Automated Derivation of Formal Properties from Requirements

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

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

Pierre de Saqui-Sannes
Fédération ONERA ISAE-SUPAERO ENAC
Université de Toulouse
Toulouse, France
pdss@isae-supaero.fr

***Abstract*—Early detection of design errors in the life cycle of systems is one of the many benefits one may expect from using an model-based systems engineering approach. The formal verification of models is thus an issue. Among verification techniques, the paper focuses on model checking. Taking as input a set of properties and a model, model checking outputs a "yes/no" answer for each property saying whether the property is verified or not. The relevance of this proof relies on the relevance of the defined properties. The paper proposes solutions for automated generation of properties from a SysML model and, more specifically, from the requirements expressed in the SysML model. We introduce an AI- and syntactic rule-based process that assists systems engineers in the generation of syntactically correct CTL properties, along with complementary contributions for computing and propagating requirement coverage from formal verification results, backtracing them directly in the SysML requirement diagram. The approach is mathematically defined, with explanatory comments accompanying the formalism to support comprehension. The proposed approach is further implemented by the free software TTool-AI. The latter is an open source SysML toolkit extended with an LLM-based assistant. A satellite system and an embedded maritime system serve as case studies.**

***Index Terms*—MBSE, MDE, Formal verification, Properties generation, SysML, CTL, AI, LLM.**

I. Introduction

Modeling systems using an MBSE approach has many advantages [1], [2]. Early verification of design errors in the life cycle of systems is one of these advantages. Model checking [3] is a way to achieve that early verification objective. Model checking takes as input a model and a set of properties, such as computational tree logic (CTL) properties. For each property, a "yes/no" answer indicates whether the property is satisfied or not. Putting model checking into practice predominantly requires us to express the properties to be verified. Properties can be designed manually without computer-based assistance. This is risky. According to [10], "current practices require significant manual efforts to create such properties, making them time-consuming, costly, and error-prone."

The question of how to automate the generation of verification properties is the subject discussed in the current paper, focusing the discussion on systems modeled in SysML [8]. The contributions of this work are twofold: (i) an AI-based and rule-based process for helping systems engineers generate syntactically correct CTL properties from a SysML design model and a set of requirements, and (ii) additional contributions enabling the integration of this process into the MBSE workflow, including automated requirement extraction from requirement diagrams, computation of requirement coverage from the results of model checking of the generated properties, and backtracing of these coverage indicators to requirement diagrams. These contributions are fully implemented by the free software TTool, an open source SysML toolkit extended with an LLM-based assistant.

The paper is organized as follows. Section II introduces TTool-AI. Section III surveys related work. Section IV presents our AI-driven CTL property generation process and its complementary contributions. Section V illustrates the application of our contributions to two systems. Section VI concludes the paper and outlines future work.

II. TTool-AI: Context

TTool is an open-source SysML toolkit that supports a wide range of MBSE tasks, from specification analysis to code generation. It enables users to design models both graphically and textually, including SysML analysis diagrams (requirements, use case, sequence, and activity diagrams) and design models (block diagrams and state machine diagrams). These diagrams are grounded in formal semantics, allowing three key capabilities: syntax checking, simulation, and formal verification through model checking [16].

Over the last two years, TTool has been extended with a LLM-based assistant called TTool-AI [17], which supports users across several of the toolkit's features, such as automatic diagram generation from textual specifications, diagram enrichment via natural language requests, and consistency improvement [18]. TTool-AI relies on retrieval-augmented generation (RAG) and an automated feedback loop based on algorithmic analysis of LLM outputs: when a response fails to meet a predefined quality threshold for correctness, TTool-AI iteratively refines its prompts until the LLM converges to a satisfactory answer. The present paper builds on this framework to extend TTool-AI with a CTL property generation feature that ensures the advanced syntactic correctness of the generated properties.

III. Related Work

A. Formalized Requirements

In [11] Bruel et al. discuss the importance and role of formalism in system requirements. The latter should be both understandable and precise. Most requirements are written in natural language, good for understandability but lacking in precision. To make requirements precise, researchers have for years advocated the use of mathematics-based notations and methods, known as "formal". Many exist, differing in their style, scope, and applicability. The paper [11] analyzes formal and semi-formal modeling languages, SysML being a semi-formal one. Regarding SysML, the authors of [11] say that a formal expression of requirements, and thus their formal verification is not possible in SysML. A point of view that could be moderated, according to the authors of the current paper: indeed, SysML profiles with formal foundations exist. For example, AVATAR [19], a SysML profile supported by TTool that includes implemented transformations to SysML/SysML v2. In particular, AVATAR *design models* (*i.e.*, a block diagram and a set of state machine diagrams, one per block) can be verified against CTL properties by model checking, without requiring any intermediate transformation to another modeling formalism [16]. The present paper builds on this capability to translate requirements into CTL properties and verify these properties on SysML design models to estimate requirement coverage.

According to [7], existing methodologies often rely on natural language requirements, prone to ambiguities and errors. Furthermore, several methods formalize the requirements using temporal logic languages, which can be checked algorithmically. Currently, a gap remains in integrating the comprehensive generation of operating states and transitions with the formalization in temporal logic languages, specifically for the use in MBSE development environments. [7] introduces a multi-step approach that formalizes requirements using Linear Temporal Logic (LTL) and generates operating states and transitions, which are then modeled in a SysML v2 state machine diagram, enhancing both the development process and the reliability of the system.

