System Architects Are not Alone Anymore: Automatic System Modeling with AI

Ludovic Apvrille, Bastien Sultan

System Architects are not alone Anymore: Automatic System Modeling with AI

Ludovic Apvrille[1] and Bastien Sultan[1]
[1]*LTCI, Télécom Paris, IP Paris, 450 route des Chappes, Sophia-Antipolis, France*
{*ludovic.apvrille, bastien.sultan*}*@telecom-paris.fr*

Keywords: Modeling, Verification, Artificial intelligence, LLM, Analysis, Design

Abstract: System development cycles typically follow a V-cycle, where modelers first analyze a system specification before proposing its design. When utilizing SysML, this process predominantly involves transforming natural language (the system specification) into various structural and behavioral views employing SysML diagrams. With their proficiency in interpreting natural text and generating results in predetermined formats, Large Language Models (LLMs) could assist such development cycles. This paper introduces a framework where LLMs can be leveraged to automatically construct both structural and behavioral SysML diagrams from system specifications. Through multiple examples, the paper underscores the potential utility of LLMs in this context, highlighting the necessity for feeding these models with a well-defined knowledge base and an automated feedback loop for better outcomes.

1 INTRODUCTION

Artificial Intelligence (AI), specifically Large Language Models (LLMs) like OpenAI's GPT, surely have tremendous potential to revolutionize the realm of system design. The complex nature of designing efficient systems requires a deep understanding of the interplay between various components. However, the intricate details, coupled with the need for optimal performance, often pose significant challenges for designers. In this context, AI can play a pivotal role. LLMs, trained on extensive datasets, can generate valuable insights and recommendations, aiding designers in creating highly effective systems. By leveraging these AI models, designers can navigate the complexity of system design more effortlessly and make informed decisions that lead to enhanced systems.

In particular, with their ability to understand natural text, Large Language Models (LLMs) can often get the essence of system specification and output a view of them according to some aspects (e.g., requirements, design) in a selected format, for instance in SysML.

The research presented in this paper investigates how customized utilization of large language models (LLMs), like ChatGPT, can effectively aid system architects in developing preliminary designs. In this context, 'design' refers to the process of identifying system components, establishing interconnections between these components, and discerning each component's behavior. As the paper demonstrates, simply querying these LLM-based AI engines to extract components, interconnections, and behaviors from a system specification, or from existing diagrams, is not sufficient. Instead, we need to enhance the AI engine's context and implement a feedback loop. This loop helps identify errors in the AI's responses and subsequently refine the questioning strategy to provoke more accurate answers.

The paper first explains in Section 2 the context of the work: SysML, TTool, LLMs and ChatGPT. Then, Section 3 overviews the general framework of our contribution. Building on this, Section 4 shows how our framework can be successfully applied to automatically design systems with SysML blocks and state machines from system specifications. Following, Section 5 evaluates our contribution in the scope of several case studies. Finally Section 6 compares our approach with the state-of-the art, before concluding in Section 7.

2 CONTEXT

2.1 System Development with SysML

The SysML specification outlines the structure of a language, yet it doesn't provide a method detailing its application. Notwithstanding, numerous published methods exist that advise on how to leverage SysML efficiently for system development. One common approach involves initially capturing system requirements from the system specification. Subsequently, this specification undergoes analysis using a variety of diagrams, such as use case diagrams, sequence diagrams, and activity diagrams.

Use case diagrams primarily aim to encapsulate the central functions of the system and their interactions with the surrounding environment (actors). Moreover, sequence diagrams and activity diagrams concentrate on analyzing the behavior of the system to be designed. Eventually, a design typically uses blocks to decompose the system into its constituent components, and relies on state machines for the behavior of block.

The contribution of the paper enables TTool to identify all the diagrams mentioned above. However, the paper's focus is specifically centered on the design phase.

2.2 TTool

TTool[1] is a free and open-source toolkit for the edition of UML and SysML diagrams, as well as their subsequent verification. The verification aspect covers safety, cyber-security, and performance evaluation. This process is achieved via the utilization of internal tools, such as model-checkers and simulators, as well as external tools, including ProVerif, which is used for security evaluation.

Our choice to rely on TTool stems from its excellent extensibility and its operability through a command-line interface. These attributes simplify the connection evaluation to an AI engine.

2.3 LLM

Large Language Models (LLMs), such as OpenAI's GPT (OpenAI, 2023), draw upon techniques from deep learning and neural networks. They are a new shift in the field of natural language processing with better understanding and generation capabilities. But their utility goes beyond merely processing natural text because they can understand both the syntax and semantics of many different languages (including computer code) and they can generate their answer in various formats of data, including models and code.

A key feature developed in the next section is the capacity of ChatGPT to incorporate custom knowledge as input (also known as "user-defined contextual embeddings"). This empowers users to tailor various aspects such as the application domain, as well as the format, constraints, and other aspects of the responses. As shown, this feature forms the foundation for the contribution presented in our work.

