.8 | — | — | — | — | 1.6 | 1.7 | |
| | | Min | 5 | 5 | — | — | — | — | 3.3 | 6.7 | |
| | | Max | 10 | 9.2 | — | — | — | — | 10 | 10 | |
| | | Median | 8.3 | 8.8 | — | — | — | — | 5 | 8.8 | |
| | Time (s) | Average | <900 | 34.8 | — | — | — | — | <900 | 34 | |
| | | Std. Dev. | — | 2.3 | — | — | — | — | — | 2.2 | |
| | Connected device | Grade (/10) | Average | 6.2 | 8.7 | 9.2 | 9.3 | 6.6 | 6.3 | 4.7 | 9.3 |
| | | | Std. Dev. | 1.6 | 1.3 | 1.2 | 1.1 | 1.1 | 0.5 | 1.9 | 0.7 |
| | | | Min | 3.3 | 6.7 | 7.5 | 7.5 | 5 | 5.8 | 1.7 | 8.8 |
| Max | | | 8.3 | 10 | 10 | 10 | 10 | 6.7 | 6.7 | 10 | |
| Median | | | 5.8 | 9.2 | 10 | 10 | 6.7 | 6.7 | 1.7 | 8.8 | |
| Time (s) | | Average | <900 | 31.8 | <2250 | 62.2 | <2250 | 201.6 | <900 | 31.4 | |
| | | Std. Dev. | — | 12 | — | 23 | — | 47.3 | — | 1.7 | |

Overall thoughts

Overall, the analysis of these results demonstrates that TTool-AI, even without human engineer intervention, generates high-quality diagrams that are as good as, or better than, those produced by students. However, the primary strength of the tool lies in the minimal time required to design these diagrams. As an assistant, TTool-AI excels in providing a more-than-correct initial draft for the system architect, allowing them to then refine the diagrams by focusing on finer details. However, while the quality of the automatically generated diagrams is satisfactory, they can still be improved. To help addressing some inconsistencies, users can utilize TTool-AI's consistency verification feature [29] but ultimately, it is always

the user's responsibility to assess the quality and relevance of the diagrams, and to refine them. In a few words, while TTool-AI cannot replace engineers, it serves as a valuable new class of engineering assistant, helping save time and quickly explore alternative architecture patterns.

5.3 Threats to Validity

It should be kept in mind that this evaluation, even if it provides a basis for discussing the relevance of our contributions, has several limitations.

5.3.1 Construct validity

The first one is that even if the evaluation criteria are well defined, their appreciation still bears a part of subjectivity: for instance, two different evaluators will not necessarily have the same criterion for evaluating readability of a diagram.

5.3.2 Internal validity

Another limit is the estimation of the time spent by the students to realize the different diagrams. Indeed, we evaluated this time according to the global time allotted for the exam and the respective difficulties of the questions. But we did not ask the students to write the time they spent on each diagram, therefore these estimated times are essentially indicative.

5.3.3 External validity

Finally, this evaluation compares engineers and a tool. The results likely underestimate the value of TTool-AI, which is designed to complement engineers, not replace them. To accurately assess TTool-AI's effectiveness as an assistant to human engineers, it should be interesting to evaluate two groups of engineers: one using only TTool, and the other using TTool-AI. This comparison will help measure the improvement brought by TTool-AI.

6 Conclusions

Extending our previous work [4], the paper presents a comprehensive overview of TTool-AI, an LLM-based assistant integrated within TTool. Designed to assist system architects in MDE processes, TTool-AI offers various features aimed at generating diverse types of models with a single click from a system specification written in natural language. Additionally, it enables for the incremental refinement of these models based on simple descriptions of the desired changes provided by the user. TTool-AI relies on core mechanisms common to its various features, including contextual knowledge injection, automated prompt generation, and iterative analysis and feedback of the LLM responses. These mechanisms aim to produce high-quality diagrams while sparing the user from manually checking the responses for syntactic and basic semantic errors. The results of the different evaluations, showing a good to excellent performance of TTool-AI across most of its features, suggest that it can be a highly valuable MDE assistant.