B. Property Generation for Verification Purposes

[14] Wu et al. state that model checking can be viewed in terms of automatically generating properties, directly verifying properties, finding counter-examples, and refining abstraction based on counter-examples. The current paper emphasizes on automatic generation of properties and so does the current section.

In [4] Cimatti et al. propose a new methodology for requirements validation, based on the use of formal methods. The methodology is a three-step one: (1) an informal analysis resulting in a structured version of the requirements where each fragment is classified according to a fixed taxonomy; (2) each fragment is then mapped onto a subset of UML with precise semantics and enriched with static and temporal constraints; (3) application of specialized formal analysis techniques, optimized to deal with properties (rather than models).

The authors of [9] propose a property generation approach for verification purposes. In the first phase, a set of predefined properties is hypothesized over the design behavior, which is described by a given simulation trace. During the subsequent second phase, the previously found properties are combined to form new, more complex candidates. These candidates are checked in the simulation trace once again. "Surviving" property candidates are recombined until no more valid properties can be created. The extracted functional behavior is then presented to the verification engineer for manual inspection.

In [13], Bao et al. propose GAPG, an automatic property generation method based on generative adversarial networks. GAPG takes as input either a set of CTL properties or a set of CTL properties together with a system model (represented as a Kripke structure). From these inputs, GAPG generates new CTL properties that complement those provided.

To our knowledge, the work most closely related to ours was proposed by Ankireddy et al. [10]. Their paper introduces LASA, an LLM-based framework for generating security properties expressed in temporal logic in the context of System-on-Chip (SoC) design. LASA extracts behavioral descriptions from register-transfer level (RTL) code and specification documents, which are provided as input to the LLM along with contextual information from additional sources (*e.g.*, manuals). Security properties are then generated through a feedback loop that ensures that only non-vacuous properties are produced and subsequently translated into syntactically correct SystemVerilog Assertions (SVAs) via another LLM-based loop. Finally, SVAs are algorithmically analyzed using Cadence JasperGold to compute a score quantifying their ability to cover system behavior. LASA shares several similarities with our approach, including an RAG-based LLM loop with algorithmic syntax verification and an automated feedback mechanism. However, the two approaches differ in several key aspects: their input modeling languages (RTL for LASA vs. SysML for our method), their scope (LASA focuses on SoC design, whereas our approach targets generic systems engineering) and the purpose. Unlike LASA, our approach takes system requirements as input, translates them into CTL properties, evaluates these properties on the SysML model using model checking, and computes requirement coverage values, which are then backtraced to the requirements to indicate whether each requirement is fully covered, partially covered, or uncovered.

IV. Contributions to Automated CTL Property Generation within MBSE

This section presents our theoretical and methodological contributions aimed at assisting engineers in deriving CTL properties that enable verification of requirement satisfaction by a system design (see Figure 1). It first introduces preliminary definitions that formalize the notions of requirements and requirement diagrams on which our framework relies, providing the foundation for the further contributions. Then, it describes our AI- and syntactic-rule-based CTL property generation process, which generates syntactically correct CTL

properties from a design model and a set of requirements. Finally, it presents complementary contributions that combine with the generation process to form a framework that integrates into the MBSE, and, in particular, in the formal verification workflow. All of these contributions have been implemented in TTool.

A. Preliminary Definitions

Definition 1: Basic Sets

- $\mathsf{Requirements}$ is the universe set of *requirements*.
- $\mathsf{CTLFormulas}$ is the universe set of *CTL formulas*.
- $\mathcal{A} = \{a, A, \ldots, z, Z\} \cup \{0, 1, \ldots, 9\} \cup \{\mathtt{.}, \mathtt{_}, \mathtt{;}, \mathtt{,}, \llcorner\}$ is the alphabet and $\mathcal{A}*$ is the set of finite words induced by the Kleene closure over $\mathcal{A}$.
- $\mathbb{L}_3 = \{\top, \mathbb{U}nknown, \bot\}$ is the set of truth values in ternary logic, and $\mathbb{B} = \{\top, \bot\}$.

The following two definitions formalize the notions of *requirement* and *requirement diagram* as used in TTool. These definitions are compatible with the SysML (v2) standard and do not aim to capture the full semantics of these concepts as implemented in TTool. They rather focus on the essential aspects required to support the presentation of our subsequent contributions.

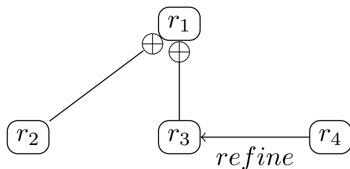
Definition 2: Requirement Description
A *requirement description* is a triplet $\langle id, name, text\rangle \in \mathbb{N} \times \mathcal{A}* \times \mathcal{A}*$ where id is a requirement identifier, $name$ names the requirement and $text$ gives its textual description.

$\mathsf{desc} : \mathsf{Requirements} \to \mathbb{N} \times \mathcal{A}* \times \mathcal{A}*$ is a total function that assigns a requirement description to each requirement. It is such that $\pi_{\mathbb{N}} \circ \mathsf{desc}$ is injective (where $\pi_{\mathbb{N}}$ is the canonical projection from $\mathbb{N} \times \mathcal{A}* \times \mathcal{A}*$ onto $\mathbb{N}$).

