AI-Driven Consistency of SysML Diagrams

Bastien Sultan
bastien.sultan@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Sophia-Antipolis, France

Ludovic Apvrille
ludovic.apvrille@telecom-paris.fr
LTCI, Télécom Paris, Institut Polytechnique de Paris
Sophia-Antipolis, France

ABSTRACT

Graphical modeling languages, expected to simplify systems analysis and design, present a challenge in maintaining consistency across their varied views. Traditional rule-based methods for ensuring consistency in languages like UML often fall short in addressing complex semantic dimensions. Moreover, the integration of Large Language Models (LLMs) into Model Driven Engineering (MDE) introduces additional consistency challenges, as LLM's limited output contexts requires the integration of responses. This paper presents a new framework that automates the detection and correction of inconsistencies across different views, leveraging formally defined rules and incorporating OpenAI's GPT, as implemented in TTool. Focusing on the consistency between use case and block diagrams, the framework is evaluated through its application to three case studies, highlighting its potential to significantly enhance consistency management in graphical modeling.

CCS CONCEPTS

• **Computing methodologies** → **Model development and analysis**; **Artificial intelligence**; • **Software and its engineering** → **System modeling languages**; **Unified Modeling Language (UML)**.

ACM Reference Format:
Bastien Sultan and Ludovic Apvrille. 2024. AI-Driven Consistency of SysML Diagrams. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3640310.3674079

1 INTRODUCTION

In the context of MDE methods, system design typically involves the creation of various models, each providing a distinct perspective of the system. For example, UML and SysML employ several types of diagrams, including use case diagrams (UCDs) for describing the high-level functional architecture and interactions with the system's environment, class/block diagrams for detailing the system's logical architecture, and state-machine diagrams (SMDs) for modeling algorithmic behavior aspects. Ensuring consistency between those diagrams is challenging and time consuming, while being critical to ensure system correctness. Traditionally, consistency has been managed through the definition and enforcement of sets of rules [14, 27, 28]. These approaches, while capturing critical semantic relationships, may leave more complex semantic gaps unbridged between the models. Recent contributions in the domain of Large Language Models (LLMs) suggest a promising way for addressing these more advanced semantic gaps more effectively [9, 17].

Furthermore, il will become increasingly common to generate different system views from the same system specification using LLMs [3, 6]. New challenges then arise, due to the inherent limitations of these models, such as their maximum token output (for instance, capped at 4,096 tokens for OpenAI's GPT-3.5 and 4). This limitation induces the generation of diagrams through multiple queries and answers. Due to the stochastic nature of LLMs, this multi-step generation introduces the risk of inconsistencies not only between different diagrams but also within different parts of the same diagram, highlighting the need for adapted mechanisms to ensure diagram consistency [6].

This paper introduces a new approach to (1) use LLMs to ensure coherence between SysML diagrams and (2) ensure the internal consistency of LLM-generated diagrams. To achieve this, we propose a novel framework that builds upon TToolAI [3], incorporating a feedback loop that leverages rules, LLMs, and, when required, human engineers to enforce consistency both across diagrams and within diagrams. Our general consistency approach applies to any UML or SysML diagram, yet the paper and our current implementation focuses on the internal consistency of UCDs and block diagrams (BDs), and on the coherence between UCDs (used for system analysis) and BDs (used for system design).

The rest of the paper is organized as follows. Section 2 gives an overview of the related works. Section 3 introduces formal definitions of SysML UCDs and BDs, a set of formal rules tailored to ensure internal consistency of LLM-generated UCDs and BDs and to ensure coherence[1] between these two classes of diagrams. Section 4 provides the description of our framework. Section 5 evaluates the contribution on three case-studies and provides a step-by-step description of the approach. Section 6 discusses our approach in light of its evaluation and Section 7 concludes the article.

2 RELATED WORKS

2.1 Enforcing UML/SysML Models Consistency

The design of complex systems always involves the development of various models employing diverse formalisms. Each model emphasizes a specific aspect of the system, such as software architecture through class diagrams, discrete behavior via automata/statecharts, continuous behavior through differential equations and state representations, or the physical architecture through plans. Establishing semantic correspondences among these heterogeneous views

[1] In this paper, we treat the terms *consistency* and *coherence* as synonymous.

MODELS'24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0504-5/24/09
https://doi.org/10.1145/3640310.3674079

presents a significant challenge in the field of complex systems engineering [10]. Even when narrowing the scope to a unified formalism like UML/SysML, this issue persists: UML indeed encompasses multiple views, each based on distinct concepts. Some of these concepts recur across different views: for instance, actors appear in both UCDs and sequence diagrams, while signals are found in block/class diagrams and statecharts. This recurrence provides a basis for reinforcing consistency across different views.

A substantial body of work focuses on ensuring the consistency of UML/SysML views, including maintaining cross-view consistency as well as internal consistency within each view. Some studies introduce new models aimed at deriving various views from a unified metamodel, such as [23], others rely on mapping models to formal ontologies to check for their internal consistency with a set of constraints—including OCL constraints—as demonstrated by [18], and most of them focus on the definition and application of *consistency rules*. Comprehensive literature reviews on these rule-based methodologies are available [15, 26, 27]: Torre et al. [27] identified 95 publications, starting from 2013, that address UML diagram consistency by proposing at least one consistency rule. As explained in [28], these publications propose approaches ranging from manual to automated consistency rule checks, including tool-assisted methods [20]. While these strategies are highly effective and relevant, they inherently only address inconsistencies defined by rules or predefined constraints, potentially leaving semantical inconsistencies unresolved. As evidence of this, to the best of our knowledge, some rules we propose in Section 3, especially those focusing on crossed-consistency between SysML UCDs and BDs, have not been covered in existing literature reviews.

Furthermore, existing rules may not adequately address the internal consistency of diagrams generated by LLMs: since they where mainly designed to ensure consistency of human-generated diagrams, they may not cover trivial aspects (e.g., ensuring a UCD is not empty) that however need to be enforced when generating models with LLM-based tools.

Finally, some semantic inconsistencies may be difficult to address by rules that can be checked algorithmically (e.g., if two synonymous names are used in different diagrams to designate the same modeled object—as Subsection 5.1 will illustrate), therefore calling for complementary approaches.

2.2 Integrating LLMs into MDE

In the past two years, the broad release of LLMs like OpenAI's GPT [1] has opened new promising solutions for enhancing engineering methodologies, particularly in modeling stages. The available literature reports the use of LLMs for generating diverse types of models with promising outcomes, including domain models [8], goal models [7, 21], business process models [11], and scenario-based models [12]. In direct link with this paper's very subject is the work by Ahmad et al. [2], which provides an insightful overview of ChatGPT's application in various software architecture tasks, including generating requirements, creating UML models, and evaluating these models. Similarly, Camara et al. [6] have conducted a detailed investigation into the use of GPT for generating UML class diagrams from natural language specifications. Their findings highlight that while GPT can efficiently produce syntactically correct models, these models often lack consistency and require iterative refinement to achieve semantic accuracy, demanding considerable effort from the users in the modeling process. TTool-AI [3] tackles this iterative refinement issue. This GPT-based framework, integrated with the MDE toolkit TTool, incorporates an automated feedback loop, significantly reducing the need for manual intervention in refining LLM outputs. Consequently, TTool-AI enables for the generation of SysML BDs and SMDs directly from natural language specifications through a single-click operation. However, the diagrams generated by TTool-AI may still require manual refinement for improving their consistency, and the framework does not yet support AI-based generation for all SysML diagrams. The work we present in this paper builds upon TTool-AI and takes an initial step towards addressing these two limitations and more broadly, towards helping TTool users ensuring the consistency of their models throughout the entire MDE process, as discussed below.