Another significant advantage of ChatGPT lies in its API design. With just a simple HTTP request, one can interact with the ChatGPT AI engine. For example, the 'curl' command-line utility can be used to make queries to ChatGPT directly from a terminal. However, this isn't limited to terminal usage. Any programming language capable of issuing HTTP requests can interact with the ChatGPT AI engine. Thus, a standard query encompasses several key elements: the target URL, authorization (which requires an API key obtained from OpenAI), the specific language model you intend to use (in this case: "gpt-3.5-turbo"), and the user's question presented in a JSON format (for instance, "Hello!").

```
$ curl https://api.openai.com/v1/chat/
completions -H "Content -Type:
application/json"
-H "Authorization: Bearer OPENAI_API_KEY"
-d '{ "model": "gpt-3.5-turbo",
"messages": [{"role": "user",
"content": "Hello!"}] }'
```

The answer to the HTTP request is in JSON format, with an id of the chat, the answer given by the assistant ("Hello! How can I assist you today?") and finally the tokens used by the query and by the answer. The billing is based on both kinds of tokens.

```
{ "id": "chatcmpl
-7YzJ6Gmh0VjbWl3plUxqwzDrtDnnk",
"object": "chat.completion",
"created": 1688573172,
"model": "gpt-3.5-turbo-16k-0613",
"choices": [ { "index": 0, "message":
{ "role": "assistant",
"content": "Hello! How can I assist you
today?" }, "finish_reason": "stop" } ], \
"usage": { "prompt_tokens": 8,
"completion_tokens": 9,
"total_tokens": 17 } }
```

[1] https://ttool.telecom-paris.fr

3 CONTRIBUTION OVERVIEW

The principal contribution of this paper is a generic framework designed to support system architects throughout the product development cycle. The forthcoming section initially presents an overview of this framework, subsequently providing a thorough exploration of its primary elements.

3.1 Our framework: *TTool-AI*

An overview of our framework, called *TTool-AI*, is given in Figure 1.

Since the main idea is to automate the creation of SysML diagrams from a system specification, the elements of the top of Figure 1 highlight the necessary inputs: a system specification, and a question type (e.g., "identify the requirements from the following system specification"). The latter is crucial for guiding the AI in determining the specific type of diagram expected in the output. Depending on the choice of AI engine, certain proprietary system engineering knowledge might be required. This knowledge encompasses the format of diagrams (for instance, the output is expected in SysML V2 textual format), their semantics (e.g., a transition between states in a state machine is executable only if its guard condition evaluates to true), constraints (for instance, two blocks cannot share the same name), and overarching principles (e.g., too many or too few requirements). Notably, pre-existing modeled diagrams could also serve as input to the AI, aiding it in enhancing the resultant diagram and maintaining coherence between the expected new diagram and the pre-existing ones.

Upon consolidating the aforementioned information, the resulting combination is anticipated to be input into an AI engine, such as ChatGPT. The engine then generates an appropriate response in the form of a textual output adhering to the specified format (e.g., JSON, XML, SysML V2 textual format, etc.).

Given the inherent randomness of Large Language Model AI engines in answer selection, coupled with their existing limitations regarding answer relevance, our framework incorporates an automated feedback loop. The key objective of this feature is to scrutinize the AI's response in order to assess its quality. If this quality is determined as too low, the feedback loop is triggered, introducing a revised input text to guide the AI towards a more precise and effective answer. This provision for iterative improvement enhances the robustness and reliability of our framework.

Upon generating a satisfactory response, or after reaching a predefined limit of automated feedback iterations, the framework offers the designer the opportunity to seek additional refinements from the AI engine. Once the designer is pleased with the quality of the AI-generated output, he/she can instruct the system to render the resultant diagram automatically in TTool, demonstrating the seamless integration with SysML diagramming tools.

The section now delves into each main stage of *TTool-AI*.

3.2 Knowledge

As explained before, the idea of knowledge is to customize answers provided by the AI engine. The following example emphasizes on the interest of knowledge. Let us consider the following JSON message sent to ChatGPT:

```
"messages": [{"role": "user", "content":
"Is Nice sunny today?"}]
```

A typical answer provided by ChatGPT would be:

```
"I m sorry, as an AI language model,
I dont have access to current weather
conditions. However, Nice is located
in the French Riviera, which has a
generally sunny and Mediterranean
climate..."
```

Let us now inject some knowledge, adding a new *user* role as well as the answer expected to be provided by the AI engine. For this, we use the *assistant* role:

```
{"messages":[{"role":"user","content":
"Today, in Nice,
the sun shines with no clouds."},
{"role":"assistant","content":"ok"},
{"role":"user","content":
"Is Nice sunny today?"}]
```

```
"Yes, you mentioned earlier that the
sun is shining with no clouds in Nice.
So, it is indeed sunny today in Nice.
Enjoy the beautiful weather!"
```

The same approach can be used for system analysis design. First, we can specify a general domain for answers using the *system* role:

```
"messages":[{"role":"system",
"content":"You are a helpful
assistant for system engineering."},
```

However, this is far from sufficient: the AI engine remains uninformed about the specific diagram we expect, its required format, and the constraints associated with this diagram. Essentially, our strategy involves defining the expected diagram and its format first, followed by stipulating the constraints that the diagram must meet. The subsequent section provides