Definition 3: Requirement Diagram
A *requirement diagram* (RD) is a triplet $\langle Req, \mathcal{R}, \mathsf{type}\rangle$ where:

- $Req \subset \mathsf{Requirements}$ is a finite set of requirements
- $\mathcal{R} \subseteq Req \times Req$ is an irreflexive binary relation that induces a directed acyclic graph on Req
- $\mathsf{type} : \mathcal{R} \to \{compose, derive, refine\}$ is a function that types edges of $\mathcal{R}$. For $type \in \{compose, derive, refine\}$, we denote by $\mathcal{R}_{type}$ the set $\{r \in \mathcal{R} \mid \mathsf{type}(r) = type\}$
- $\mathcal{R}$ is such that $(r_2, r_1) \in \mathcal{R}_{compose} \implies \exists r_3 \in Req, r_3 \neq r_2 \wedge (r_3, r_1) \in \mathcal{R}_{compose}$.

Example 1: Requirement Diagram
Consider the RD $\langle\{r_1, r_2, r_3, r_4\}, \{(r_2, r_1), (r_3, r_1), (r_4, r_3)\}, \{((r_2, r_1), compose), ((r_3, r_1), compose), ((r_4, r_3), refine)\}\rangle$.



In this RD, r_1 is composed of r_2 and r_3: its meaning is fully captured by r_2 **and** r_3 together, which decompose r_1 into sub-requirements of narrower scope. r_4 refines r_3: it provides further details, at a lower level of abstraction, about (part of) r_3's meaning.

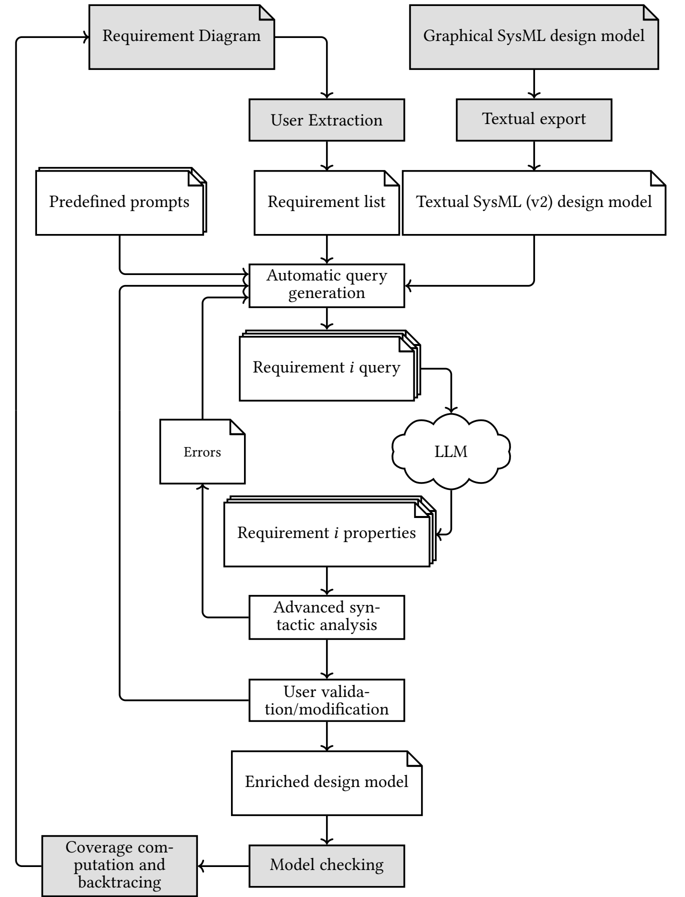


Fig. 1. AI-driven property generation process. Grey nodes represent the complementary steps (some of them being contributions of this paper) and data enabling the process integration within the MBSE workflow.

B. An AI-Driven CTL Property Generation Process

The first contribution of the paper is a TTool-AI extension that enables user assistance in the generation of CTL properties from a design model and a list of requirements (see Figure 1). The process operates through the following steps:

1. Query generation: based on a SysML design model (a block diagram and a set of associated state machine diagrams) and a list of requirements, TTool-AI generates a set of queries (one per requirement). Each query includes the textual representation of the model, the corresponding requirement, and a set of predefined prompts that enable RAG aiming at enriching the LLM's context with specific knowledge, including: the system specification, the expected response format, the syntax and semantics of the CTL subset we rely on, and an instruction prompting the LLM to generate zero (if the requirement cannot be translated), one, or several CTL properties.

2. Response verification: each query is sent to the LLM, whose response is then parsed and algorithmically verified using both an ad-hoc syntax checker for the LLM output format and TTool's built-in CTL syntax checker. This verification step detects several types of errors:

- Structural errors in the LLM's response (*e.g.*, unparsable format).
- Syntactic errors in the CTL properties (*i.e.*, elements that do not respect the CTL grammar).
- Advanced syntactic errors (*i.e.*, references to nonexistent blocks, states, or attributes in the design model).

If any errors are detected, a new query summarizing these issues is sent back to the LLM, and the process iterates until either (i) all errors are resolved or (ii) a user-defined iteration limit is reached.

3. Model enrichment: once validated, all generated properties are compiled into a global list. The user may request further refinement of this list or accept it as is. In the latter case, a single-click action automatically integrates the generated CTL properties into the design model.

In addition, we propose a set of complementary contributions that integrate this CTL property generation process into MBSE processes. Starting from an RD and a design model, our contributions constitute a framework that operates in four stages.