Future works include refining the queries generated for SMDs generation and formalizing and enriching of consistency rules for these diagrams, as has been done for BDs and UCDs [29]. Additionally, new consistency rules will be introduced and implemented to systematically address discrepancies in BDs between attribute names and signal parameter names. Another planned extension of TTool-AI is a new feature for generating CTL* properties based on system models and requirements. Furthermore, a promising area of research involves providing clear explanations or justifications for the design and modeling choices made by the assistant. Currently, TTool-AI provides the comprehensive interaction history between the LLM and the tool, enabling users to trace the reasoning process. However, this history can be lengthy and difficult to interpret. Further research will also focus on improving the feedback loop by introducing a mechanism (in the model generation features) for selecting the “best” generated models. Model quality would be assessed, for instance, using two criteria: (1) syntactic correctness and (2) semantic correctness with respect to the specification, verified through model checking. This direction is currently under investigation. Finally, TTool-AI would benefit from complementary evaluations, including comparative studies measuring the performance of professional engineers using standard TTool versus TTool-AI, as well as further assessments based on objective, quantitative metrics.

Declarations

Conflict of interest

On behalf of all authors, the corresponding author states that there is no conflict of interest.

Funding information

N/A

Authors contributions

Both authors contributed to the different aspects of this work, including the design of the contributions, their implementation in TTool, the evaluation, and the writing of the paper. All authors read and approved the final manuscript.

Data availability

The source code of TTool is available at <https://gitlab.telecom-paris.fr/mbe-tools/TTool> and the models generated by TTool-AI, and their detailed gradings, are available at <https://github.com/zebradile/ttool-ai>.

Research involving human and/or animals

N/A

Informed consent

N/A

References

- [1] Ahmad A, Waseem M, Liang P, et al (2023) Towards human-bot collaborative software architecting with ChatGPT. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, pp 279–285
- [2] Alarcia RMG, Russo P, Renga A, et al (2024) Bringing Systems Engineering Models to Large Language Models: An Integration of OPM with an LLM for Design Assistants. In: Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering-MBSE-AI Integration, pp 334–345
- [3] Apvrille L, Roudier Y (2016) SysML-Sec attack graphs: compact representations for complex attacks. In: Graphical Models for Security: Second International Workshop, GraMSEC 2015, Verona, Italy, July 13, 2015, Revised Selected Papers 2, Springer, pp 35–49
- [4] Apvrille L, Sultan B (2024) System Architects are not Alone Anymore: Automatic System Modeling with AI. In: Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024), INSTICC. SciTePress, pp 27–38, <https://doi.org/10.5220/0012320100003645>
- [5] Aquino ER, De Saqui-Sannes P, Vingerhoeds RA (2020) A Methodological Assistant for Use Case Diagrams. In: 8th MODELSWARD: International Conference on Model-Driven Engineering and Software Development, pp 1–11
- [6] Birchler De Allende A, Sultan B, Apvrille L (2024) Automated Attack Tree Generation Using Artificial Intelligence and Natural Language Processing. In: International Conference on Risks and Security of Internet and Systems, Springer, pp 141–156
- [7] Birchler De Allende A, Sultan B, Apvrille L (2024) From Attack Trees to Attack-Defense Trees with Generative AI & Natural Language Processing. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems. Association for Computing Machinery, New York, NY, USA, MODELS Companion '24, p 561–569, <https://doi.org/10.1145/3652620.3687804>, URL <https://doi.org/10.1145/3652620.3687804>
- [8] Calvino AT, Apvrille L (2021) Direct model-checking of SysML models. In: 9th International Conference on Model-Driven Engineering and Software Development, SCITEPRESS-Science and Technology Publications, pp 216–223
- [9] Cámara J, Troya J, Burgueño L, et al (2023) On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. Software and Systems Modeling pp 1–13
- [10] Chami M, Zoghbi C, Bruel JM (2019) A First Step towards AI for MBSE: Generating a Part of SysML Models from Text Using AI. A First Step towards AI
- [11] Chen B, Chen K, Hassani S, et al (2023) On the Use of GPT-4 for Creating Goal Models: An Exploratory Study. In: 2023 IEEE 31st International Requirements Engineering Conference Workshops (REW), IEEE, pp 262–271
- [12] Combemale B, Gray J, Rumpe B (2023) ChatGPT in software modeling. Software and Systems Modeling 22(3):777–779
- [13] Coudert S, Apvrille L, Sultan B, et al (2024) Incremental and Formal Verification of SysML Models. SN Computer Science 5(6):714
- [14] Di Rocco J, Di Ruscio D, Di Sipio C, et al (2025) On the use of large language models in model-driven engineering: On the use of large language... Softw Syst