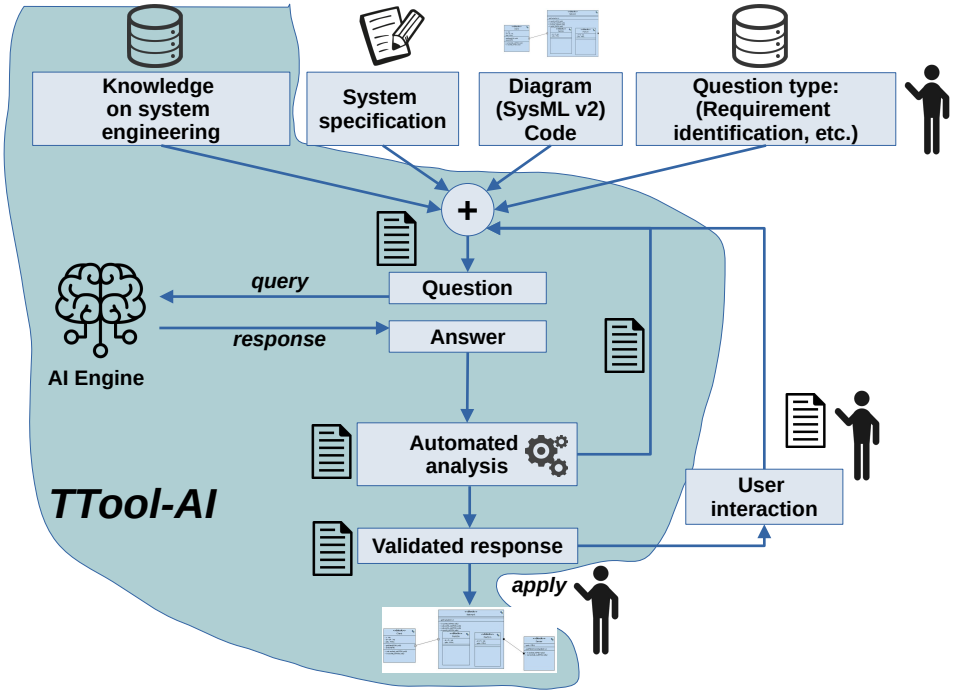

Figure 1: Overview of our framework.

tangible examples for both block and state machine diagrams. Yet, in a generic form, the knowledge typically looks like this:

```
"messages":[{"role":"system", "content":
"You are a helpful assistant for system
engineering."},

{"role":"user",
"content":"When you are asked to
identify ..., use the following JSON
format: {element: [{ "name": "Name of
element""}] ...
Also, respect the following contraints:
- 1. do not use ...
- 2. do prefer to list ...
etc.
```

Splitting the knowledge in different JSON messages may help if the knowledge becomes too large.

3.3 Automated feedback loop

The automated feedback loops checks two points:

1. The answer respects the expected output format (e.g., JSON).
2. All constraints provided in the knowledge are satisfied.

Algorithm 1 outlines the operation of the automated analysis. Essentially, it constructs the subsequent question q to pose to the AI engine by formulating a sub-question for each identified error in the current answer. If q remains empty after executing Algorithm 1, the feedback loop ends, allowing the user to take control of the response. Otherwise, the prior question and answer are incorporated into the knowledge base, and the new question q is submitted to the AI engine.

Data: Answer a provided by AI engine
Result: String q (question)
$q = \emptyset$
$n = numberOfErrorsIn(a)$
if $n! = 0$ **then**
  if $format(a) = invalid$ **then**
    append(q, "Invalid format at line ...")
  foreach *constraint* **do**
    if $(error = constraint(a,c))! = null$ **then**
      append(q, "constraint c is not satified: " + error)
end

Algorithm 1: Feedback loop.

3.4 User is also in the Loop

The user becomes involved in the process once all errors in the preceding response have been rectified, or

when the feedback loop limit has been met.

In the first scenario, the user must review any additional constraints that are not mathematically expressible, such as informal quality parameters of the answer. For example, the block names might not align with the user's expectations. To address these issues, the user can take on the role of the feedback loop by posing new queries to the AI engine, with each prior question and answer continuously added to the knowledge base.

The second scenario poses more of a challenge, for instance, if the AI engine consistently provides responses in an incorrect format or does not satisfy the constraints. The user has the choice to either provide its own knowledge to better guide the AI engine to produce the correct response, or to make manual corrections to all these errors, or finally to amend the initial system specification or diagrams to typically impose additional constraints.

Once the user is satisfied with the generated response, *TTool-AI*can draw the corresponding diagram from the AI answer's JSON representation.

4 SYSTEM DESIGN

This section shows how *TTool-AI*, presented in previous section, can be used to design a system from its specification. By system design, we mean a set of interconnected SysML blocks, with blocks' behavior described with state machine diagrams.

4.1 Case study

We first introduce a simple case study—a coffee machine—that will be used all along this section to illustrate the creation of a SysML design from a system specification. The evaluation of our approach with more case studies is done in the next section.

The specification of this case study is: *This coffee machine dispenses a beverage only after two coins have been deposited. If there's a substantial delay between the insertion of the first and second coins, the machine returns the initial coin. Likewise, if a beverage isn't selected promptly after the deposit of the two coins, both coins are automatically ejected. If either of the beverage buttons (tea or coffee) is pressed before the coins are ejected, the machine begins to prepare the selected drink. Notably, it takes 10 seconds to brew a coffee and 8 seconds to make a tea. Once the beverage has been collected, the machine is ready to accept new coins for the next order.*.

4.2 Structural Aspects

The structural representation involves outlining the blocks, encompassing their attributes, methods, ports, and interfaces (as seen in SysML Block Definition Diagrams), and demonstrating how these blocks can be instantiated and interconnected (as depicted in SysML Internal Block Diagrams).

Given the considerable amount of knowledge required to describe these two diagrams and their constraints, we've divided it into two subprocesses: one for identifying blocks, block instances, and block attributes, and another for recognizing signals (i.e., interfaces), ports, and connections between ports. This division required us to modify the framework depicted in Figure 1. We now need to introduce an initial set of knowledge, engage in a feedback loop focusing solely on this initial knowledge, and then introduce the second set of knowledge once the automated feedback loop is complete, as illustrated in Figure 2. The upcoming section delves further into this approach using the running example.

4.2.1 Knowledge and question for Question #1

Our methodology for identifying blocks is predicated on the specification of the enforced JSON format and constraints concerning blocks. The user is only responsible for supplying the system specification (in our case study: the one of the coffee machine) and for selecting the question type (here, identify system blocks).