1. A list of requirements is extracted from the RD (*contribution of this paper*).

2. A set of CTL properties is generated using the generation process (*contribution of this paper*) and refined by the engineer,

3. The CTL properties are verified by model checking on the design model (using TTool's model checker [16]),

4. A coverage analysis of the requirements contained in the RD is performed based on model checking results and back-traced to the RD (*contribution of this paper*). This provides a visual means of identifying requirements not covered by the design, as well as understanding the path along $\mathcal{R}$ that leads to partial coverage (or lack of coverage) of the composite or refined requirements.

We detail these complementary contributions in the following.

C. Requirements Extraction

Given an RD $\langle Req, \mathcal{R}, \mathsf{type}\rangle$, this stage consists of extracting a set $Req_{trad} \subseteq Req$ that serves as input to the CTL property generation process. The construction of this set is left to the user's discretion, using a new GUI-based selection mechanism implemented in TTool. In general, if the requirement diagram contains composite requirements, Req_{trad} will not include these requirements, as they are semantically covered by their sub-requirements, whose translation into CTL properties may likely be easier due to their narrower scope and potentially lower level of abstraction. Therefore, the user will construct Req_{trad} as a subset of $Req \setminus Im(\mathcal{R}_{compose})$.

D. Coverage Computation

This section introduces the logic formulas used to compute requirement coverage in the RD based on the model checking results of the generated CTL properties. For each requirement, two coverage values are computed: a complete coverage score and a partial coverage score, each taking a value in $\mathbb{L}_3$. The main idea is that coverage values are obtained by propagating model checking results along refinement and composition relations. Reading the formulas in full detail is not essential to understand the contribution.

The CTL property generation process outputs a design model enriched with a set of CTL formulas. Mathematically, this output is defined as follows.

Definition 4: Enriched design model
Given a design model $design$, an RD $\langle Req, \mathcal{R}, \mathsf{type}\rangle$ and an extracted requirement set $Req_{trad} \subseteq Req$, an *enriched design model* is a triplet $\langle design, prop, \bowtie\rangle$ where:

- $prop$ is a set of CTL formulas derived from $design$ and Req_{trad} using the process defined above.
- $\bowtie \subseteq Req_{trad} \times prop$ is the relation that associates requirements with CTL properties such that $r \bowtie p$ iff p was generated from r in the generation process.

For $r \in Req_{trad}$, $prop_r$ denotes the set $\{p \in prop \mid r \bowtie p\}$.

Once the enriched design model is obtained, model checking can be performed: the design model $design$ is checked against each property p in $prop$ to evaluate $design \models p$. The result can take one of the three values of $\mathbb{L}_3$: $\top$ if p is satisfied, $\bot$ if p is not satisfied, and $\mathbb{U}$ if the computation fails to complete. The coverage of each requirement by the model $design$ is then computed on the basis of the model checking results. Depending on the verification status of its associated properties, a requirement can be fully covered, partially covered, uncovered, or have unknown coverage. Beyond model checking, the coverage of a requirement may also derive from the coverage of other requirements linked to it through the *compose* and refine *relations*. We have chosen to disable coverage propagation through the *derive* relation, as its semantics tend to allow a broad range of interpretations among requirements engineers. Definition 5 details the computation of coverage values.

Definition 5: Coverage
Given an RD $\langle Req, \mathcal{R}, \mathsf{type}\rangle$, an extracted requirement set $Req_{trad} \subseteq Req$ and a related enriched design model $\langle design, prop, \bowtie\rangle$, the *coverage functions* are two functions $completeCov, partialCov : Req \to \mathbb{L}_3$ defined as follows[1]:

1) general case ($r \notin Im(\mathcal{R}_{compose}) \land r \notin Im(\mathcal{R}_{refine})$):

$$completeCov(r) = \begin{cases} \bigwedge\limits_{p \in prop_r} design \models p & \text{if } prop_r \neq \varnothing, \\ \mathbb{U} & \text{otherwise.} \end{cases}$$

[1]In these formulas, $\land$ and $\lor$ are evaluated according to Kleene's logic.

$$partialCov(r) = \begin{cases} \bigvee_{p \in prop_r} design \models p & \text{if } prop_r \neq \varnothing, \\ \mathbb{U} & \text{otherwise.} \end{cases}$$

> A requirement that is neither composite nor refined is completely covered if all its properties are verified, and partially if at least one is.

2) if $r \notin Im(\mathcal{R}_{compose}) \land r \in Im(\mathcal{R}_{refine})$:

$$completeCov(r) = \begin{cases} \bigwedge_{p \in prop_r} design \models p \land \\ \Big(\bigwedge_{r' \in \mathcal{R}^{-1}_{refine}(r)} completeCov(r') \Big) \\ \text{if } prop_r \neq \varnothing, \\ \mathbb{U} \text{ otherwise.} \end{cases}$$

$$partialCov(r) = \begin{cases} \Big(\bigvee_{p \in prop_r} design \models p \Big) \lor \\ \Big(\bigvee_{r' \in \mathcal{R}^{-1}_{refine}(r)} partialCov(r') \Big) \\ \text{if } prop_r \neq \varnothing, \\ \Big(\bigvee_{r' \in \mathcal{R}^{-1}_{refine}(r)} partialCov(r') \Big) \\ \text{otherwise.} \end{cases}$$

> A requirement that is refined but not composite is completely covered if all its properties are verified and all its refining requirements are completely covered, and partially if at least one of its properties is verified, or if at least one of its refining requirements is partially covered.

