Towards an Agentic LLM-based Approach to Requirement Formalization from Unstructured Specifications

Alberto Tagliaferro[1], Bruno Guindani[1], Livia Lestingi[1], and Matteo Rossi[1]

Politecnico di Milano, Milan, Italy
name.surname@polimi.it

Abstract

Early-stage specifications of safety-critical systems are typically expressed in natural language, making it difficult to derive formal properties suitable for verification and needed to guarantee safety. While recent Large Language Model (LLM)–based approaches can generate formal artifacts from text, they mainly focus on syntactic correctness and do not ensure semantic alignment between informal requirements and formally verifiable properties. We propose an agentic methodology that automatically extracts verification-ready properties from unstructured specifications. The modular pipeline combines requirement extraction, compatibility filtering with respect to a target formalism, and translation into formal properties. Experimental results across three scenarios show that the pipeline generates syntactically and semantically aligned formal properties with a 77.8% accuracy. By explicitly accounting for modeling and verification constraints, the approach is a paving step towards exploiting Artificial Intelligence (AI) to bridge the gap between informal descriptions and semantically meaningful formal verification.

Keywords

Formal Property Elicitation, Requirement Formalization, AI for Requirements Engineering, AI for Formal Methods, Large Language Models, Agentic Workflow

1 Introduction

When engineering software-intensive safety-critical systems, providing sound guarantees that the delivered product satisfies set requirements is of pivotal importance. For example, for service robots requiring close interaction with humans to complete their tasks, it is crucial to ensure that their behavior complies with both safety and societal norms. Model-driven approaches in combination with formal verification techniques are traditionally adopted to guarantee that the final software-controlled system will satisfy the desired properties. To this end, a formal specification of the system under analysis, amenable to verification, and of the properties to be satisfied must be available. However, at an early stage, the software development process is usually carried out by means of *informal* specifications, such as natural language descriptions of the Human-Robot Interaction (HRI) scenarios that the robotic agent must be able to tackle, and eliciting verifiable requirements from free text remains an open challenge.

The existing body of work on requirement elicitation widely explores the issue of extrapolating meaningful requirements from unstructured text with an automated process [22, 17]. On the other hand, the goal of automatically *formalizing* requirements out of natural language descriptions has only recently been pursued with the advent of generative Artificial Intelligence (AI) and, specifically, Large Language Models (LLMs). However, recent work mostly focuses on

approaches that ensure the *syntactic* correctness of the AI-generated artifacts. Syntactic correctness can be checked automatically by validating models against a grammar, compiling them, or verifying well-formedness constraints defined by modeling guidelines. While these checks ensure structural consistency, they do not guarantee that the generated requirements also *semantically* align with the intended system's behavior. Verifying semantic correctness is substantially more difficult: properties must reflect the intent expressed in natural language specifications, be expressible in a formal language, and be verifiable on the resulting model.

This paper addresses this challenge by proposing a methodology that automatically extracts a set of formal properties from an informal specification, with the goal of operationalizing requirements that are not only syntactically but also semantically correct with respect to the initial specification. The approach explicitly accounts for the characteristics and constraints of the target model, ensuring that the generated properties are meaningful and verifiable. Thanks to the modular structure of the methodology, it is possible to trace and inspect each stage of the property elicitation process.

The presented methodology proposes an agentic approach based on LLMs to elicit verification-ready properties from unstructured textual specifications. The approach is *agentic* as it coordinates multiple LLMs with specialized roles and integrates verification tools within the pipeline, allowing candidate properties to be extracted, assessed, and translated while being validated against the constraints of the target verification framework. Specifically, the pipeline envisages the following stages: (i) an LLM extracts candidate natural language requirements from informal specifications, (ii) candidate requirements are then filtered by a judge LLM to exclude the ones corresponding to properties that are not compatible with or cannot be expressed in the target formal language, (iii) eligible candidates are translated into properties expressed in the target formalism, and, (iv) finally, the syntactic correctness of the generated properties is programmatically checked through the target verification tool while a second judge LLM evaluates the semantic alignment with the original specification.