The knowledge injected for Question #1 is defined as follows:

```
When you are asked to identify SysML
blocks, return them as a JSON
specification formatted as follows:
{blocks: [{"name": "Name of
block", "attributes": ["name":
"name of attribute", "type": "int or
bool" ...} ...]}
```

The typical constraints added to the knowledge are:

```
# Respect: each attribute must be of type
"int" or "bool" only
# Respect: Any identifier (block,
attribute, etc.) must no contain any
space. Use "_" instead.
...
```

Finally, Question #1 which is asked by *TTool-AI*to the AI engine is, with the system specification concatenated at the end of it:

```
From the following system specification,
using the specified JSON format, identify
```

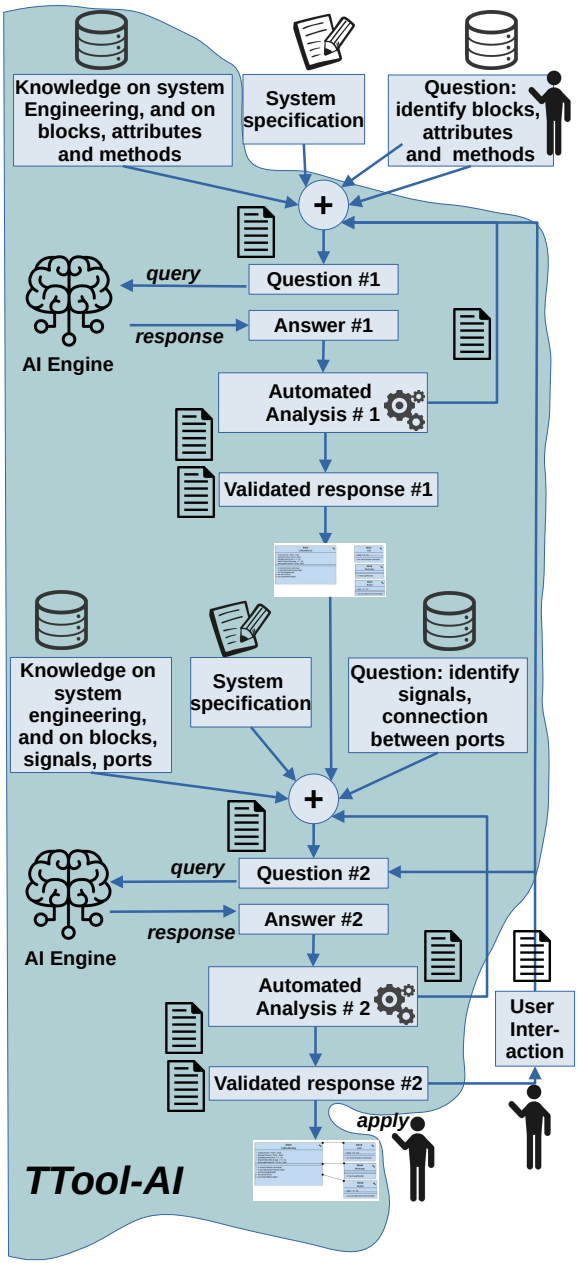

Figure 2: Application of our framework to the identification of blocks and their connections.

```
the typical system blocks and their
attributes. Do respect the JSON format,
and provide only JSON (no explanation
before or after).
```

4.2.2 Knowledge and question for Question #2

In the process of identifying signals and establishing connections between blocks, the JSON format remains to be used for the output is quite straightforward. However, it is crucial to respect many syntactic and semantical constraints. For example, only signals with corresponding attributes can be linked. Furthermore, in a connection, one signal must be classified as an output, while the other is an input signal. Since our diagram is a closed system, a signal must be connected exactly to one other signal, among other considerations.

```
...
#Respect: Two signals with the same name are
assumed to be connected: this is the only
way to connect signals.
#Respect: Two connected signals must have
the same list of attributes, even if they
are defined in two different blocks.
One of them must be output, the other one
must be input.
#Respect: all input signals must have
exactly one corresponding output signal,
i.e., an output signal with the same name
...
```

A noteworthy aspect depicted in Figure 2 is that the input for Question #2 is the result derived from Question #1. To accomplish this, we ask to update the JSON of Answer #1 with signals and connections. Given that the general framework in our system automatically incorporates the previous round's answer into the knowledge between two questions, we merely need to require the update of the preceding JSON specification. Finally, Question #2 can be sketched as follows:

```
From the previous JSON and system
specification, update this JSON with
the signals you have to identify. If
necessary, you can add new blocks and new
attributes. Connect the signals accordingly
to constraints to be respected.
```

4.2.3 Example

Let us apply all this to our running example. We set the maximum number of successive feedback loop iterations to 20. We do not specify how many blocks we expect. In this toy example, no feedback was necessary, i.e., the two questions were answered correctly with regards to the expected output format (JSON) and with respect to the constraints (e.g., connection between signals). The obtained diagram is given in Figure 3. Blocks, attributes, signals and connections are correctly defined, as a (good) engineer would do. The choice to use 5 blocks for this system is the one of ChatGPT: our solution features four blocks.