2.3 TTool

TTool[2] is an open-source MDE framework. It supports graphical and textual creation of SysML models using formally defined SysML profiles targeting real-time systems and design space exploration [16, 22], code generation, and formal verification through simulation and model-checking—the formal semantics of SysML profiles enable direct model-checking of SysML models [5] without the need for intermediate formalisms, minimizing the discrepancy between the model creator's intentions and the semantics being verified. TTool incorporates mechanisms to ensure the consistency of certain SysML views within its graphical interface. For example, in a block diagram, the user cannot connect two signals unless they were previously defined within the blocks. However, the scope of these mechanisms is limited and does not cover all diagram types. In particular, there are no mechanisms to enforce consistency between analysis diagrams—such as use-case and activity diagrams—and design diagrams—such as block diagrams. Our contribution enhances TTool by introducing a modeling assistant that takes a first step towards helping users in creating consistent diagrams from system analysis to system design.

2.4 AI-Driven UML/SysML Models Consistency

While LLMs have been widely used for model generation, to the best of our knowledge, no research has been published on their application in ensuring both internal consistency within modeling views and consistency across different views. In a related area, a recent bachelor's thesis explores the use of LLMs to maintain consistency between UML activity diagrams and the corresponding generated source code [4]. On connected research topics, LLMs have also been used to detect inconsistencies, whether between source code and its description in natural language [17], or for detecting grammatical inconsistencies [9].

Given that their functioning relies on detecting similarities between a text and a corpus of texts, and given their proficiency in understanding models and specifications—as demonstrated by the works cited above—LLMs might be effective tools for determining whether two models, and therefore two texts, accurately describe

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

the same object. Consequently, they should constitute an interesting complementary approach to rule-based methods for achieving cross-view consistency in UML/SysML diagrams.

3 THEORETICAL CONTRIBUTIONS

Our contribution integrates rule-based and LLM-based methodologies to guarantee both internal and cross-consistency among LLM-generated or manually-generated SysML diagrams. In this section, we outline the specific consistency rules incorporated into our framework. As in this paper we emphasize UCDs and BDs, we initiate our discussion with their formal definitions. These definitions then enable for the accurate establishment of our consistency rules.

3.1 Formal Semantics of SysML Use Case and Block Diagrams

Definition 1 (Alphabet and Names).

- $\mathcal{A} = \{a, A, b, B, \ldots, z, Z\} \cup \{\_\}$ *is the* alphabet.
- $\mathcal{A}*$ *is the set of all finite sequences over* $\mathcal{A}$, $*$ *being the Kleene star operator.*
- $\mathcal{A}*_V \subsetneq \mathcal{A}*$ *is the set of* verbs, *and* $\mathcal{A}*_N \subsetneq \mathcal{A}*$ *is the set of* nouns.
- $\forall a \in \mathcal{A}*$ *with a length exceeding* $i \in \mathbb{N}$, *we define* a_i *as the subsequence composed of its initial* i *characters.*

3.1.1 Use Case Diagrams.

Definition 2 (Basic Sets for Use Case Diagrams).

- $\text{Types}_\text{E} = \{include, extend, specialize, associate\}$ *is a set of* edge types.
- $\text{Types}_\text{V} = \{actor, usecase\}$ *is a set of* vertex types.

Definition 3 (Use Case Diagram).
A use case diagram *is a 4-uple* $\langle (V, E), type_V, name_V, type_E \rangle$ *where:*

- (V, E) *is a directed graph.*
- V *is a set of vertices.*
- $type_V : V \rightarrow \text{Types}_\text{V}$ *is a function that types vertices.*

Note: $V = V_A \sqcup V_U$[3], *where* $V_A = \{v \in V | type_V(v) = actor\}$ *and* $V_U = \{v \in V | type_V(v) = usecase\}$. *Each vertex of* V *has a name, and a vertex of* V_U *may have in addition an* extension point *name. We denote with* $V_{Uext} \subseteq V_U$ *the set of vertices that have an extension point, and* $V_{U\overline{ext}} = V_U \setminus V_{Uext}$.

- $name_V : \begin{cases} V_A \sqcup V_{U\overline{ext}} \rightarrow \mathcal{A}* \\ V_{Uext} \rightarrow \mathcal{A}*^2 \end{cases}$ *is a total function that names vertices (and their possible extension point). It is such that* $\forall v \in V_A, \exists i \in \mathbb{N}$ *s.t.* $name_V(v)_i \in \mathcal{A}*_N$, $\forall v \in V_{Uext}, \exists i \in \mathbb{N}$ *s.t.* $name_V(v)_i \in \mathcal{A}*_V$ *and* $\forall v \in V_{U\overline{ext}}, \exists (i, j) \in \mathbb{N}^2$ *s.t., if we denote* $name_V(v)$ *with* $(name_V(v)_1, name_V(v)_2)$, $name_V(v)_{1i} \in \mathcal{A}*_V \wedge name_V(v)_{2j} \in \mathcal{A}*_V$.
- $E \subset V^2$ *is a set of edges. It is an irreflexive and antisymmetric relation such that* $\forall \alpha \in V_A, \exists (u, \alpha) \in E$.
- $type_E : E \rightarrow \text{Types}_\text{E}$ *is a function that types edges, such that* $type_E : E \cap V_U \times V_A \rightarrow \{associate\}$, $type_E : E \cap V_U^2 \rightarrow \text{Types}_\text{E} \setminus \{associate\}$ *and* $type_E : E \cap V_A^2 \rightarrow \{specialize\}$.

[3] "$\sqcup$" denotes the disjoint union operator.

3.1.2 Block Diagrams.

The definitions below derive from the block diagrams definitions previously introduced in [24, 25].

Definition 4 (Basic Sets for Block Diagrams).

- Blocks *is a set of SysML* blocks.
- $\text{Types}_\text{A} = \{\mathbb{B}ool, \mathbb{Z}, \mathbb{N}\}$ *is a set of* attribute types.
- Attr *is a set of* attributes, *typed by* $\text{type}_\text{A} : \text{Attr} \rightarrow \text{Types}_\text{A}$.
- $\text{Profiles} = \{(t_1, \cdots, t_n) \mid n \in \mathbb{N} \wedge \forall 1 \leq i \leq n,\ t_i \in \text{Types}_\text{A}\}$.
- $\text{Sign} = \text{InSign} \sqcup \text{OutSign}$ *is a set of* signals, *typed by* $profile : \text{Sign} \rightarrow \text{Profiles}$.
  Signals may be input *or* output *signals:* InSign *contains* input *signals and* OutSign *contains* output *signals.*
- Meth *is a set of* methods, *typed by* $profile : \text{Meth} \rightarrow \text{Profiles}$.
- Ports *is a set of untyped* ports *enabling the connection of signals between* blocks *over* links. *A* link *is a pair of ports having a communication semantics.*
- $\text{CommSemantics} = \{(\{synchronous\} \times \{broadcast, unicast\} \cup \{asynchronous\} \times \{lossy, unlossy\} \times \{blocking\ write, non-blocking\ write\}) \times \{public, private\}\}$ *is the set of links communication semantics.*

Definition 5 (Block Description).
A block description *is a 10-uple* $\langle A, M, P, S_i, S_o, name_A, name_M, name_{S_i}, name_{S_o}, type \rangle$ *where* $A \subset \text{Attr}$, $M \subset \text{Meth}$, $S_i \subset \text{InSign}$, $S_o \subset \text{OutSign}$, $P \subset \text{Ports}$ *and all these sets are finite. For* $X \in \{A, M, S_i, S_o\}$, $name_X : X \rightarrow \mathcal{A}*$ *is an injective function that names each element of* X. $type \in \{system, environment\}$ *defines the type of the block.*

The concept of block *type* is introduced in Definition 5 to clearly distinguish between blocks in the BD that represent parts of the system, and those necessary for simulation or modeling the system's external communications, which depict its environment

Definition 6 (Block Diagram).
A block diagram *is a 6-uple* $\langle \mathcal{B}, d, name_B, \mathcal{L}, \sigma, C \rangle$ *where:*