While the approach is domain-agnostic, our preliminary evaluation focuses on informal specifications in the HRI domain. Each specification is translated into a Language for Interactive Agents (LIrAs) specification, a Domain-Specific Language (DSL) designed for describing multi-agent interaction patterns [23, 24], which automatically generates a Stochastic Hybrid Automaton (SHA) model compatible with the Uppaal tool.[1] Consequently, the target formalism for this work is the query language provided by Uppaal in support of Statistical Model Checking (SMC), but as mentioned, the proposed pipeline can be applied to different domains, formalisms, and verification tools. Experiments are performed with the *gemini-3.1-pro-preview* model.[2] Results show that: (i) in the extraction phase, the model achieves an 81.8% combined rate of exact and partial semantic matches for the generated requirements; (ii) in the verifiability classification stage, it has an overall accuracy of 88.7% and a recall of 94.2%, successfully filtering out unverifiable constraints; and (iii) in the final formal translation step, 95.8% of the generated queries are syntactically correct, with a 77.8% effective semantic translation accuracy when accounting for valid logical equivalences and alternative formulations.

This paper is structured as follows: Section 2 surveys related work, Section 3 outlines preliminary concepts, Section 4 describes the proposed methodology, Section 5 presents experimental results, and Section 6 concludes and presents future research directions.

[1] www.uppaal.org

[2] The code and artifacts for this project are publicly available at doi.org/10.5281/zenodo.19023550.

2 Related work

This section surveys related work on requirement formalization from informal specifications and the use of LLMs in requirements engineering, positioning our work with respect to the state-of-the-art.

Requirement Formalization from Informal Specifications. Beg et al. [4] survey (semi-)automated approaches for generating formal specifications from Natural Language (NL) requirements, including NL, ontology-based domain modeling, and LLMs, and discuss key challenges identified across the literature. These include semantic ambiguity, the absence of ground-truth datasets, limited tool interoperability, traceability across artifacts, and concerns regarding explainability and user trust.

Recent work increasingly leverages LLMs for direct translation into temporal logics. For instance, Li et al. [16] generate Temporal Logic (TL) specifications from software documentation using both end-to-end and two-step pipelines, reporting issues of specification oversimplification and fabrication. Cosler et al. [9] introduce nl2spec, an open-source framework that translates unstructured NL requirements into TL statements via LLMs, while Zhao et al. [30] focus specifically on automated conversion into Computation Tree Logic (CTL).

Several approaches introduce intermediate representations to structure the translation process. In Ma et al. [19], LLMs map informal requirements to a JSON-like language, OnionL, which is subsequently translated into Linear Temporal Logic (LTL) through rule-based techniques. Similarly, Guidotti et al. [14] convert NL requirements into Property Specification Patterns (PSP), an intermediate formalism enabling translation into multiple target logics.

Beyond temporal logic generation, Wen et al. [26] and Faria et al. [11] employ LLM-based techniques to synthesize pre- and post-condition annotations in various programming languages. Related research also explores the generation of formal theorem statements and proofs, notably the autoformalization work of Wu et al. [27]. However, while these approaches address the correctness or plausibility of generated formal artifacts, they do not explicitly evaluate their consistency with the originating informal specification.

Compared to rule-based requirement extraction or Natural Language Processing (NLP) pipelines designed to derive verifiable properties, LLM-based approaches offer greater flexibility and portability. Traditional methods often require substantial customization for each domain, modeling language, or verification framework. In contrast, the proposed methodology can be adapted with minimal effort: changes typically involve revising prompt instructions and textual examples provided to the LLM, rather than redesigning extraction rules or retraining specialized models. This makes the pipeline a practical and extensible solution for bridging the gap between informal system descriptions and formal, semantically meaningful verification properties.

LLM-Assisted Requirements Engineering. The usage of LLMs to assist software engineering is a relevant subject in the state of the art as well as the industry [15]. Consistency checking has long been a central topic in requirements engineering (see, e.g., Yan et al. [28]), but LLMs have recently expanded the methodological landscape. The survey by Zadenoori et al. [29] shows that "most of the studies focus on using LLMs for requirements elicitation and validation, rather than defect detection and classification", where *validation* denotes assessing whether "requirements accurately reflect stakeholder intent". However, validation in this sense does not typically include verifying that derived requirements remain coherent with a given textual system specification.

Work on "defect detection and classification" (as categorized by Zadenoori et al. [29]) likewise omits this perspective. These studies focus primarily on inconsistencies *between* requirements, logical flaws within individual requirements, or violations of writing best practices.