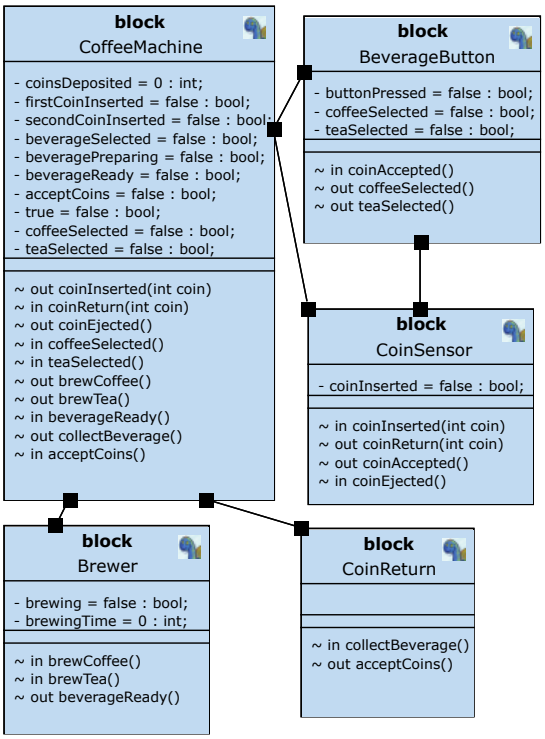

Figure 3: Block diagram obtained for the running example.

4.3 Behavioral aspects

4.3.1 Identification of State Machines

Unlike the identification of blocks that needs two successive questions, the identification of the behavior can be done with only one question, thus following the generic framework of Figure 1. But contrary to previous questions, the user must first select an internal block diagram that the question takes as input. Also, the system specification must preferably be provided since a definition of blocks is generally not enough to deduce the correct behaviour of blocks. Finally, the complexity for the AI mostly lies in the fact that the semantics of state machines, while being quite easy to understand for humans, is quite complex to explain as a knowledge. So, this is frequent that the feedback loops must operate several times, sometimes many times, before a syntactically and semantically correct state machine can be obtained.

Finally, typical constraints that we specify are:

```
...
# Respect: in actions, use only attributes
and  signals already defined in the
corresponding block
# Respect: at least one state must be
called "Start", which is the start state
# Respect: if a guard, an action, or an
after is empty, use an empty string "",
do not use "null\"
# Respect: an action contains either a
variable affectation, e.g. "x = x + 1"
or a signal send/receive
...
```

The question *TTool-AI*asks for each block of the design is the following:

```
From the  system specification, and from
the definition of blocks and their
connections, identify the state machine
of block: ...
```

4.3.2 Example

We now proceed to apply the process of identifying state machines to the block shown in Figure 3. We've set a limit of 10 iterations for the feedback loop for each state machine. Because the block diagram has two blocks, we have two states machines to identify, thereby necessitating an execution of the identification question for each block (this is automated).

The resulting state machine diagram for the *CoffeeMachine* block is depicted in Figure 4. We didn't make any alterations to the states or transitions generated by the AI engine, and we haven't incorporated any user feedback. However, even after 10 iterations, there is still useless guard (with "true"), and one signal is used with the incorrect number of attributes. These two issues could easily be handled by hand.

Figure 5 presents a screenshot of the AI window in TTool. Notably, the selected question ("Identify state machines") is visible at the top. The left text area displays the user input (here, the specification), while the AI's response can be seen in the right text area, highlighted in red. Furthermore, the feedback loop of TTool reports errors in blue in the right text area: the screenshot reports 4 errors to the AI engine.

5 IMPLEMENTATION AND EVALUATION

5.1 *TTool-AI*: Implementation

Integrating the *TTool-AI* process from Figure 1 into TTool presents two primary challenges. Firstly, we must select the most pertinent knowledge to automatically transmit before user interactions, tailoring GPT-3.5 to suit our particular requirements. Secondly, we need to handle GPT's responses in a way that (i) allows easy corrections of any JSON content within the automated feedback loop, and (ii) ensures a smooth integration into TTool.

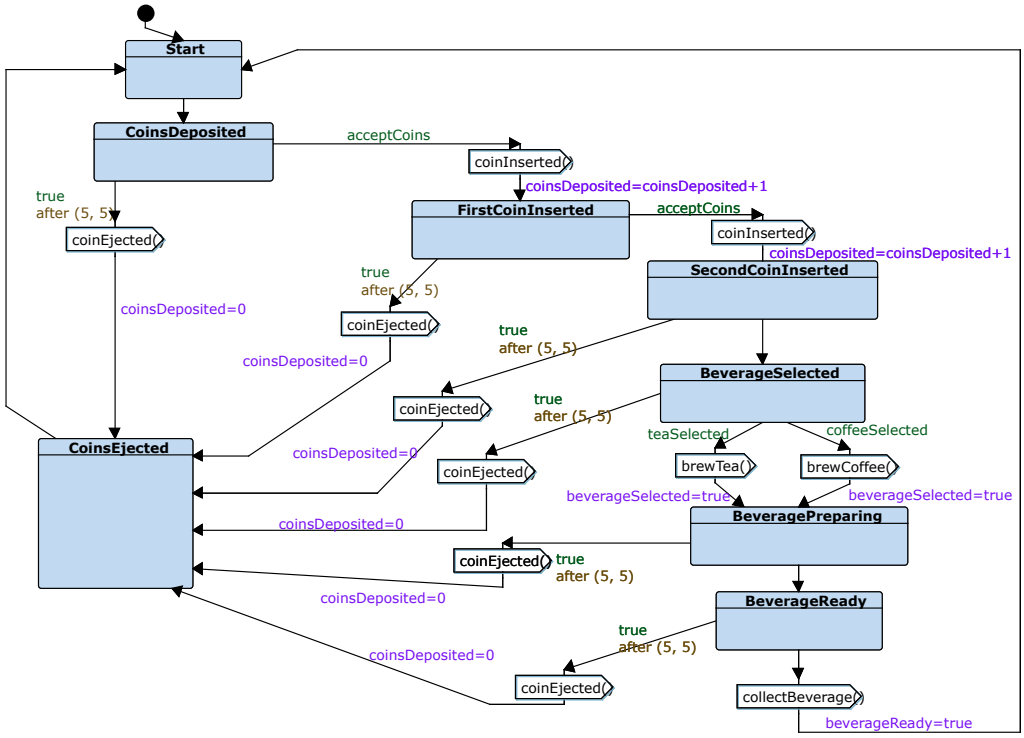

Figure 4: State machine diagram of block CoffeeMachine obtained for the running example. signal sending and receiving actions are depicted using a custom operator for a better emphasis on data exchange