- $\mathcal{B}$ *is a finite set of* blocks.
- *The function* d *assigns a description to each block in the set* $\mathcal{B}$. *For* $B \in \mathcal{B}$, *we denote* $d(B)$ *with* $\{A_B, M_B, P_B, S_{iB}, S_{oB}, name_{A_B}, name_{M_B}, name_{S_iB}, name_{S_oB}, type_B\}$, $\bigsqcup_{B \in \mathcal{B}} P_B$ *with* $\mathcal{P}$, $\bigsqcup_{B \in \mathcal{B}} S_{o_B}$ *with* $\mathcal{S}_o$ *and* $\bigsqcup_{B \in \mathcal{B}} S_{i_B}$ *with* $\mathcal{S}_i$.
- $name_B : \mathcal{B} \rightarrow \mathcal{A}*$ *is an injective function that names blocks.*
- $\mathcal{L} \subset \mathcal{P} \times \mathcal{P}$ *is a set of* links. *It is an irreflexive and antisymmetric partial injection.*
- *The function* $\sigma : \mathcal{L} \rightarrow \text{CommSemantics}$ *assigns a communication semantics to each link.*
- $C \subseteq \mathcal{L} \times \mathcal{S}_o \times \mathcal{S}_i$ *is a set of* connections $\langle \langle p_o, p_i \rangle, s_o, s_i \rangle$ *such that* p_o *and* s_o *belong to the same block, and* p_i *and* s_i *belong to the same block.*

3.2 Consistency Rules

The rule-based approach we employ in our contribution is primarily intended to tackle common errors made by LLMs (and sometimes even by experts) in constructing SysML diagrams. Consequently, the rules we propose below are rules firstly designed for LLM-generated outputs rather than comprehensive consistency rules. It

Table 1: UCD internal consistency rules

Rule	Formal expression	Reference (when applicable)
RU1. There is at least one actor and one use case in the diagram	$V_A \neq \emptyset \wedge V_U \neq \emptyset$	–
RU2. Any link shall involve two actors/use cases existing in the diagram	$E \subset V^2$. Algorithmically, our framework enforces this condition: $\forall (v_1, v_2) \in E, \exists (v_3, v_4) \in V^2$ such that $(name_V(v_3) = name_V(v_1) \wedge name_V(v_4) = name_V(v_2))$	–
RU3. Each actor/use case shall have a name	$name_V$ is a total function	–
RU4. Actor names shall start with a noun	$\forall v \in V_A, \exists i \in \mathbb{N}$ s.t. $name_V(v)_i \in \mathcal{A}*_N$	[13]
RU5. Use case names shall start with a verb	$\forall v \in V_{Uext}, \exists i \in \mathbb{N}$ s.t. $name_V(v)_i \in \mathcal{A}*_V$ $\forall v \in V_{U\overline{ext}}, \exists (i,j) \in \mathbb{N}^2$ s.t., if we denote $name_V(v)$ with $(name_V(v)_1, name_V(v)_2)$, $name_V(v)_{1i} \in \mathcal{A}*_V \wedge name_V(v)_{2j} \in \mathcal{A}*_V$	Derives from [27] and [13]
RU6. Any link between an actor and a use case shall be an association link	$type_E : E \cap V_U \times V_A \rightarrow \{associate\}$	[13]
RU7. Any link between two actors shall be an specialization link	$type_E : E \cap V_A^2 \rightarrow \{specialize\}$	–
RU8. Each actor shall be linked to at least one use case	$\forall \alpha \in V_A, \exists (u, \alpha) \in E$	–
RU9. At most one link shall exist between two given elements	E is an antisymmetric relation	–
RU10. Any link between two use cases shall be either a specialization, inclusion or extension link	$type_E : E \cap V_U^2 \rightarrow \text{Types}_\text{E} \setminus \{associate\}$	Derives from [13]

should however be noted that TTool, the framework we rely on, incorporates a syntax checker for SysML diagrams. This feature enforces numerous consistency rules established in existing literature directly through syntax verification, eliminating the necessity for these rules to be replicated in our LLM-specific rule set. Our focus, therefore, is on developing tailored rules that mitigate LLM-related inaccuracies during diagram generation. This rule list was developed through (i) a literature review on SysML consistency rules, (ii) a literature review on the use of LLMs in UML/SysML modeling, and (iii) intensive testing of BDs and UCDs LLM-based generation. These processes enabled us to identify the inconsistencies most frequently introduced by LLMs, incorporating feedback from the community as well as our own observations. Despite our rule list addresses all the inconsistencies detected—and that can be tackled with formal rules with respect to the definitions introduced in Subsection 3.1—, it is possible that some inconsistencies have not been captured in the rule definition process. Therefore this list is set to evolve.

3.2.1 UCD Internal Consistency Rules. The LLM on which our contribution relies generates UCDs that may not comply with the constraints defined in Definition 3. Our implementation ensures that generated diagrams will feature actors, use cases, and their interconnections; however, these connections may have typing errors or incorporate actors and use cases not present in the diagram. More formally, the LLM generates a 4-uple $\langle (V, E), type_V, name_V, type_E \rangle$ where $V = V_A \sqcup V_U \subset \text{Vertices}$ is a finite set of vertices, Vertices being the universe set containing all vertices, $E \subset \text{Vertices}^2$ is a finite set, $type_V$ is defined as per Definition 3, $name_V : \text{Vertices} \rightarrow \mathcal{A}*$ is a function that name vertices, and $type_E : E \rightarrow \text{Types}_\text{E}$ types elements of E. To ensure that the output is a valid UCD as per Definition 3, the generated 4-uple must be checked for the set of contraints introduced in Table 1.

3.2.2 BD Internal Consistency Rules. Similarly to UCD generation, BDs may not comply with all constraints introduced in Definition 6. Indeed, it may include attributes with types different than boolean or integer (the default ones supported by TTool), and similarly, it can produce methods whose signatures are not restricted to integer and boolean types. It may also generate signals that are not categorized as input or output, fail to type the communication semantics of links between blocks, and generate connections between blocks and signals that do not yet exist in the BD. More formally, the LLM generates a 6-uple $\langle \mathcal{B}, d, name_B, \mathcal{L}, \sigma, C \rangle$ that differs on Definition 6 on the following points:

- $\mathcal{L} \subset \text{Blocks}^2$.
- $C \subset \text{Blocks}^2 \times \text{Sign}_{gen}^2$, where where Sign_{gen} is a set of *signals*.

$\forall B \in \mathcal{B}$, if we denote $d(B)$ with $\langle A, M, P, S, name_A, name_M, name_{S_i}, name_{S_o}, type \rangle$,

- $A \subset \text{Attr}_{gen}$ where Attr_{gen} is a set of *attributes*.
- $M \subset \text{Meth}_{gen}$ where Meth_{gen} is a set of *methods*.
- $S \subset \text{Sign}_{gen}$.
- For $X \in \{B, A, M, S_i, S_o\}$, $name_X$ is not injective.