- Model 24(3):923–948. <https://doi.org/10.1007/s10270-025-01263-8>, URL <https://doi.org/10.1007/s10270-025-01263-8>
- [15] Dori D (2011) Object-process methodology. In: Encyclopedia of knowledge management, second edition. IGI Global Scientific Publishing, p 1208–1220
 - [16] Fincantieri (2024) Fin stabilizers system. URL https://www.fincantieri.com/globalassets/prodotti-servizi/sistemi-e-componenti/sistemi-e-componenti-navali/fin-stabilizers_mp-03-14.pdf
 - [17] Gelhausen T, Tichy WF (2007) Thematic Role Based Generation of UML Models from Real World Requirements. In: International Conference on Semantic Computing (ICSC 2007), pp 282–289, <https://doi.org/10.1109/ICSC.2007.13>
 - [18] Ghosh S, Elenius D, Li W, et al (2016) ARSENAL: automatic requirements specification extraction from natural language. In: NASA Formal Methods: 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings 8, Springer, pp 41–46
 - [19] Landhäuser M, Körner SJ, Tichy WF (2014) From requirements to UML models and back: how automatic processing of text can support requirements engineering. Software Quality Journal 22:121–149
 - [20] Nakagawa H, Honiden S (2023) MAPE-K Loop-Based Goal Model Generation Using Generative AI. In: 2023 IEEE 31st International Requirements Engineering Conference Workshops (REW), IEEE, pp 247–251
 - [21] Netz L, Reimar J, Rumpe B (2024) Using Grammar Masking to Ensure Syntactic Validity in LLM-based Modeling Tasks. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. Association for Computing Machinery, New York, NY, USA, MODELS ’24
 - [22] OpenAI (2023) GPT4 Technical Report. Tech. rep., URL <https://cdn.openai.com/papers/gpt-4.pdf>
 - [23] Pedroza G, Apvrille L, Knorre D (2011) AVATAR: A SysML environment for the formal verification of safety and security properties. In: 2011 11th Annual International Conference on New Technologies of Distributed Systems, IEEE, pp 1–10
 - [24] Petrovic N, Pan F, Lebioda K, et al (2024) Synergy of Large Language Model and Model Driven Engineering for Automated Development of Centralized Vehicular Systems. arXiv preprint arXiv:240405508
 - [25] Riesener M, Doelle C, Perau S, et al (2021) Methodology for iterative system modeling in agile product development. Procedia Cirp 100:439–444
 - [26] Roudier Y, Apvrille L (2015) SysML-Sec: A model driven approach for designing safe and secure systems. In: 2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), INSTICC, pp 655–664
 - [27] Savary-Leblanc M, Burgueño L, Cabot J, et al (2023) Software assistants in software engineering: A systematic mapping study. Software: Practice and Experience 53(3):856–892
 - [28] Schröder E, Bernijazov R, Foullois M, et al (2022) Examples of AI-based Assistance Systems in context of Model-Based Systems Engineering. In: 2022 IEEE International Symposium on Systems Engineering (ISSE), IEEE, pp 1–8
 - [29] Sultan B, Apvrille L (2024) AI-Driven Consistency of SysML Diagrams. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems. Association for Computing Machinery, New York, NY, USA, MODELS ’24, p 149–159, <https://doi.org/10.1145/3640310.3674079>, URL <https://doi.org/10.1145/3640310.3674079>
 - [30] Sultan B, Frénot L, Apvrille L, et al (2023) AMULET: a Mutation Language Enabling Automatic Enrichment of SysML Models. ACM Trans Embed Comput Syst URL <https://doi.org/10.1145/3624583>

- [31] Zhou D, Schärli N, Hou L, et al (2022) Least-to-most prompting enables complex reasoning in large language models. arXiv preprint arXiv:220510625