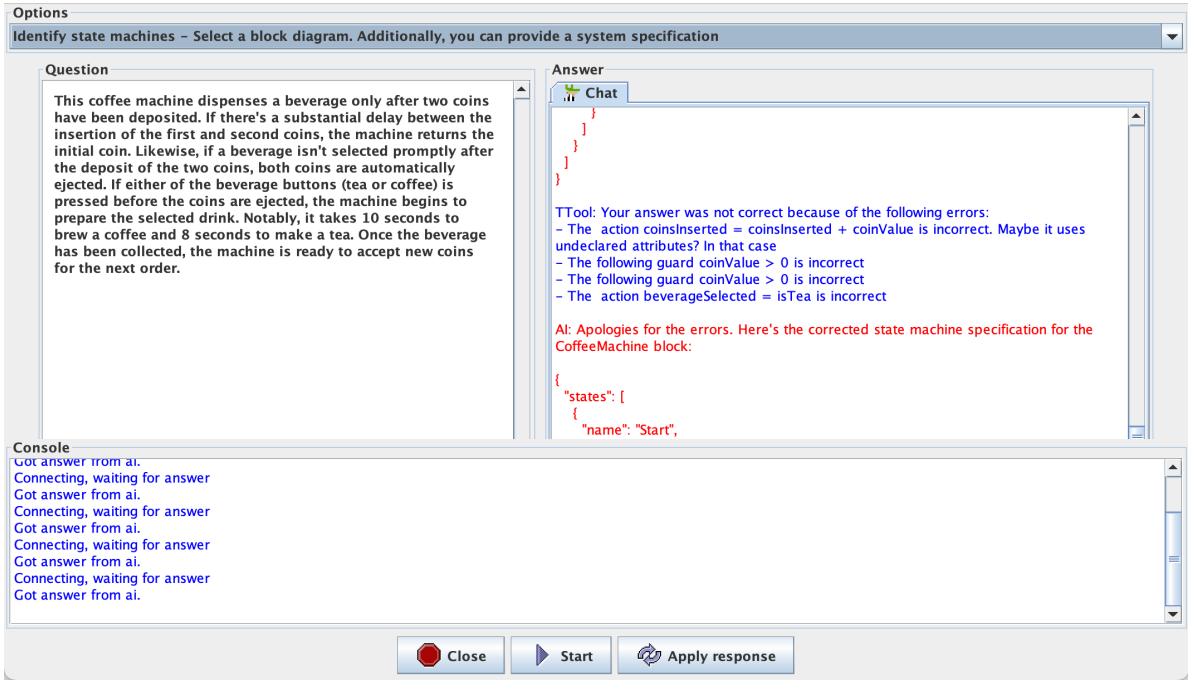

Figure 5: Screen capture of TTool when identifying state machines.

5.1.1 Designing the most effective preliminary knowledge

Preliminary remark: our implementation leverages the capability of OpenAI's API to inject knowl-

edge in GPT chats as a couple {input, expected answer}. Thus in this section a *knowledge instance* always refers to a couple of Strings that is sent by *TTool-AI* prior to the specification injection. In other terms, these knowledge instances solely encompasses *knowledge on system engineering (and on blocks, attributes and methods)* depicted in Figures 1 and 2.

Regarding knowledge injection, GPT-3.5 has two constraints. Firstly, the *context* within a chat is restricted to a fixed number of tokens (16,000 in GPT-3.5 *turbo*, equivalent to approximately 12,000 words). This context encompasses the entire conversational history, including questions and answers exchanged in the chat. Consequently, the knowledge injection is subject to these limitations. In addition, the financial cost of sending data to GPT through its API is proportional to the number of tokens handled by the LLM. Secondly, during our interactions with GPT, we observed that it often fails to effectively process lengthy knowledge instances, often facing challenges in assimilating all the information contained within such instances. Therefore, the knowledge instances shall be as concise as possible.

The knowledge instances we inject thus share the following characteristics:

1. They are limited to 200 words each. If necessary, we divide the injected knowledge into several knowledge instances to ensure GPT can process them without losing information.
2. As shown in the examples in Section 4, they consist solely of the following elements: (i) the expected JSON format and (ii) a set of constraints elucidating the main syntactic features of SysML profiles utilized in TTool.

It's worth noting that whenever an incorrect model is provided in a response, *TTool-AI* resets the chat history and reintroduces only the relevant knowledge instances to solve the errors. This helps maintain a concise context and prevents potential future errors, ensuring that GPT doesn't capitalize on the wrong model.

5.1.2 Handling GPT's responses

The process of handling GPT's responses primarily involves two key stages: firstly, a syntax analysis of the JSON content they contain, and secondly, either generating a new SysML model or modifying an existing one based on the generated JSON.