Table 2: BD internal consistency rules

Rule	Formal expression
RB1. There is at least one block in a BD	$\mathcal{B} \neq \emptyset$
RB2. Each block shall have a unique name	$\forall (B_1, B_2) \in \mathcal{B}^2, B_1 \neq B_2 \implies name_B(B_1) \neq name_B(B_2)$
RB3. Each attribute shall have a unique name	$\forall B \in \mathcal{B}, \forall (a_1, a_2) \in A_B^2, a_1 \neq a_2 \implies name_{A_B}(a_1) \neq name_{A_B}(a_2)$
RB4. Each method shall have a unique name	$\forall B \in \mathcal{B}, \forall (m_1, m_2) \in M_B^2, m_1 \neq m_2 \implies name_{M_B}(m_1) \neq name_{M_B}(m_2)$
RB5. Each signal shall have a unique name	$\forall B \in \mathcal{B}, \forall (s_1, s_2) \in (S_{iB} \sqcup S_{oB})^2, s_1 \neq s_2 \implies name_{S_B}(s_1) \neq name_{S_B}(s_2)$ where $name_{S_B} : S_{iB} \sqcup S_{oB} \to \mathcal{A}*$ $$s \mapsto \begin{cases} name_{S_iB}(s) & \text{if } s \in S_i \\ name_{S_oB}(s) & \text{if } s \in S_o \end{cases}$$
RB6. Attribute types shall be limited to either boolean or integer	$\forall B \in \mathcal{B}, \text{type}_A : A_B \to \{\mathbb{B}\text{ool}, \mathbb{Z}, \mathbb{N}\}$
RB7. In a method signature, parameters types shall be limited to either boolean or integer	$\forall B \in \mathcal{B}$, methods are typed by a function $M_B \to \{(t_1, \cdots, t_n) \mid n \in \mathbb{N} \wedge \forall 1 \leq i \leq n,\ t_i \in \text{Types}_A\}$
RB8. In a signal signature, parameters types shall be limited to either boolean or integer	$\forall B \in \mathcal{B}$, signals are typed by a function $S_B \to \{(t_1, \cdots, t_n) \mid n \in \mathbb{N} \wedge \forall 1 \leq i \leq n,\ t_i \in \text{Types}_A\}$
RB9. Signals shall be either *input* or *output*	$\forall B \in \mathcal{B}, S_B \subset \text{InSign} \sqcup \text{OutSign}$
RB10. Any link shall involve two blocks existing in the diagram	$\mathcal{L} \subset \mathcal{B}^2$. Algorithmically, TTool-AI [3] enforces this condition: $\forall (B_1, B_2) \in \mathcal{L}, \exists (B_3, B_4) \in \mathcal{B}^2$ such that $(name_B(B_3) = name_B(B_1) \wedge name_B(B_4) = name_B(B_2))$
RB11. Any link shall have a valid communication semantics	$\sigma : \mathcal{L} \to \text{CommSemantics}$
RB12. Any connection shall involve two signals existing in the blocks involved in the connection	$C \subseteq \mathcal{L} \times \mathcal{S}_o \times \mathcal{S}_i$ s.t. $\forall \langle \langle p_o, p_i \rangle, s_o, s_i \rangle \in C, \exists (B_1, B_2) \in \mathcal{B}^2$ s.t. $p_o \in P_{B_1} \wedge s_o \in SoB_1 \wedge p_i \in P_{B_2} \wedge s_i \in SiB_2$

Table 3: UCD/BD crossed consistency rules

Rule	Formal expression
RC1. No link shall exist between two *environment* blocks	$\nexists \langle B_1, B_2 \rangle \in \mathcal{L}$ s.t. $type_{B_1} = environment \wedge type_{B_2} = environment$
RC2. Any *environment* block shall have at least one link with a *system* block	$\forall B \in \mathcal{B}$ s.t. $type_B = environment, \exists \langle B, \beta \rangle \in \mathcal{L}$ s.t. $type_\beta = system$
RC3. Any *environment* block shall correspond to an actor defined in the UCD	Given a UCD $\langle (V, E), type_V, name_V, type_E \rangle$, $\forall B \in \mathcal{B}$ s.t. $type_B = environment, \exists v \in V_A$ s.t. $name_V(v) = name_B(B)$

Therefore, in order to ensure that the generation output is a valid BD as per Definition 3, the set of contraints introduced in Table 2 shall be enforced on the generated 6-uple. Please note that this not encompass all internal consistency constraints applicable to BDs: it only lists the constraints that the LLM on which TTool-AI relies usually fails to respect. However, constraints not covered in this table and ensuring the correctness of the generated diagram as per Definition 6 are addressed through the syntactic verification built into TTool. Therefore, in the worst-case scenario, if a rule not listed in Table 2 is violated by TTool-AI, users will be notified: they can either decide to ask the AI to resolve the errors, or they can manually make the necessary corrections.

3.2.3 UCD/BD Crossed Consistency Rules. Lastly, we have defined a set of consistency rules between UCDs and BDs. To the best of our knowledge, such rules do not exist in the literature. These rules target the *environment* blocks of BDs[4], ensuring that they match actors previously defined in the correspondent UCD, that these actors do not communicate between them[5] but they communicate with at least a *system* block. These rules are detailed in Table 3.

4 METHODOLOGICAL CONTRIBUTIONS AND TOOLS

Our framework is an extension of TTool-AI [3], using OpenAI's GPT-4-turbo, and GPT-4o as underlying LLMs—their integration to TTool is achieved through the implementation of a mechanism that

[4]An environment block is designed to encapsulate the environment rather than the system itself. It plays a crucial role in bridging system signals with those from the environment, enabling the simulation or verification of the system through an environmental model. It's important to note that this rule can be system-specific or based on custom recommandations.

[5]We assume that the system is already complex, so we usually assume that modeling exchanges between environmental elements would lead to capture unnecessary relations

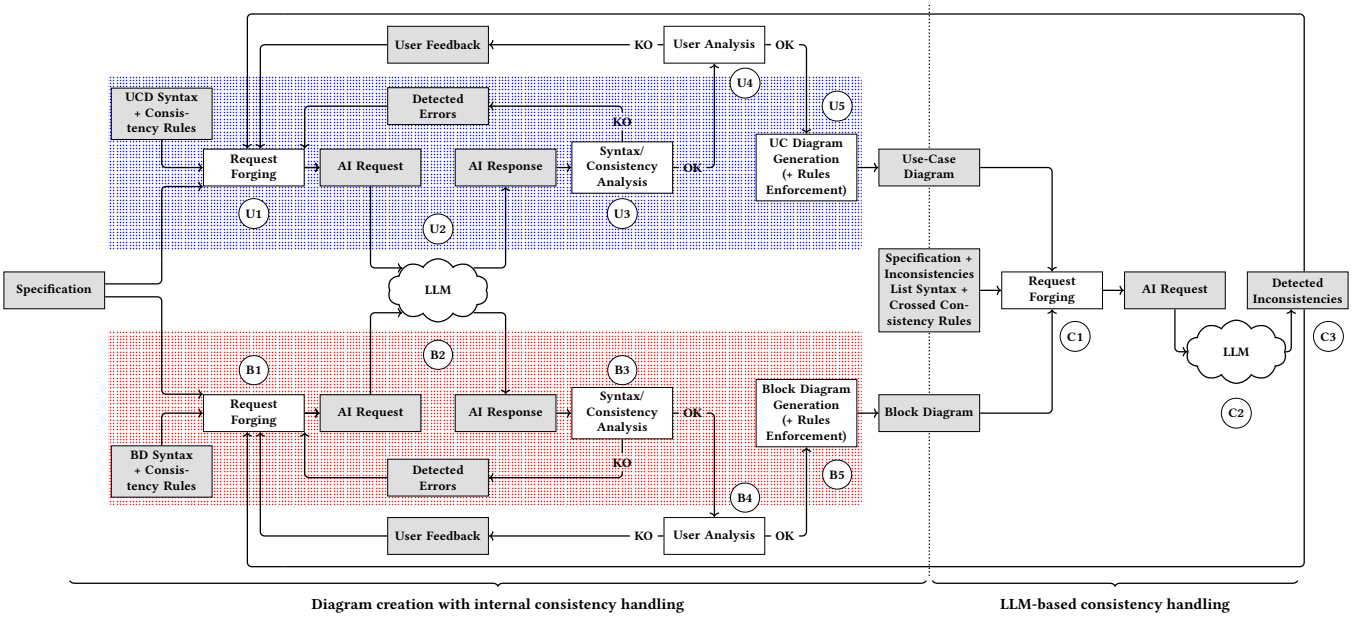

Figure 1: Functional architecture of our TTool-AI extension. Grey blocks represent data, white blocks represent actions.

allows querying the OpenAI API via a network interface. Among the upgrades performed for the present contribution, we have introduced **a new feature enabling the generation of UCDs** through AI from a system specification written in natural language. In addition, we have implemented **a new LLM-based consistency loop** to ensure consistency between several modeling views as well as internal consistency. Figure 1 gives an overview of this extension, demonstrating its application to the creation of UCDs and BDs. However, it's important to acknowledge that the same procedural framework can be applied to consistently generate any other pair of diagram types supported by TTool-AI.