For example, Fazelnia et al. [12] address *i)* classification into categories such as security, performance, and portability, *ii)* detection of defects such as ambiguity or non-compliance with requirements-writing guidelines, and *iii)* identification of inter-requirement conflicts. Similarly, Mahbub et al. [20] examine ambiguity, inter-requirement inconsistencies, and incompleteness in large industrial requirement sets, while Luitel et al. [18] use LLMs as external knowledge sources to detect incompleteness in NL requirements. Bertram et al. [5] focus on pattern classification and translation into Structured English with stricter grammatical constraints. Finally, Gärtner and Göhlich [13] and Chen et al. [8] investigate contradiction detection *between* requirements, the former by combining formal reasoning decision tree-based frameworks with LLMs and the latter by generating imperative SMT-based code checkers.

While existing approaches emphasize syntactic correctness, formal well-formedness, or successful translation into verification artifacts, they provide limited support for assessing whether the generated formal properties faithfully capture the intent of the original informal specification. Explicitly evaluating this semantic consistency is the central objective of our work.

3 Preliminaries

In the following, we outline preliminary concepts underlying our work, i.e., LLMs, few-shot learning, and Uppaal's SMC query language.

LLMs and Few-shot Learning. An LLM is an autoregressive and probabilistic generative model for text trained on massive amounts of data. LLMs are built upon the *Transformer* architecture that processes sequences of text tokens through the *self-attention* mechanism [25]. As autoregressive models, LLMs work by predicting, one token at a time, the most likely new token based on all the context at hand and all the previously generated tokens.

When *prompted* with some text, LLMs generate a completion (i.e., the response) for the given prompt. To explicitly steer this generation, a *system prompt* is often employed to define an overarching persona, specify operational boundaries, and dictate the general rules the LLM must follow throughout the interaction. Furthermore, several approaches have been developed to structure the prompt (also referred to as prompting *strategies*): *few-shot learning* (sometimes called in-context learning [6]) is among the most commonly adopted strategies. Few-shot learning envisages incorporating into the prompt examples of input-output pairs (i.e., the *shots*) together with the task instructions. This helps the LLM identify patterns in the text and clarifies the expected input/output formats [6].

Uppaal SMC. Stochastic formal models are amenable to SMC [1], which can be performed through Uppaal [10]. SMC applies statistical techniques to a set of *runs* of a formal model M with stochastic features to estimate the *probability* of the desired property holding. Uppaal specifically computes the value of expression $\mathbb{P}_M(\psi)$ to estimate the probability of property ψ holding for model M [10]. Property ψ is expressed in Metric Interval Temporal Logic (MITL) [2], a linear temporal logic interpreted over timed state sequences that allows users to capture real-time constraints. In particular, MITL includes operator $\Diamond_{[a,b]}$ (with $a < b$), where formula $\Diamond_{[a,b]}\phi$ means that property ϕ holds at a point in the future (i.e., "eventually") that is no less than a and no more than b time units from the current one. Specifically, ψ in expression $\mathbb{P}_M(\psi)$ is of the form $\Diamond_{\leq\tau}$ `ap`, where $\Diamond_{\leq\tau}$ is an abbreviation for $\Diamond_{[0,\tau]}$ and `ap` $\in$ AP is an atomic proposition. Formula $\Diamond_{\leq\tau}$ `ap` is true in the first instant of a run of model M if `ap` holds within τ time units from time instant 0. In the Uppaal SMC query language, $\mathbb{P}_M(\Diamond_{\leq\tau}$ `ap`$)$ is expressed as `P[<=τ](<> ap)`, where `ap` is either an atomic proposition or a conjunction/disjunction thereof.

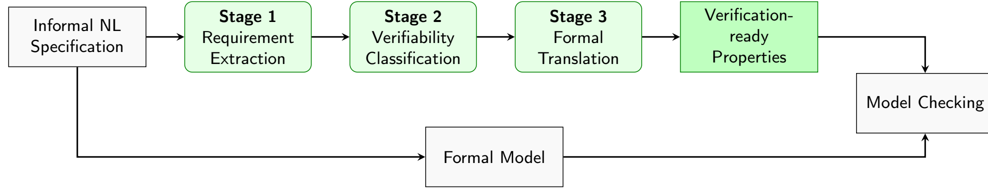

Figure 1: Proposed agentic pipeline. In green, the part explored by this work.