In practical terms, when *TTool-AI* receives a response from GPT, it initially attempts to extract a JSON structure from the response. If this extraction process fails, as detailed in Algorithm 1, *TTool-AI* logs any parsing errors that occurred and appends them to the feedback question to be sent. If the JSON is successfully extracted, *TTool-AI* proceeds to construct a SysML model from it. In this stage, if any syntax errors are detected, *TTool-AI* includes them in the feedback question to be sent. The feedback question is subsequently sent to GPT and this syntax analysis loop continues until one of the following conditions is met: (1) *TTool-AI* receives a correct response (i.e., without JSON or syntax errors), or (2) n consecutive feedback questions have been sent (default in TTool: $n = 20$ or 10). This n prevents excessively lengthy feedback loops. It's worth noting that this automated feedback loop has allowed us to streamline the set of syntactic constraints included in the knowledge instances, since GPT is generally capable of rectifying its responses effectively through the iterative feedback questions.

Once the loop has terminated, if the user chooses to apply the response as depicted in Figures 1 and 2, then the SysML model built from GPT's response is drawn.

5.2 Testing environment

Tests were conducted using the most recent version of TTool (nightly build, October 2023) paired with the ChatGPT 3.5 *turbo* model. All tested systems are available in a public, anonymous GitHub repository at https://github.com/zebradile/ttool-ai. Within this repository, the directories named *platooning*, *spacebasedsystem*, and *AutomatedBraking* contain both the system specifications (in *desc* files) and the models generated by TTool-AI (in *.xml* files). These system specifications were sourced from use-case specifications of European projects. A *README* file at the repository's root offers instructions for replicating the results. Additionally, a *result.ods* file details computation times, grading out of 100, and an overview of the results which is also visualized in Table 1.

These system specifications were provided to approximately 15 master-level students after 21 hours of instruction. These students were allotted 1.5 hours to design and draw the state machines. They had the opportunity to practice with at least three different specifications prior to the assessment. It's noteworthy to mention that 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, among others, the adequacy of the diagrams to the specification (does the proposed architecture meet the specification? Is the behavior of the state machines, observed through the TTool simulator, in line with this specification?), the quantity of exchanges between blocks, diagrams

readability, number of blocks and states and the consistency of their naming, and the absence of attributes declared in blocks but not utilized in state-machine diagrams. They also include the syntactic correctness of the models (i.e., the number of errors and warnings detected by TTool's syntax checker).

5.3 Results and discussion

A summary of the results can be found in Table 1. In the table, BD stands for Block Diagram and SMD for State Machine Diagram. In general, *TTool-AI*slightly outperforms the students, both in block diagrams and state machine diagrams. Analogous to the students' performance, TTool-AI excels more at discerning the system's structure than identifying its behavior in the context of state machines. The grading consistency for TTool-AI is also notable: it has a standard deviation of 15 points, whereas students exhibit a deviation near 30 points.

Delving into detailed results (as shown in *results.ods*), it's evident that *TTool-AI*adeptly manages both the platooning and space-based systems. However, with the automated braking system, which boasts a more lengthy, intricate, and ambiguously-written specification, students have a slight edge over TTool-AI for state machines (but not for block diagrams).

Does this mean that engineers are being overshadowed? Fortunately, the answer is no. *TTool-AI*excels as a tool, laying out a system's structure and producing initial state machine diagrams swiftly and with commendable accuracy. Its efficiency does wane when confronted with intricate systems, ironically where its efficiency would be most desired. However, it's essential to note that for this assessment, the human interaction aspect in *TTool-AI*was disabled. We believe that if students had paired their efforts with TTool-AI within the 1.5-hour timeframe, they would've likely achieved superior grades. Similarly, we anticipate engineers to benefit immensely: harnessing *TTool-AI*for initial, time-intensive architecture and state machine designs, and subsequently refining these preliminary drafts, whether manually or in tandem with the AI.

6 RELATED WORK

The automatic generation of (formal) models from system specifications has been a persistent research challenge. As elucidated in the comprehensive literature review contained in (Landhäußer et al., 2014), this area of study has been active since the late 1990s. However, the process of model generation often requires imposing constraints on the syntax of input requirements or necessitates manual preprocessing, as exemplified in the work by Gelhausen et al. (Gelhausen and Tichy, 2007). Recent advancements, such as the ARSENAL framework (Ghosh et al., 2016), have introduced model generation approaches that minimize restrictions on the input language. Nonetheless, even with these powerful tools, certain natural language expressions can still pose challenges, eluding their automated transformation into formal models. We are of the opinion that the recent advancements in the practical applicability of generative AI models, such as GPT, present an opportunity for handling system specifications written in totally-free natural language. Leveraging these AI models, as emphasized in the preceding sections, helps reducing the research effort on language processing but directs it toward tailoring the model to suit the requirements of the modeling process.

More broadly, the subject of modeling assistants is not a recent development in research and engineering. In a comprehensive survey conducted by Savary-Leblanc et al. (Savary-Leblanc et al., 2023), which encompassed papers published between 2010 and 2022, the authors identified 11 notable papers introducing tools aimed at aiding engineers in the process of model design. Among these papers, four specifically concentrated on UML models, with one of them addressing SysML models, introducing a tool that offers support for the design of use-case diagrams (Aquino et al., 2020). Furthermore, recent research has explored the development of AI-based Model-Based Systems Engineering (MBSE) assistants within the context of the growing trend of AI-based methods and tools. In this context, Chami et al. (Chami et al., 2019) introduced a framework grounded in natural language processing (NLP) that autonomously generates SysML use-case and block diagrams from textual requirements inputs. Furthermore, Schräder et al. (Schräder et al., 2022) introduced three AI-based MBSE assistants, each serving distinct purposes: a workshop assistant capable of converting hand-drawn sketches into formal SysML models, a knowledge-based assistant offering design suggestions based on training data derived from a set of models, and a chatbot designed to process natural language queries related to modeling and provide responses in a natural language format.