This procedural framework is organized into three main stages: Initially, stages U1-5 and B1-5 run concurrently, focusing on generating preliminary UCDs and BDs using an LLM-based approach, while maintaining internal consistency within each diagram. Following this, stages C1-3 are dedicated to verifying the cross-consistency between the two diagrams and identifying any internal inconsistencies that were not previously addressed. The diagrams are then updated to rectify identified inconsistencies. A more detailed explanation of the framework's architecture is given just below.

U1 The process requires only one input from the user: a system specification, written in natural language. A textual request[6] is then formatted and integrates (1) the input specification, (2) a set of syntactic constraints "explaining" the expected response format to the LLM and (3) a set of internal UCD consistency rules (RU4, RU5, a stronger implementation of RU7[7], RU8 and a stronger version of RU10[8]).

U2 The request is then sent to the LLM we rely on. Once it has processed the request, the LLM answers with a structured response containing a UCD (described in json format).

U3 The response then undergoes a syntax and consistency analysis (RU1, RU2, RU3, RU8 and RU9 are checked at this stage). If this analysis fails, then a new request is forged from the results of the syntax analysis, then the process goes back to stage (U1).

U4 If the analysis is successful, the user can now analyze the suggested UCD. Based on their evaluation of the diagram, the user can choose to either compose a new question requesting the LLM to enhance the diagram or generate a new one (and the process then goes back to step (U2)), or decide to accept the UCD.

U5 If the user decides to accept the diagram, the framework generates a graphical UCD in TTool GUI. Consistency rules are enforced by construction at this stage, including RU1, RU2, RU3, RU6, and the stronger versions of RU7 and RU10 defined above.

B1 As in stage U1, the BD generation relies on a textual request including the system specification[9]. It may also include analysis diagrams, such as UCDs if they already exist. The request also includes a set of syntactic and consistency constraints related to BDs.

B2-5 Equivalent operations to those conducted in stages U2 through U5 are carried out in these stages, leading to a BD generation. Rules RB6 and RB8 are incorporated into the initial request. Rules RB1, RB6, RB7, and RB9 are checked at the syntax/-consistency analysis stage. At the diagram generation stage,

[6]The detailed request structure is available in https://gitlab.telecom-paris.fr/mbe-tools/TTool/-/blob/master/src/main/java/ai/AIUseCaseDiagram.java.

[7]Our implementation does not support the links between two actors. We therefore inject this rule to the LLM for RU7: $E \subset V_U \times (V_U \sqcup V_A)$.

[8]Our implementation only supports *include* links between two use cases. We therefore inject the following rule to the LLM for RU10: $type_E : E \cap V_U^2 \rightarrow \{include\}$.

[9]The detailed request structure is available in https://gitlab.telecom-paris.fr/mbe-tools/TTool/-/blob/master/src/main/java/ai/AIBlockConnAttribWithSlicing.java

rules RB3, RB5, RB6, RB7, RB10, a stronger version of RB11[10], and RB12 are enforced by construction. Stages B1 to B5 were already implemented in TTool-AI [3].

C1 After the generation of both diagrams, a new request is formulated on the basis of the two diagrams selected by the user. This request encapsulates the diagrams in a textual format along with constraints on cross-diagram consistency, which encompasses both the consistency rules and the expected syntax for responses[11].

C2 This request is sent to the LLM, which analyzes it and produces a structured list (in json) of identified inconsistencies. This can also include internal inconsistencies not detected in stages U3 and B3.

C3 With this list of inconsistencies, the process reverts to stages U1 and B1. Here, the list of inconsistencies is (partially or totally) incorporated by the user into the newly crafted requests.

This iterative process concludes either when all inconsistencies are resolved or when it reaches a predefined time limit or maximum number of iterations. Users play an integral role in this loop, choosing which inconsistencies to tackle or guiding the diagram generation by imposing additional constraints, such as 'include at least 5 actors and 10 use cases'.

5 EVALUATION

This section provides an evaluation of our framework through three distinct case studies. We begin with a step-by-step illustration of one case study, featuring generated diagrams and illustrating the interactions between our framework and the LLM. This illustration gives insights into the mechanisms of diagram generation, as well as the verification and enforcement of consistency. Subsequently, we present metrics for all three case studies to evaluate the effectiveness and relevance of our approach.

Results replicability: the experimental setup, including all input data, the models produced, and detailed guidance for using our framework and replicating our results, is documented and available on a public Zenodo archive[12].

5.1 A Step-by-Step Illustration

5.1.1 Specification Submitted to our Framework. A dynamic positioning system (DPS) is a system designed to enable a vessel to maintain a specific position and orientation, counteracting environmental forces through its propulsion and steering systems. Our specification applies to a vessel outfitted with a set of sensors including: a propeller anemometer (to measure wind force and direction), an inertial measurement unit, and a GNSS sensor. The propulsion system of the ship is equipped with actuators comprising two azimuth thrusters (which can rotate 360 degrees) and two bow thrusters (propellers located at the ship's bow on both sides of the hull). As azimuth thrusters enables the vessel to control its orientation and speed simultaneously, the propulsion and steering systems are combined.

The request is to design both the console and the controller of the DPS.

5.1.2 Diagram Generation. Following the process outlined in Figure 1, we task TTool-AI with generating both UCDs and BDs based on the given specification. The resulting UCD, produced after completing stages U1 through U5, is depicted in Figure 2. The generation logs show that the diagram initially produced by the LLM did not comply with rule RU8 (*each actor shall be linked to a use case*):

```
- Actor "Azimuth_Thrusters" must be connected to at least
    one use case
- Actor "Bow_Thrusters" must be connected to at least one
    use case.
```

These two internal inconsistencies were then addressed through the automated feedback mechanism. As a result, the final version of the UCD adheres fully to the guidelines listed in Table 1. Furthermore, TTool's syntax checker found no errors in the generated diagram, validating the efficiency of the use-case diagram generation process. However, it should be noted that there is a misspelling in the actor representing the anemometer, incorrectly labeled as `Propeller_Anerometer`.

In parallel, a BD was generated using stages B1 to B5 of our framework, as illustrated in Figure 2. The generation logs do not show any rule violations, and TTool's syntax checker confirmed the absence of errors in the diagram. However, observations from our review include:

- The presence of a potentially superfluous `DPS` block (this block is unrelated to any other block)
- The absence of a `User` block, interacting with the Console, which would enhance verification and simulation coverage by enabling user-driven scenarios.