4 Methodology

This section details the proposed agentic pipeline for bridging the gap between informal NL specifications and formal semantically aligned properties. The approach is structured into three distinct stages to ensure traceability, mitigate the overall complexity, and improve the performance of LLMs. The pipeline is illustrated in Figure 1, where **Stage 1**, **Stage 2** and **Stage 3**, corresponds to three single agent LLM-based step.

Stage 1: Requirements Extraction. The first stage transforms unstructured informal specifications into a structured set of atomic natural-language requirements, each one associated with an identifier, isolating functional and safety constraints embedded in the system description, while using the standard requirements syntax (e.g., "The system must...", "The human agent has to...", "The robot shall...").

An LLM is used to identify relevant entities, system dynamics, and operational constraints described in the specification. The extracted requirements are normalized and structured into a machine-readable representation (i.e., JSON format), enabling their systematic processing in the subsequent stages of the pipeline.

Stage 2: Verifiability Classification. The second stage filters requirements that cannot be expressed within the semantic boundaries of the target formal model. Indeed, not all elicited requirements can be formally verified within the constraints of the considered model. To perform this classification, we use an LLM, as its adaptability across different domains facilitates reuse; moreover, due to its nature, it is capable of performing classification tasks based on semantic evaluation. Thanks to this second classification stage, in the first one, the LLM focuses solely on generating a complete set of requirements without bias that can be derived by directly considering the limitations of the specified formal model.

In this stage, each requirement is analyzed with respect to the modelling assumptions and observable variables of the system. Requirements referring to unobservable properties, qualitative design intentions, or model-external phenomena are discarded. The remaining requirements constitute the subset that can be formally analyzed.

Stage 3: Formal Translation. The final stage translates the filtered requirements into formal verification queries compatible with the target formal model. The objective of this stage is to bridge the gap between natural language specifications, which are inherently ambiguous, and formal verification artifacts.

Thanks to the ability of LLMs to generate text according to a precise grammar (provided in BNF format in the system prompt), each requirement is translated into a corresponding temporal property that can be evaluated on the system model.

The resulting queries represent formalized constraints that can be directly analyzed through model checking, providing a formalization of the otherwise ambiguous semantics inherent in natural language requirements.

Table 1: Overview of the dataset size. *Ag.* denotes the number of agents defined in the LIrAs code, *Loc.* denotes the number of locations, and *Res.* denotes the number of resources. *Req.* stands for requirements, and *Ver.* stands for verifiable.

Scenario	Ag.	Loc.	Res.	GT Req.	GT Ver. Req.	GT Queries
SC1	1	3	1	19	12	15
SC2	2	2	1	26	22	23
SC3	2	6	3	26	18	34
Total	**-**	**-**	**-**	**71**	**52**	**72**

5 Experimental Evaluation

In this section, we evaluate each stage of the proposed pipeline to assess its effectiveness in translating informal NL specifications into formal verification queries. For this evaluation, we used the LIrAs DSL to generate the formal model corresponding to the informal specification. In this setting, the target formalism is SHA, and verification is performed using Uppaal SMC queries. Our evaluation is designed to answer the following Research Questions (RQs):

RQ1. Requirements Extraction: How accurately can the LLM extract requirements from informal NL specifications?

RQ2. Verifiability Classification: How effectively can the LLM classify the formal verifiability of requirements with respect to strict model boundaries?

RQ3. Formal Translation: To what extent can the LLM translate verifiable requirements into syntactically correct and semantically equivalent Uppaal SMC queries?

5.1 Experimental Setup

The evaluation is conducted on a dataset comprising three distinct cyber-physical system scenarios (SC): *SC1: Coffee_Delivery*, *SC2: User_Guided_Transport*, and *SC3: Factory_Pipeline*, inspired by the literature [21, 3, 7].

These specifications were translated into the LIrAs DSL, from which the corresponding SHA models were automatically generated. Table 1 summarizes the dataset size and associated Ground Truth (GT) information. For example, the first scenario takes as input one agent, three locations, and one resource. From the informal specification, we manually constructed the GT, obtaining 19 NL requirements. Of these, 12 are suitable for formal verification, producing 15 SMC queries (as a requirement may correspond to multiple queries). Non-verifiable requirements arise mainly from model abstractions or technical limitations of SMC.