3) if $r \in Im(\mathcal{R}_{compose}) \land r \notin Im(\mathcal{R}_{refine})$:
$completeCov(r) = \bigwedge_{r' \in \mathcal{R}^{-1}_{compose}(r)} completeCov(r')$
$partialCov(r) = \bigvee_{r' \in \mathcal{R}^{-1}_{compose}(r)} partialCov(r')$

> A requirement that is composite but not refined is completely covered if all its sub-requirements are completely covered, and partially covered if at least one of its sub-requirements is partially covered.

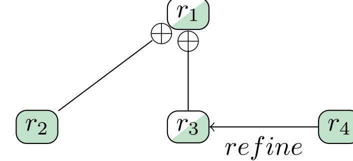
4) if $r \in Im(\mathcal{R}_{compose}) \land r \in Im(\mathcal{R}_{refine})$:
$completeCov(r) = \bigwedge_{r' \in \mathcal{R}^{-1}_{compose}(r)} completeCov(r') \land \Big(\bigwedge_{r' \in \mathcal{R}^{-1}_{refine}(r)} completeCov(r') \Big)$
$partialCov(r) = \Big(\bigvee_{r' \in \mathcal{R}^{-1}_{compose}(r)} partialCov(r') \Big) \lor \Big(\bigvee_{r' \in \mathcal{R}^{-1}_{refine}(r)} partialCov(r') \Big)$.

> A requirement that is both composite and refined is completely covered if all its sub-requirements and refining requirements are completely covered, and partially covered if at least one of its sub-requirements or deriving requirements is partially covered.

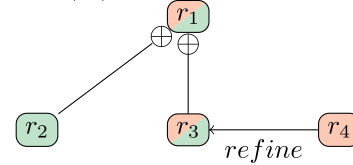
Example 2: Coverage computation in practice
In the diagrams below, each requirement node r is divided into two triangles: the upper triangle represents $completeCov(r)$, and the lower triangle $partialCov(r)$. The following color code is used: green for $\top$, white for $\mathbb{U}$, and red for $\bot$.

Consider the RD introduced in Example 1 and a related enriched design model $\langle design, prop_{r_2} \cup prop_{r_4}, \bowtie \rangle$.

First case: consider that after model checking, $\forall p \in prop_{r_2} \cup prop_{r_4}, design \models p$. As a consequence, $completeCov(\{r_2, r_4\}) = \{\top\}$. $prop_{r_3} = \varnothing$, therefore $completeCov(r_3) = \mathbb{U}$. However, since r_4 refines r_3, $partialCov(r_3) = \top$. r_1 is composed by r_2, which is completely covered, and r_3, which is only partially covered: thus $completeCov(r_1) = \mathbb{U}$ and $partialCov(r_1) = \top$.



Second case: consider now that the enriched design model is $\langle design, prop_{r_2} \cup prop_{r_3} \cup prop_{r_4}, \bowtie \rangle$ with $prop_{r_3} \neq \varnothing$, and consider that after model checking: $\forall p \in prop_{r_2}, design \models p$ and $\exists p \in prop_{r_3} : design \models p$ and $\forall p \in prop_{r_4}, \neg(design \models p)$. Therefore, $completeCov(r_2) = \top$, $completeCov(\{r_3, r_4\}) = \bot$ and $partialCov(r_3) = \top$ (since at least one of its associated properties is verified). In that case, $completeCov(r_1) = \bot$ and $partialCov(r_1) = \top$.



Characterizing Undefinedness: since a model checking result can take the value $\mathbb{U}$, the preimage of $\mathbb{U}$ under $completeCov$ and $partialCov$ contains requirements for which no CTL property is associated via $\bowtie$, or for which the model checking of one or more CTL properties associated via $\bowtie$ has failed. To distinguish between these two cases, we introduce the following function:

$$modelCheckingCompleted : \\ \{r \in Req \mid prop_r \neq \varnothing\} \to \mathbb{B} \\ r \mapsto \bigwedge_{p \in prop_r} (design \models p \neq \mathbb{U})$$

The result of this function is used, in our implementation, to distinguish the following coverage values for each requirement: *completely covered*, *partially covered*, and two sub-cases