5.1.3 LLM-Based Inconsistency Detection. The two diagrams subsequently undergo a consistency analysis (stages C1 to C3). Our strategy involves supplying the LLM with the system specification and textual representations of the UCD and the BD. TTool is equipped to generate textual specifications of these diagrams in SysML v2 format. However, this format's verbosity leads to extensive contexts, affecting both the cost and the quality of results. To mitigate this, we have developed a more concise textual format, based on element lists. For example, the textual representation of the UCD is structured as follows:

```
actors: User Propeller_Anerometer ...
Use cases: Define_PositionAndCourse ...
Connections: include(Activate_BowThrusters,
    Maintain_SetPositionAndCourse) ...
```

In addition, specific constraints are automatically included in the request (*Inconsistencies List Syntax* in Figure 1). These constraints contain mostly the specification of the output format:

```
"When you are asked to identify all the relevant
    incoherencies between two diagrams, return them as a
    JSON specification formatted as follows:{
    incoherencies: [{ \"diagram\" :\"diagram1 or
    diagram2\", \"description\": "description of the
    incoherency\"}..."]}";
```

[10] We enforce the following rule: $\sigma : \mathcal{L} \rightarrow \{(synchronous, unicast, private)\}$

[11] The detailed requests are defined in https://gitlab.telecom-paris.fr/mbe-tools/TTool/-/blob/master/src/main/java/ai/AIDiagramCoherency.java and in https://gitlab.telecom-paris.fr/mbe-tools/TTool/-/blob/master/src/main/java/ai/AIDiagramCoherencyWithFormalRules.java

[12] https://zenodo.org/doi/10.5281/zenodo.11936921

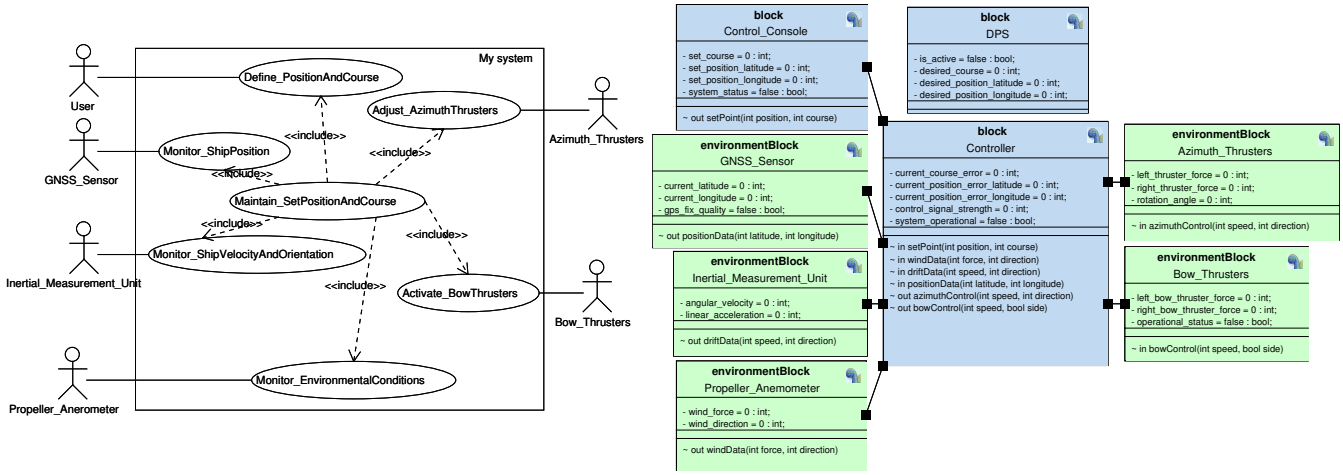

Figure 2: The generated UCD and BD

Formal rules defined in Section 3 may also be part of these constraints. They are given in natural language, for instance:

```
#Respect: In a block diagram, the blocks representing
    actors as defined in the use case diagram must bear
    identical names to their corresponding use cases.
```

Finally, the request sent to the AI engine contains the constraints, the question, the system specification and the UCD and BD in textual format added by the user. A typical answer is structured as follows:

```
AI: json
{ "incoherencies": [
  ...
  {
    "diagram": "Diagram2",
    "description": "The User actor has no direct relation
        to any block in Diagram 2."
  },
  ...
}
```

Using this approach wih the UCD and BD depicted in Figure 2, the framework identified a list of seven inconsistencies, some of which corroborated our observations:

- Misspelling of `Propeller_Anemometer` in the UCD (*The Propeller_Anemometer block is misspelled as 'Propeller_ Anerometer' in [the use case] diagram, The use case 'Monitor_ EnvironmentalConditions' is associated with 'Propeller_Anerometer', which does not have a corresponding block in [the block] diagram. The correct block name should be 'Propeller_ Anemometer'*).
- Absence of the `User` block in the BD (*The User actor has no direct relation to any block in [the block] diagram, There is no explicit block for the 'User' actor in [the block] diagram, making the connection to 'Define_PositionAndCourse' unclear, The use case 'Define_PositionAndCourse' does not have a corresponding signal or method in the blocks of Diagram 2*).
- Presence of the `DPS` block in the BD (*The 'DPS' block does not appear in [the use case] diagram, nor is it connected to any use case or actor explicitly*).

Additionally, the list included another "inconsistency" that seems irrelevant (*The block 'Controller' does not have a direct association with the 'User' or specific use cases as in [the use case] diagram*).

5.1.4 Inconsistencies Correction. Thereafter, we task our framework to correct the detected inconsistencies. This correction relies on the TTool-AI BD generation feature [3] and on the UCD generation feature (a contribution of the paper). The request sent to the LLM includes the constraints related UCD or BD (expected syntax, internal consistency rules) and a message provided by the user including the DPS specification, a textual representation of the diagrams to correct (Figure 2), and the relevant identified inconsistencies. These inconsistencies are included as follows:

```
Do correct the block diagram considering the following
    incoherencies:
1. ...
2. The User actor has no direct relation to any block in
    Diagram 2.
3. ...
Do correct incoherencies 1-3 and propose a new  block
    diagram.
```

The framework responded by producing two revised diagrams, as shown in Figure 3. We can observe that not all inconsistencies were corrected. For example, `DPS` block is still unrelated to other blocks. Nevertheless, the two primary categories of inconsistencies were addressed: the actor `Propeller_Anemometer` is now correctly spelled in the UCD, and a `User` block has been added to the BD. Another iteration on inconsistencies (stages C1 to C3) could resolve these remaining issues.

5.2 Evaluation

We have considered three different systems: an automotive braking system, a space-based system, and the dynamic positioning system. The two first systems are use cases taken from two distinct European projects. The specifications for these systems are available in the Zenodo archive that accompanies the paper. The archive includes, for each of the three systems, a *md* file detailing the system specifications and an *xml* file containing the TTool model. The

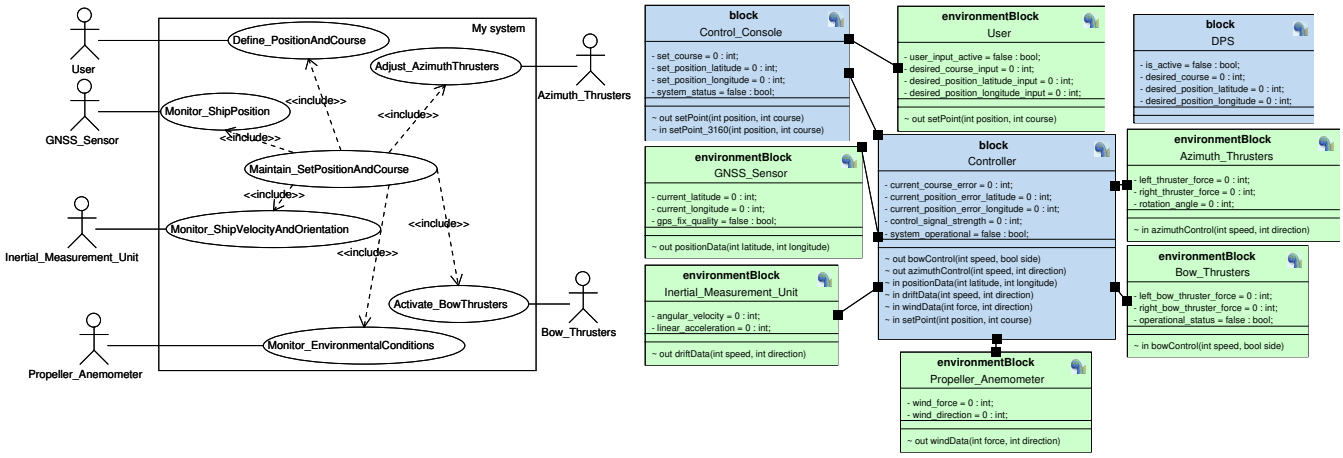

Figure 3: The updated UCD and BD after LLM-based consistency loop

model file presents diagrams and detected inconsitencies generated from the following stages:

(1) **Model generation**: stages U1 to U5 and B1 to B5 of our framework, applied to generate two BDs and two UCDs per case study. All diagrams of the automotive system and of the space-based system were generated using GPT 3.5, and the diagrams of DPS were generated with GPT 4.
(2) **Inconsistency detection**: stages C1 to C3 of our approach. GPT 4 was used to identify inconsistencies.
(3) **Inconsistency correction**: updated diagrams (BD, UCD) generated using the system specifications, the previously modeled diagrams, as well as the list of identified inconsistencies (apart from the erroneous inconsistencies that are dropped, see below). GPT 4 was used to perform these diagram updates.