To ensure a rigorous evaluation, each pipeline stage was executed and analyzed independently using the GT as the benchmark. This decoupled setup measures the performance of each component while preventing cascading errors.

RQ1: Requirements Extraction. In this first stage, a system prompt instructs the LLM to extract atomic requirements from informal specifications. The prompt directs the model to focus on physical dynamics, safety aspects, and agent roles, while constraining the output to a strict *JSON* schema to ensure seamless integration with downstream pipeline steps.

We employ a 2-shot prompting strategy in which, for each scenario, the GTs of the other two scenarios are used as examples. Specifically, the user prompt provides the informal specification,

while the assistant prompt supplies the corresponding hand-crafted requirements. This setup guides the model to correctly structure the generated requirements and format them according to the expected *JSON* output.

To evaluate the quality of this step, we compare each generated set of requirements against the corresponding ground truth. Since the two lists are difficult to compare directly—due to the inherent variability of informal requirements—we adopt an LLM-as-a-judge approach. The judge model is instructed through a system prompt to perform a semantic evaluation, allowing many-to-many matching between generated and ground-truth requirements. It applies the following three classification rules:

- Match: Same intent, scope, and constraints; differences are purely stylistic.
- Partial: The intent is similar, but the scope differs (e.g., more specific or more abstract).
- NoMatch: No GT requirement captures the semantic intent of the generated requirement.

Additionally, the judge provides a confidence score (between 0.0 and 1.0), a short justification for each match, and identifies any GT requirements that were missed.

The judge receives the informal specification (for context) and the two requirement lists (generated and ground truth), operating in a 0-shot setting.

RQ2: Verifiability Classification. In the second stage, the pipeline acts as a formal gatekeeper. We instruct the LLM to assume the role of an *expert systems engineer* specializing in *formal methods* and *SMC*. The task is to evaluate the requirements and determine whether each one is formally verifiable within the strict boundaries of the specific SHA model. To ensure independence from the previous step, we use the GT requirements as input for this stage.

The LLM is provided with the informal specification and the list of requirements to be evaluated. To reduce hallucinations, the system prompt explicitly defines the semantic boundaries of the LIrAs model, including assumptions such as the system being memoryless, the absence of physical damage detection, and the assumption that stationary machines have infinite resources and zero processing delay.

The LLM must strictly apply a set of structured classification rules to filter out unverifiable concepts, such as unobservable variables or qualitative design choices. The expected output is a strict *JSON* array that classifies each requirement as either "Yes" (verifiable) or "No" (not verifiable), together with a concise justification, helping the LLM in the reasoning process. To evaluate this step, we compare the LLM's "Yes/No" predictions with the manually defined GT verifiability labels.

RQ3: Formal Translation. The final step translates verifiable requirements into syntactically correct Uppaal SMC queries that can be verified on the model generated from the LIrAs DSL for the associated specification. The system prompt enforces strict constraints, requiring the generated queries to conform to a compact BNF grammar for SMC. Furthermore, the model is guided by explicit mapping rules that bridge the LIrAs BNF grammar and the corresponding Uppaal model. To evaluate the generated queries, we adopt a rigorous three-level strategy:

1. *Syntax Check:* First, we evaluate the syntactic validity of the queries by passing them through the Uppaal `verifyta` compiler. Any query that fails to compile is marked as a failure, preventing the propagation of invalid syntax.
2. *Exact Match:* For queries that correctly compile, we check for perfect textual equivalence between the generated queries and the GT queries.

Table 2: Distribution of requirement extraction accuracy across the evaluated scenarios.