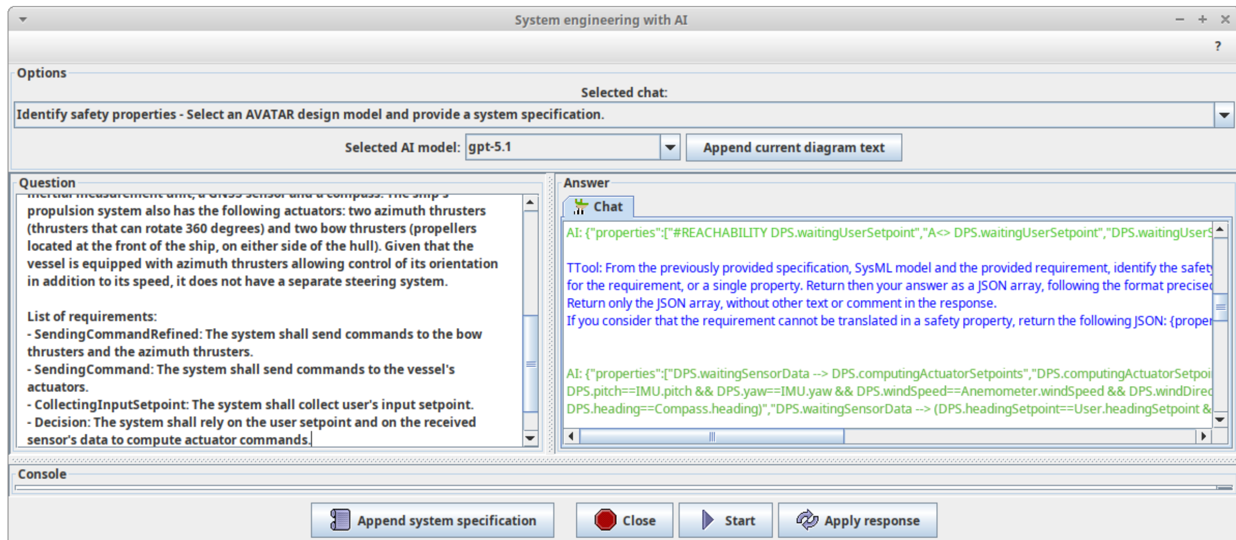


Fig. 2. TTool-AI window during the property generation process

of undefinedness, namely *not verified* (when no property is associated with the requirement) and *coverage unknown* (when the model checking result is $\mathbb{U}$).

V. Practical Insights

This section provides a practical insight of our contributions as implemented in TTool. It first presents a step-by-step walkthrough from a systems engineer's usage perspective, and then discusses the contributions based on two distinct case studies.

A. Reference Systems for Evaluation

Our contributions have been evaluated on two distinct systems. The first is a ship's dynamic positioning system (DPS). A DPS is a controller that regulates a vessel's position and heading according to setpoints provided by the crew. To achieve this, it relies on sensors measuring the vessel's position, environmental parameters (wind force and direction, sea state, etc.), and platform motions (roll, pitch, yaw, surge and sway). It then acts on actuators (thrusters, rudders, etc.) to maintain the vessel's position. The second system is a satellite fault-tolerant telemetry system, whose specification comes from the AQUAS H2020 project[2]. This system operates on a satellite and processes telecommands sent by a ground station. In response, it constructs and sends an encrypted telemetry message containing information about the satellite's status (position, temperature, battery level, and fuel quantity). The system also relies on a periodic daemon to monitor the state of other satellite subsystems and on error-correcting codes to detect bit-flips in the satellite's memory blocks.

Companion material: the systems specifications, our models, the generated properties, the full evaluation results and guidance on how to reproduce them are available on a public GitHub repository[3].

B. Step-by-Step Walkthrough

As our framework takes a design model and an RD as input, the first step is to create (or load) these two models in TTool. The DPS serves as the reference system for this walkthrough: we modeled it using a design model composed of eight blocks and state machines. The RD contains five requirements, as illustrated in Figure 3.

Next, TTool-AI's CTL property generation feature is launched. The framework automatically fills the query field with the system specification and the textual requirements extracted from the RD. The automated interaction between the framework and the LLM is then triggered. Figure 2 shows the TTool-AI window, displaying on the left part of the specification and the textual requirements added automatically, and on the right the CTL properties generated by the LLM. Once the iterative exchanges between TTool-AI and the LLM have converged to a list of syntactically correct CTL properties, they are integrated into both the RD and the design model (in the RD, each property is linked to the requirements it is associated with via the $\bowtie$ relation, see Figure 3).

The user can then manually refine the generated properties or complement them with additional ones, depending on their assessment of how well the properties capture the semantics of the requirements. For instance, in one of the evaluations performed for the needs of this paper, the LLM generated the following property: *DPS.sendingActuatorSetpoints --> (DPS.headingSetpoint == User.headingSetpoint && DPS.posXSetpoint == User.posXSetpoint && DPS.posYSetpoint == User.posYSetpoint)* for the requirement "*The system shall rely on the user setpoint and on the received*

[2] https://aquas-project.eu/

[3] https://github.com/ZebreDeSoixanteQuatorzeCanons/CTLPropertyGeneration

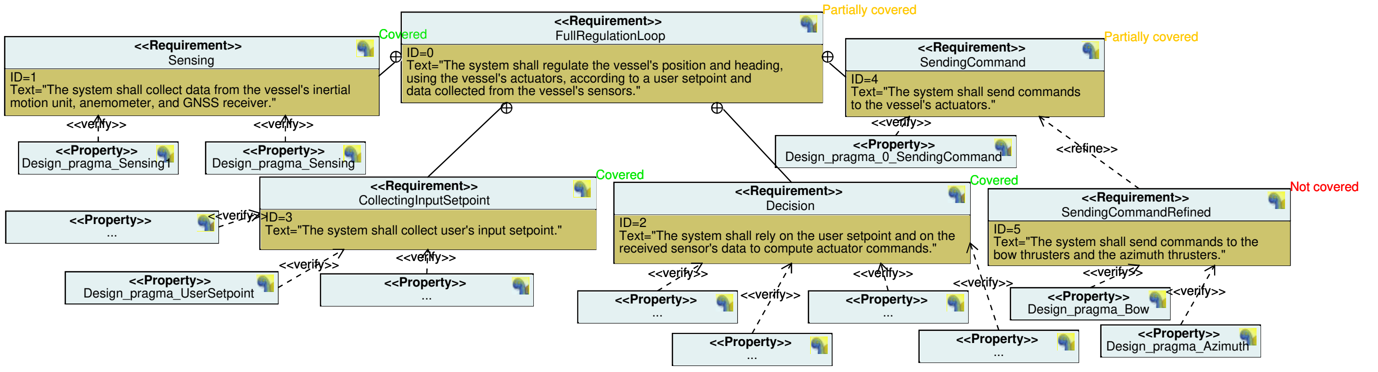


Fig. 3. RD of the DPS, following model checking of a design model in which design faults were injected