However, with the emergence of the use of GPT 3.5 in 2022, chatbots, particularly those harnessing the capabilities of large-language models (LLM), have demonstrated their potential beyond their traditional role of handling basic question-and-response

Table 1: Evaluation results: time to produce block and state machine diagrams, and related grade

	Time BD (s)	Grade BD (/100)	Time SMD (s)	Grade SMD (/100)
Average	40	81	178	63
Std dev.	10	16	97	15

(a) TTool + AI

	Time BD (s)	Grade BD (/100)	Time SMD (s)	Grade SMD (/100)
Average	2700	70	2700	58
Std dev.	—	26	—	32

(b) Students

interactions. Due to their versatility, LLMs have been adapted to address a large variety of challenges and MBSE assistance is no exception. Indeed, these chatbots can now generate responses formatted to cater specifically to the requirements of model designers, such as producing UML diagrams with a remarkably low rate of syntax errors (Cámara et al., 2023). In this study, the authors assessed ChatGPT's model generation capabilities by tasking it with producing UML class diagrams based on specifications provided in natural language. An examination of the algorithm's responses to 40 distinct modeling exercises led the authors to the observation that ChatGPT frequently succeeded in generating syntactically correct models. However, it was noted that the semantic accuracy of these models (particularly concerning the relationships between classes) was not consistently achieved. Given these imperfections, achieving an accurate model necessitates a series of iterative inquiries to refine and enhance the output. Consequently, the authors concluded that the effort required by the user is still important. We believe that our contribution plays a role in tackling this issue, thanks to the automated feedback loop our process involves, which results in significant time savings. In a more architecture process-oriented study, Ahmad et al. (Ahmad et al., 2023) present a comprehensive report detailing their use of ChatGPT for software architecture tasks, which include generating requirements, UML models, and evaluating the proposed architecture. Their experiment highlights the utility of ChatGPT as an assistant for software architects, but also raises several concerns about its responses, including response variability and ethics/intellectual property issues. In both cases, human analysis and, if necessary, iterative questioning are essential to converge towards a correct system architecture. At an earlier stage in the design cycle, LLM have also been evaluated on the generation of goal-models (Chen et al., 2023; Nakagawa and Honiden, 2023), yielding promising results when used judiciously (e.g., through the incorporation of feedback and/or running multiple prompts).

To the best of our knowledge, our contribution is the first to report the direct integration of ChatGPT into an MBSE toolkit, incorporating automatic diagram generation from the LLM's responses and automating the feedback loop.

7 CONCLUSION

This paper introduces a novel environment designed to aid engineers in their modeling endeavors. To achieve this, we have integrated a tool, TTool, with a chatbot, ChatGPT. Two primary contributions stand out: the knowledge we embed within the AI system and the feedback loop that refines and steers the AI toward a more accurate model. Our evaluation, with various systems, underscores the significance of our approach.

However, applying this to full-scale problems remains a challenge. The constraints on the AI's knowledge capacity, paired with the degradation in model quality when overloading it with information, need addressing. Traditional methods such as hierarchical modeling, incremental modeling, and refinement processes offer potential solutions. We expect to make further contributions in this realm in the upcoming months.

REFERENCES

Ahmad, A., Waseem, M., Liang, P., Fahmideh, M., Aktar, M. S., and Mikkonen, T. (2023). Towards human-bot collaborative software architecting with chatgpt. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 279–285.

Aquino, E. R., De Saqui-Sannes, P., and Vingerhoeds, R. A. (2020). A methodological assistant for use case diagrams. In *8th MODELSWARD: International Conference on Model-Driven Engineering and Software Development*, pages 1–11.

Cámara, J., Troya, J., Burgueño, L., and Vallecillo, A. (2023). On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml. *Software and Systems Modeling*, pages 1–13.

Chami, M., Zoghbi, C., and Bruel, J.-M. (2019). A first step towards ai for mbse: Generating a part of sysml models from text using ai. *A First Step towards AI*.

Chen, B., Chen, K., Hassani, S., Yang, Y., Amyot, D., Lessard, L., Mussbacher, G., Sabetzadeh, M., and Varró, D. (2023). On the use of gpt-4 for creating goal models: An exploratory study. In *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, pages 262–271. IEEE.

Gelhausen, T. and Tichy, W. F. (2007). Thematic role based generation of uml models from real world requirements. In *International Conference on Semantic Computing (ICSC 2007)*, pages 282–289.

Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., and Steiner, W. (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*, pages 41–46. Springer.

Landhäußer, M., Körner, S. J., and Tichy, W. F. (2014). From requirements to uml models and back: how automatic processing of text can support requirements engineering. *Software Quality Journal*, 22:121–149.

Nakagawa, H. and Honiden, S. (2023). Mape-k loop-based goal model generation using generative ai. In *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, pages 247–251. IEEE.

OpenAI (2023). GPT4 Technical Report. Technical report.

Savary-Leblanc, M., Burgueño, L., Cabot, J., Le Pallec, X., and Gérard, S. (2023). Software assistants in software engineering: A systematic mapping study. *Software: Practice and Experience*, 53(3):856–892.

Schräder, E., Bernijazov, R., Foullois, M., Hillebrand, M., Kaiser, L., and Dumitrescu, R. (2022). Examples of ai-based assistance systems in context of model-based systems engineering. In *2022 IEEE International Symposium on Systems Engineering (ISSE)*, pages 1–8. IEEE.