Scenario	GT Req.	Gen. Req.	Match	Partial	NoMatch	Missed GT
SC1	19	16	7 (43.8)%	7 (43.8)%	2 (1.25)%	6 (31.6)%
SC2	26	15	9 (60.0)%	3 (20.0)%	3 (20.0)%	15 (57.7)%
SC3	26	24	8 (33.3)%	11 (45.8)%	5 (20.8)%	7 (26.9%)
Total	**71**	**55**	**24 (43.6%)**	**21 (38.2%)**	**10 (18.2%)**	**28 (39.4%)**

3. *Semantic Evaluation:* For queries that successfully compile but differ textually from the GT, we employ an LLM-as-a-judge to assess semantic equivalence. The judge is instructed with specific equivalence rules, such as logical commutativity, spatial symmetry in distance functions, and tolerance for minor variations in SMC simulation time bounds. Conversely, it is explicitly instructed to penalize critical mismatches, such as confusing reachability with safety invariants. While some mathematical equivalences could in principle be verified through stricter formal rewriting rules, the LLM is particularly useful in identifying higher-level logical correspondences between queries that arise from different modeling perspectives. For instance, verifying that an agent reaches a target location can be expressed either by directly checking the agent's coordinates or by verifying the completion of the corresponding navigation action. The LLM-based judge allows us to capture such semantically equivalent formulations that are difficult to encode through purely syntactic comparison rules. The judge outputs a Boolean match flag together with a concise justification, enabling the computation of a relaxed, yet formally grounded, semantic accuracy metric.

All experimental evaluations were conducted using the *gemini-3.1-pro-preview* model. To reduce the inherent randomness of the language model, the temperature parameter was set to 0.1. Additionally, the model output was strictly constrained to the *application/json* format to ensure automated parsing and compliance with the provided validation schemas.

5.2 Results

RQ1: Requirements Extraction. To answer RQ1, we evaluate how accurately the pipeline extracts structured atomic requirements from informal specifications by comparing the generated requirements against the manually constructed GT. As shown in Table 2, the model achieved an exact semantic match for 43.6% of the generated requirements, a partial match for 38.2%, and no match for 18.2%. The recall of the captured GT requirements is 60.6%, with 28 out of the total 71 requirements not mapped to any generated requirement.

Across the evaluated scenarios, the generated requirements consistently capture the core operational logic, including basic navigation and overall task sequences (e.g., object transport, synchronization, and deployment). However, the qualitative analysis reveals two recurring failure modes in the extraction process.

First, the LLM frequently omits strict logical boundaries and formal prerequisites present in the GT, such as conditions preventing infinite loops, guarantees of task completion, and mechanisms for failure detection. Notably, *SC2* represents a negative outlier in this regard: the model completely missed fundamental state transitions and physical constraints (e.g., battery charge and human fatigue) that were successfully captured in *SC1* and *SC3*.

Second, the model shows a persistent tendency to introduce unstated but plausible constraints derived from general domain knowledge rather than from the provided text. Examples

Table 3: Confusion matrix for verifiability classification (Yes/No). TP = True Positive, FN = False Negative, FP = False Positive, TN = True Negative.

	Predicted Verifiable	**Predicted Unverifiable**
Actual Verifiable	$TP = 49$	$FN = 3$
Actual Unverifiable	$FP = 5$	$TN = 14$

include explicitly identifying employees in *SC1* and enforcing safe following distances in *SC2*. Furthermore, as observed in *SC3*, the LLM struggles with maintaining the intended level of abstraction, often decomposing high-level transport operations into highly granular manipulation steps. This behavior increases the number of partial matches, as the generated requirements diverge in scope from the GT.

RQ1 summary. The LLM effectively extracts the core functional logic and navigation sequences from informal specifications. However, it struggles with formal rigor, often omitting strict logical constraints and occasionally hallucinating plausible but unmodeled domain behaviors. Overall, the combined rate of exact and partial matches reaches 81.8% respect to the total number of generated requirements.

RQ2: Verifiability Classification. For **RQ2**, we evaluate the ability of the pipeline to filter out unverifiable requirements based on the system's structural constraints. Table 3 summarizes the classification performance. The model achieved an overall accuracy of 88.7% (ranging from 84.2% to 92.3% across the evaluated scenarios), correctly classifying 63 out of 71 requirements. The precision is 90.7%, while the recall reaches 94.2%.

These results highlight the importance and effectiveness of this intermediate gatekeeping step. The high recall indicates that the LLM rarely discards valid, formalizable constraints, thereby preserving the core functional requirements of the system.

Importantly, the model correctly identified and filtered out 14 unverifiable requirements (true negatives). This pre-filtering mechanism is essential for the overall efficiency and robustness of the pipeline. By discarding qualitative goals, unobservable properties, or structurally incompatible requirements early in the process, the system reduces token consumption during the computationally expensive generation of Uppaal queries. More importantly, it prevents downstream failures by ensuring that the Uppaal `verifyta` compiler is not burdened with mathematically inexpressible queries, or unmodelled variables, that would inevitably trigger syntax or compilation errors.