TABLE I
EVALUATION METRICS

(a) Dynamic Positioning System

Before manual refinement			After manual refinement		
	Average	Min–Max		Average	Min–Max
Generation time (s)	29	11.5–40.3	Nb. of modified properties	0.8	0–2
Nb. of generated properties	12.4	8–21	Nb. of deleted properties	2.6	2–4
Nb. of relevant properties	9.2	5–19	Nb. of added properties	4.6	4–6
Rate of relevant properties (%)	70.1	55.6–90.5	Nb. of properties after modification	14.6	12–23
Nb. of associated requirements	4/5	3–5/5	Refinement time (min)	7 min 37 s	6 min 32 s – 8 min 23 s
			Quality score /10	5.3	3.7–7.6

(b) Satellite System

Before manual refinement			After manual refinement		
	Average	Min–Max		Average	Min–Max
Generation time (s)	51.3	30–85.1	Nb. of modified properties	0.7	0–2
Nb. of generated properties	8.7	5–11	Nb. of deleted properties	0.7	0–1
Nb. of relevant properties	7	2–10	Nb. of added properties	2	1–4
Rate of relevant properties (%)	73.6	40–90.9	Nb. of properties after modification	10	8–12
Nb. of associated requirements	3.2/5	1–4.5/5	Refinement time (min)	6 min 36	5 min – 7 min 42 s
			Quality score /10	6.9	3.3–9.2

sensor's data to compute actuator commands." The LLM made here an error on the meaning of the "leads to" (-->) operator: the property has to be replaced, in order to match the intended semantics, by *A[] (!(DPS.sendingActuatorSetpoints) or (DPS.headingSetpoint == User.headingSetpoint && DPS.posXSetpoint == User.posXSetpoint && DPS.posYSetpoint == User.posYSetpoint))*.

Once this step is completed, on the user's request, the framework performs model checking and coverage computation according to Definition 5. The values of $completeCov$ and $partialCov$ are then propagated back to the RD (see Figure 3).

C. Evaluation Metrics and Discussion

Table I summarizes some evaluation metrics for the two case studies. For the DPS, five successive iterations of the walkthrough presented above were carried out, using GPT-5.1 as the underlying LLM. For the satellite system, three iterations were performed, using three distinct LLMs: GPT-5.1, Magistral-small-2509, and Qwen3-coder-30b. For each iteration, a quality score was computed by assigning each property of the set *AI-generated properties* ∪ *engineer-added properties* a value out of 10: 0 if an AI-generated property was deleted, 5 if modified, 10 if kept unchanged, and 0 for engineer-added properties.

Overall, the evaluation indicates that the framework effectively meets its main objective: helping systems engineers derive CTL properties from a SysML design model and its associated RD. On average, across the two case studies, approximately 70% of the properties generated by the AI were semantically relevant and therefore retained in the final property set. Some of these accurate properties were highly complex (*e.g.*, a single formula containing a negation, a disjunction, and ten conjunctions). Across the two models, the average time required for a human to analyze and refine the suggested properties to obtain a set that fully captures the intended meaning of the requirements was about 7 minutes and 7 seconds, while the initial property generation took 40.2 seconds. We also note that, as expected by construction, 100% of the properties generated were syntactically correct. Together, these metrics support the usefulness of the approach in assisting systems engineers with the CTL property design task.

However, the evaluation also makes clear that the contribution cannot be used as an unsupervised tool to generate

CTL properties that fully and accurately capture the semantics of the requirements. In most iterations, some requirements were not associated with any property; in almost all iterations, some generated properties had to be deleted; and in all cases, additional properties had to be created to achieve complete semantic coverage of the requirements. Nevertheless, in certain instances, the results prior to human refinement were excellent, highlighting the tool's potential as a powerful assistant. For one of the DPS iterations, 90.5% of the generated properties were relevant, and after manual refinement, 83% of the final property set originated from the tool.

We should also note an important observation that emerged during the evaluation: the AI-based generation process produces better results when provided with a complete design model than when the input model contains faults. This is expected, as faulty design models often lack states, attributes, or even blocks that are necessary to capture the full meaning of the requirements. The same challenge occurs when engineers attempt to elaborate such properties manually while examining the model: this is, moreover, a good way to realize that the design is incomplete. This further reinforces that the tool should be used as an assistant that automates the process, rather than as a standalone generator that makes the process fully automatic.

Finally, although the metrics presented here provide a good basis for an initial discussion of these contributions, we must highlight two limitations of the evaluation: first, it was conducted over a limited number of repetitions of the process (eight in total); second, we did not measure reference times for generating the properties entirely manually, without the assistance of the tool, and therefore we have no basis for comparison regarding these times.