For these evaluations, TTool version 3.0 beta, build: 14731, was used.

5.3 Results

We evaluated the outlined processes of diagram generation, inconsistency identification, and inconsistency resolution on the three systems. The outcomes are summarized in Table 4. We differentiate between internal inconsistencies (within a single diagram) and external inconsistencies (between a block diagram and a use case diagram). Inconsistencies incorrectly identified are cataloged in the "Error" column; they are excluded from the total inconsistency count and are not addressed during the correction phase.

Our methodology successfully detected multiple inconsistencies per pair (UCD, BD), averaging 4 inconsistencies for BDs and 1.7 for UCDs. Across all evaluations, 6 inconsistencies were erroneously identified—these are invalid incoherences, such as the erroneous assertion that two already connected blocks should be connected. This represents 8% of the detected inconsistencies, meaning that 92% of detected inconsistencies were relevant. In terms of inconsistency correction, our approach enabled for an automatic resolution of 87% of the inconsistencies on average, demonstrating a slightly higher correction rate for external (cross-diagram) inconsistencies.

6 DISCUSSION

The evaluation of our approach, as detailed above, shows encouraging outcomes in both the initial generation of syntactically consistent diagrams and the subsequent detection and correction of inconsistencies within and across those diagrams. Our analysis highlights the significant benefits derived from the integration of internal consistency mechanisms and LLM-based approaches. Despite these advancements, there are several avenues for further improvement, as now discussed.

The management of rules outlined in Table 3 within the LLM-based consistency handling loop was challenging. Specifically, when these rules are incorporated into the knowledge database injected in the consistency request (see Stage C1 of Figure 1), the LLM tends to exclusively focus on these rules, thus ignoring other consistency aspects. To address this issue, we have introduced two separate features in TToolAI: one enabling users to evaluate consistency considering the embedded rules in the request, and the other allowing for consistency checks to be performed without these rules. As a result, to achieve a comprehensive cross-view consistency evaluation, users are currently required to engage TToolAI sequentially in two different operations. This may also be seen as an advantage since users of TTool can somehow customize the consistency rules they intend to address. In a related manner, the detection of inconsistencies related to these rules (concerning *environment* blocks) could be improved: currently, the block type as defined in Definition 5 is not exported to the textual format generated by TTool from BDs. Therefore, the classification of system/environment blocks relies on the LLM's analysis, based on the provided UCD and specification. Exporting it to the textual format would reduce the possible LLM's interpretation errors here. Likewise, it would be advantageous to enforce these rules by construction, during the diagram generation phase, as it is done in stages U5 and B5 for internal consistency rules. This textual format provided by TTool condenses the description of the exported diagram, offering a succinct way to communicate the diagram's details. This efficiency is particularly beneficial for minimizing the use of tokens when submitting requests to the LLM. However, given that this format is not a standard format, exporting

Table 4: Key metrics on cross-view consistency handling.

System	Test	Inconsistencies detected					Inconsistencies corrected		
		Diagram	Internal	External	Errors	Total	Internal	External	Total
Automated braking	BD1 vs UCD1	*BD1*	1	2	0	3	1	2	3/3
		UCD1	0	0	0	0	0	0	—
	BD1 vs UCD2	*BD1*	0	1	0	1	0	1	1/1
		UCD2	0	3	0	3	0	2	2/3
	BD2 vs UCD1	*BD2*	5	1	1	6	4	1	5/6
		UCD1	0	1	1	1	0	1	1/1
	BD2 vs UCD2	*BD2*	4	2	0	6	3	1	4/6
		UCD2	2	2	0	4	2	2	4/4
Space-based system	BD1 vs UCD1	*BD1*	3	6	0	9	3	5	8/9
		UCD1	0	0	0	0	0	0	—
	BD1 vs UCD2	*BD1*	4	1	0	5	3.5	1	4.5/5
		UCD2	3	1	0	4	2.5	1	3.5/4
	BD2 vs UCD1	*BD2*	2	2	0	4	1	2	3/4
		UCD1	1	1	1	2	1	1	2/2
	BD2 vs UCD2	*BD2*	1	4	0	5	1	4	5/5
		UCD2	0	2	0	2	0	2	2/2
Dynamic positioning system	BD1 vs UCD1	*BD1*	1	1	0	2	1	0	1/2
		UCD1	0	0	1	0	0	0	—
	BD1 vs UCD2	*BD1*	2	2	0	4	2	1.5	3.5/4
		UCD2	2	0	0	2	0	2	2/2
	BD2 vs UCD1	*BD2*	1	0	0	1	1	0	1/1
		UCD1	1	1	0	2	1	1	2/2
	BD2 vs UCD2	*BD2*	3	0	1	3	3	0	3/3
		UCD2	0	0	1	0	0	0	—
Total			36	33	6	69	30	30.5	60.5/69

diagrams in SysML v2 format may enhance the LLM's understanding of the diagrams—even though this understanding is already excellent with the current export format. This may decrease the rate of "false positives" among the identified inconsistencies (7% in our evaluation).

The process of correcting these detected inconsistencies could also be improved. In our experiments, we integrated the entire list of detected inconsistencies (excluding those categorized as errors) into the message input in TToolAI for generating revised diagrams. Adopting a strategy of addressing each inconsistency individually could potentially elevate the correction rate (which varies between 50% and 100% per diagram, averaging at 87%). LLMs tend indeed to produce more accurate results when their input is more concise: therefore, providing the LLM with one inconsistency at a time would probably help it focus better and improve its performance. Note also that there is subjectivity in the classification of the detected inconsistencies (determining their relevance to specific diagrams, identifying them as errors). However, until recently, crossed-view consistency was mostly performed manually until now in TToolAI (as in most UML/SysML toolkits). Moreover, the presence of an error in the list of detected inconsistencies does not necessarily means that this error will be introduced into the updated diagrams. Indeed, the automated feedback loop of our framework, and the enforcement by design of several internal consistency rules, help eliminate errors introduced in the LLM generation process. However, we have not yet quantified this phenomenon, but it would surely be interesting to evaluate it in the future.

Moreover, our evaluation relied on three case studies involving relatively simple diagrams and it would be interesting to assess our framework using more complex diagrams. Given the graphical nature of our approach and the limited input context sizes of the LLMs we utilize, we think that the most effective method to manage scalability is by decomposing complex models into several sub-models (e.g., by using model decomposition and hierarchical representations). Methods such as [19] exist for this, and a comprehensive strategy could integrate these methods with our proposed approach to manage complexity. Additionally, one of the goals of our approach is to ensure the consistency of model segments designed in different silos, thereby maintaining overall consistency during the reconstruction of the entire model. Therefore, this decomposition/recomposition approach is a possible way to scale our framework.

Finally, our evaluation evaluates the cross-consistency of only two classes of diagrams: BDs and UCDs. It would be interesting to evaluate it on other views, particularly on cross-consistency between UCDs and SMDs: indeed, when generated by TTool-AI, SMDs often contain errors detected by TTool-AI's syntax checker.