Although the model missed some unverifiable requirements (5 false positives), the high precision (90.74%) keeps this leakage limited. Furthermore, the pipeline architecture is resilient to such minor misclassifications, since any erroneous inclusions at this stage will be detected during the strict syntactic and semantic validation performed in the final translation step.

RQ2 summary. The LLM classifies requirement verifiability with 88.7% accuracy and retains more than 94% of the requirements that correspond to valid formal constraints, with respect to our GT.

RQ3: Formal Query Translation. This question investigates the generation of formal SMC queries. To evaluate this capability, we adopt a three-level evaluation methodology: (i) syntactic validation using the Uppaal verification engine, (ii) strict string matching against the

Table 4: Evaluation of formal translation into UPPAAL SMC queries.

Scenario	Queries	Compiled Syntax	Exact Match	LLM Judged Valid	Accuracy
SC1	15	15 (100%)	8 (53.3%)	3 (42.9%)	**11 (73.3%)**
SC2	23	22 (95.7%)	10 (43.5%)	3 (25%)	**13 (56.5%)**
SC3	34	32 (94.1%)	7 (20.6%)	25 (100%)	**32 (94.1%)**
Total	**72**	**69 (95.8%)**	**25 (34.7%)**	**31 (70.5%)**	**56 (77.8%)**

GT queries, and (iii) semantic equivalence evaluation using an LLM-as-a-judge for syntactically valid but textually divergent queries. The results of these evaluations are reported in Table 4.

Out of the 72 generated queries, 69 (95.8%) successfully compiled in UPPAAL, indicating a strong adherence to both the DSL structure and the UPPAAL BNF grammar. However, only 25 queries achieved an exact textual match with the GT. After the semantic evaluation, the LLM-based judge assessed the remaining syntactically valid queries and identified 31 additional queries as semantically correct. As a result, the overall relaxed accuracy increased to 77.8%.

These results highlight a well-known limitation when evaluating code or formal query generation: strict string matching (exact match) is an inherently limited and overly restrictive metric. While only 34.7% of the generated queries exactly match the GT, this metric does not accurately reflect the model's logical capabilities, since multiple syntactically different expressions can represent the same mathematical semantics.

Our qualitative analysis of the rejected exact matches shows that the LLM frequently produced valid formal variations. For example, the model often applied logical commutativity (e.g., evaluating $A \wedge B$ instead of $B \wedge A$), implication equivalences (e.g., expressing $A \implies B$ as $\neg A \vee B$), or leveraged spatial symmetry in Cartesian distance functions (e.g., computing the distance from the human to the robot rather than from the robot to the human).

While some of these equivalences could be captured through deterministic rewriting rules, the main challenge lies in identifying semantically equivalent queries that arise from different modeling abstractions. For instance, verifying that an agent completes an action can be expressed either by directly checking some physical parameters or by verifying the completion state of the corresponding action orchestrated by the system. Such formulations are logically equivalent at the specification level but may differ significantly in their syntactic structure.

The semantic evaluation step is therefore critical. Using an LLM-as-a-judge guided by formal equivalence rules, we identified 31 additional queries functionally equivalent to the GT. Its impact is particularly evident in *SC3*: while exact matching yields only 20.6% accuracy, the model often produces valid alternative formulations, such as checking a sequence orchestrator's completion state instead of raw spatial coordinates. Accounting for these equivalents validates 25 additional queries, raising *SC3*'s semantic accuracy to 94.1% and overall accuracy to 77.8%.

RQ3 summary. 95.8% of the LLM-generated verification-ready queries are syntactically correct. Furthermore, semantic evaluation shows that strict string matching significantly underestimates performance: accounting for logical equivalences and alternative formulations increases the effective translation accuracy from 34.7% to 77.8%.

Overall, the results show that the pipeline progressively refines informal specifications into formally verifiable queries: the LLM effectively extracts functional requirements (**RQ1**), reliably filters unverifiable constraints (**RQ2**), and generates syntactically valid and semantically correct UPPAAL queries (**RQ3**).