VI. Conclusions

Motivated by the need to support systems engineers in generating properties for the formal verification of requirements within the MBSE context, this paper proposes a new engineering assistance framework. The framework features an AI- and syntactic-rule-based property generation process that translates requirements into syntactically correct CTL properties, along with complementary functionalities that enable automatic extraction of requirements from an RD, as well as requirement coverage computation and backtracing from checking results. These features ensure efficient integration of the framework within the MBSE workflow. While the focus of this work is on SysML, the approach is applicable to other modeling formalisms, provided that (1) they support model checking of temporal logic properties, and (2) a requirements structure supporting composition and refinement relations is supplied as input.

All contributions have been implemented in TTool and the initial evaluation results indicate the relevance and potential of the approach. However, further evaluations are required, including comparative studies of CTL property design with and without the framework. Future work also aims to extend the framework to other types of properties, such as security and performance properties. Ultimately, the goal is to combine this property-generation capability with model checking within LLM-based SysML model generation frameworks, such as TTool-AI, to achieve semantic correctness by construction for AI-generated models and to make AI-based engineering assistants more reliable and effective for systems engineers.

References

[1] Henderson, K. , Salado, A., "Value and benefits of model-based systems engineering (MBSE): Evidence from the literature," Systems Engineering, 2021, 24(1):51–66.

[2] de Saqui-Sannes, P., Vingerhoeds, R. A., Garion, C. , Thirioux, X. "A taxonomy of MBSE approaches by languages, tools and methods," IEEE Access, 2022, 10:120936–120950.

[3] de Saqui-Sannes, P. , Apvrille, L., Vingerhoeds, R. A., "Checking SysML models against safety and security properties," Journal of Aerospace Information Systems, 2021, 18(12):1–13.

[4] Cimatti, A., Roveri, M., Susi, A., Tonetta. S., Fantechi, A., "From informal requirements to property-driven formal validation," Formal Methods for Industrial Critical Systems, 2009, Springer Berlin Heidelberg, pp. 166–181.

[5] Coudert, S., Apvrille, L. , Sultan, B. , Hotescu, O. , de Saqui-Sannes, P.,"Incremental and Formal Verification of SysML Models," SN COMPUT. SCI. 5, 714 (2024).

[6] Kausch, H., Pfeiffer, M., Raco, D. et al., "Model-driven development for functional correctness of avionics systems: a verification framework for SysML specifications," CEAS Aeronaut J 16, 33–48 (2025).

[7] Dehn, S., Jacobs, G., Höck, P. et al. , "Requirements: integrating temporal logic and SysML v2 for comprehensive state and transition modeling," Forsch Ingenieurwes 89, 53 (2025).

[8] OMG Systems Modeling Language, "Object Management Group," 2017, https://www.omg.org/spec/SysML/1.5.

[9] Rogin, F., Klotz, T., Fey, G., Drechsler, R., Rulke, S., "Advanced verification by automatic property generation. Computers & Digital Techniques," IET. 3. 338 - 353. 10.1049/iet-cdt.2008.0110. (2009).

[10] Ankireddy, D., Paria, S., Dasgupta A., Ray, S., Bhunia, S."LASA: Enhancing SoC Security Verification with LLM-Aided Property Generation". (2025).

[11] Bruel, J.-M., Ebersold, S., Galinier, F., Mazzara, M., Naumchev, A. and Meyer, B., "The Role of Formalism in System Requirements," ACM Comput. Surv. 54, 5, Article 93 (June 2022), 36 pages. (2021).

[12] Lienhardt, M., "Applied Formal Methods at ONERA: An Experience Report, "In: Mezzina, C.A., Schmitt, A. (eds) Components Operationally: Reversibility and System Engineering. Lecture Notes in Computer Science, vol 16065. Springer.(2026).

[13] Gao, B. D., Miao, H., Yang, X., Duran Barroso, R. J., Walayat, H., "A novel gapg approach to automatic property generation for formal verification: The gan perspective," ACM Trans. Multimedia Comput. Commun. Appl. 19, 1 (2023), 1–22.

[14] Wu, N., Li, Y., Yang, H., Chen, H., Dai, S., Hao, C., Yu, C., Xie, Y., "Survey of Machine Learning for Software-assisted Hardware Design Verification: Past, Present, and Prospect," ACM Trans. Des. Autom. Electron. Syst. 29, 4, Article 59 (July 2024), 42 pages.

[15] Dhaussy, P., Pillain, P.-Y., Creff, S., Raji, A., Le Traon, Y, Baudry, B., "Context and Requirement Formalization for Embedded Software Validation" (In French), 2012.

[16] Tempia Calvino, A., Apvrille, L., "Direct model checking of SysML models. In : 9th International Conference on Model-Driven Engineering and Software Development," SCITEPRESS-Science and Technology Publications, 2021. p. 216-223.

[17] Sultan, B., Apvrille, L., "TTool-AI: A Large Language Model-Based Assistant for Model Driven Engineering," SN COMPUT. SCI. 6, 886 (2025).

[18] Sultan, B., Apvrille, L., "AI-Driven Consistency of SysML Diagrams," In Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24). Association for Computing Machinery, New York, NY, USA, 149–159. (2024).

[19] Pedroza, G and Apvrille, L and Knorreck, D., "AVATAR: A SysML environment for the formal verification of safety and security properties," In 11th Annual International Conference on New Technologies of Distributed Systems, IEEE, 1–10. (2011).