7 CONCLUSIONS

The paper presents a new LLM-based framework designed to enhance the cross-consistency of SysML diagrams, along with introducing an automated UCD generation feature. Additionally, it introduces adapted consistency rules aimed at ensuring the internal consistency of LLM-generated UCDs and BDs. This framework has been implemented as an extension of TTool-AI. Through evaluations conducted on three case studies, it has demonstrated effectiveness in generating internally consistent diagrams and in detecting and rectifying inconsistencies between UCDs and BDs.

However, there is potential for further enhancements. Future work will focus on refining and fully automating the correction process to enhance accessibility for the potential users, enabling them to easily insert their own rules, ensuring by construction cross-consistency rules, minimizing the LLM's interpretive scope by detailing the block types, and assessing—and, if necessary, adapting—our framework for additional diagram types such as requirement and state-machine diagrams. Currently, our implementation supports UCDs, BDs, and SMDs and to add support for other diagram types, we need to implement the textual format export for these additional diagrams. Integrating our implementation with other LLMs would also be beneficial for comparing performance and diversifying responses. Additionally, interfacing our framework with other modeling tools using TTool's command-line interface could be interesting. This interfacing should include automated export, consistency-checks/improvements, and reimportation of the models into the other tools.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Aakash Ahmad, Muhammad Waseem, Peng Liang, Mahdi Fahmideh, Mst Shamima Aktar, and Tommi Mikkonen. 2023. Towards Human-Bot Collaborative Software Architecting with ChatGPT. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. 279–285.

[3] Ludovic Apvrille. and Bastien Sultan. 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'24*. INSTICC, SciTePress, 27–38. https://doi.org/10.5220/0012320100003645

[4] Oskar Berglund. 2024. Assessing Strategies for Behaviour Consistency Checking Using LLMs. B.Sc. Thesis.

[5] Alessandro Tempia Calvino and Ludovic Apvrille. 2021. Direct Model-Checking of SysML Models. In *9th International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS-Science and Technology Publications, 216–223.

[6] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. 2023. On the Assessment of Generative AI in Modeling Tasks: an Experience Report with ChatGPT and UML. *Software and Systems Modeling* 22, 3 (2023), 781–793.

[7] Boqi Chen, Kua Chen, Shabnam Hassani, Yujing Yang, Daniel Amyot, Lysanne Lessard, Gunter Mussbacher, Mehrdad Sabetzadeh, and Dániel Varró. 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, 262–271.

[8] K. Chen, Y. Yang, B. Chen, J. Hernandez Lopez, G. Mussbacher, and D. Varro. 2023. Automated Domain Modeling with Large Language Models: A Comparative Study. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE Computer Society, 162–172. https://doi.org/10.1109/MODELS58315.2023.00037

[9] Musheng Chen, Guowei He, and Junhua Wu. 2024. ZDDR: A Zero-Shot Defender for Adversarial Samples Detection and Restoration. *IEEE Access* (2024).

[10] Sylvain Guérin, Joel Champeau, Jean-Christophe Bach, Antoine Beugnard, Fabien Dagnat, and Salvador Martínez. 2022. Multi-Level Modeling with Openflexo/FML: a contribution to the multi-level process challenge. *Enterprise Modelling and Information Systems Architectures (EMISAJ)* 17 (2022), 9–1.

[11] Leon Görgen., Eric Müller., Marcus Triller., Benjamin Nast., and Kurt Sandkuhl. 2024. Large Language Models in Enterprise Modeling: Case Study and Experiences. In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering - MODELSWARD*. INSTICC, SciTePress, 74–85. https://doi.org/10.5220/0012387000003645

[12] David Harel., Guy Katz., Assaf Marron., and Smadar Szekely. 2024. On Augmenting Scenario-Based Modeling with Generative AI. In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering - MODELSWARD*. INSTICC, SciTePress, 235–246. https://doi.org/10.5220/0012427100003645

[13] Noraini Ibrahim, Rosziati Ibrahim, Mohd Zainuri Saringat, Dzahar Mansor, and Tutut Herawan. 2010. On well-formedness rules for UML use case diagram. In *Web Information Systems and Mining: International Conference, WISM 2010, Sanya, China, October 23-24, 2010. Proceedings*. Springer, 432–439.

[14] Noraini Ibrahim, Rosziati Ibrahim, Mohd Zainuri Saringat, Dzahar Mansor, and Tutut Herawan. 2011. Consistency Rules Between UML Use Case and Activity Diagrams using Logical Approach. *International Journal of Software Engineering and its Applications* 5, 3 (2011), 119–134.

[15] Diana Kalibatiene, Olegas Vasilecas, and Ruta Dubauskaite. 2013. Rule Based Approach for Ensuring Consistency in Different UML Models. In *Information Systems: Development, Learning, Security: 6th SIGSAND/PLAIS EuroSymposium 2013, Gdańsk, Poland, September 26, 2013. Proceedings 6*. Springer, 1–16.

[16] Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. 2013. Formal System-Level Design Space Exploration. *Concurrency and Computation: Practice and Experience* 25, 2 (2013), 250–264. https://doi.org/10.1002/cpe.2802 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.2802

[17] Ziyu Li and Donghwan Shin. 2024. Mutation-Based Consistency Testing for Evaluating the Code Understanding Capability of LLMs. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*. 150–159.

[18] Shan Lu, Alexey Tazin, Yanji Chen, Mieczyslaw M Kokar, and Jeff Smith. 2023. Detection of inconsistencies in SysML/OCL models using OWL reasoning. *SN Computer Science* 4, 2 (2023), 175.

[19] Qin Ma, Pierre Kelsen, and Christian Glodt. 2015. A generic model decomposition technique and its application to the Eclipse modeling framework. *Software & Systems Modeling* 14 (2015), 921–952.

[20] Tom Mens, Ragnhild Van Der Straeten, and Jocelyn Simmonds. 2005. A framework for managing consistency of evolving UML models. In *Software Evolution with UML and XML*. IGI Global, 1–30.

[21] Hiroyuki Nakagawa and Shinichi Honiden. 2023. MAPE-K Loop-Based Goal Model Generation Using Generative AI. In *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. IEEE, 247–251.

[22] Gabriel Pedroza, Ludovic Apvrille, and Daniel Knorreck. 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*. 1–10. https://doi.org/10.1109/NOTERE.2011.5957992

[23] Iris Reinhartz-Berger. 2005. Conceptual Modeling of Structure and Behavior with UML–The Top Level Object-Oriented Framework (TLOOF) Approach. In *International Conference on Conceptual Modeling*. Springer, 1–15.

[24] Bastien Sultan, Ludovic Apvrille, Philippe Jaillon, and Sophie Coudert. 2023. W-Sec: A Model-Based Formal Method for Assessing the Impacts of Security Countermeasures. In *Model-Driven Engineering and Software Development*, Luís Ferreira Pires, Slimane Hammoudi, and Edwin Seidewitz (Eds.). Springer Nature Switzerland, Cham, 203–229.

[25] Bastien Sultan, Léon Frénot, Ludovic Apvrille, Philippe Jaillon, and Sophie Coudert. 2023. AMULET: a Mutation Language Enabling Automatic Enrichment of SysML Models. *ACM Trans. Embed. Comput. Syst.* (sep 2023). https://doi.org/10.1145/3624583

[26] Damiano Torre, Yvan Labiche, and Marcela Genero. 2014. UML Consistency Rules: a Systematic Mapping Study. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. 1–10.

[27] Damiano Torre, Yvan Labiche, Marcela Genero, and Maged Elaasar. 2018. A Systematic Identification of Consistency Rules for UML Diagrams. *Journal of Systems and Software* 144 (2018), 121–142.

[28] Carlos Mario Zapata, Guillermo González, and Alexander Gelbukh. 2007. A Rule-Based System for Assessing Consistency Between UML Models. In *MICAI 2007: Advances in Artificial Intelligence: 6th Mexican International Conference on Artificial Intelligence, Aguascalientes, Mexico, November 4-10, 2007. Proceedings 6*. Springer, 215–224.