6 Conclusion

In this paper, we presented a pipeline that bridges informal natural language specifications and formal verification using LLMs. The approach decomposes the process into three stages: (i) extraction of structured atomic requirements, (ii) classification of their formal verifiability with respect to the system model, and (iii) translation of verifiable requirements into UPPAAL SMC queries. Evaluation of three cyber-physical system scenarios shows that the pipeline effectively supports this transformation. The results indicate that LLMs capture the core functional intent of informal specifications, reliably filter unverifiable requirements, and generate syntactically valid and semantically correct formal queries. Notably, semantic evaluation shows that strict string matching substantially underestimates formal query generation performance.

Future work will investigate intrinsic evaluation metrics to automatically assess intermediate outputs and enable refinement loops that improve each pipeline stage while mitigating error propagation. We also plan a broader empirical evaluation with additional LLMs, more experimental repetitions, and a wider range of scenarios and models. Finally, we will explore fine-tuned models for requirement extraction and refine system prompt rules to address recurring LLM failure cases.

Acknowledgments

The authors would like to thank Vincenzo Scotti for the provision of computational resources and expert guidance.

References

[1] Gul Agha and Karl Palmskog. "A survey of statistical model checking". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 28.1 (2018), pp. 1–39.

[2] Rajeev Alur et al. "The benefits of relaxing punctuality". In: *Journal of the ACM (JACM)* 43.1 (1996), pp. 116–146.

[3] Markus Bajones et al. "Results of Field Trials with a Mobile Service Robot for Older Adults in 16 Private Households". In: *ACM Transactions on Human-Robot Interaction (THRI)* 9.2 (2019), pp. 1–27.

[4] Arshad Beg et al. "Leveraging LLMs for Formal Software Requirements–Challenges and Prospects". In: *International Workshop on Artificial Intelligence and fOrmal VERification, Logic, Automata, and sYnthesis*. ceur-ws.org. 2025, pp. 95–105.

[5] Vincent Bertram et al. "Leveraging natural language processing for a consistency checking toolchain of automotive requirements". In: *International Requirements Engineering Conference*. IEEE. 2023, pp. 212–222.

[6] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. 2020.

[7] *Case studies - Service robots*. 2018. URL: https://ifr.org/case-studies/service-robots-case-studies.

[8] Boqi Chen et al. "LLM-based Satisfiability Checking of String Requirements by Consistent Data and Checker Generation". In: *International Requirements Engineering Conference*. 2025, pp. 231–243.

[9] Matthias Cosler et al. “nl2spec: Interactively translating unstructured natural language to temporal logics with large language models”. In: *International Conference on Computer Aided Verification*. Springer. 2023, pp. 383–396.

[10] Alexandre David et al. “Uppaal SMC tutorial”. In: *International journal on software tools for technology transfer* 17.4 (2015), pp. 397–415.

[11] Joao Pascoal Faria et al. “Automatic Generation of Formal Specification and Verification Annotations Using LLMs and Test Oracles”. In: *arXiv preprint arXiv:2601.12845* (2026).

[12] Mohamad Fazelnia et al. “Lessons from the use of natural language inference (nli) in requirements engineering tasks”. In: *International Requirements Engineering Conference*. IEEE. 2024, pp. 103–115.

[13] Alexander Elenga Gärtner and Dietmar Göhlich. “Automated requirement contradiction detection through formal logic and LLMs”. In: *Automated Software Engineering* 31.2 (2024), p. 49.

[14] Dario Guidotti et al. “Translating Requirements in Property Specification Patterns using LLMs”. In: *31st RCRA Workshop*. ceur-ws.org. 2024, pp. 68–80.

[15] Jasmin Jahić and Ashkan Sami. “State of Practice: LLMs in Software Engineering and Software Architecture”. In: *International Conference on Software Architecture Companion*. 2024, pp. 311–318.

[16] Hui Li et al. “Extracting Formal Specifications From Documents Using LLMS for Test Automation”. In: *International Conference on Program Comprehension*. IEEE. 2025, pp. 1–12.

[17] Sachiko Lim et al. “Data-driven requirements elicitation: A systematic literature review”. In: *SN Computer Science* 2.1 (2021), p. 16.

[18] Dipeeka Luitel et al. “Improving requirements completeness: Automated assistance through large language models”. In: *Requirements Engineering* 29.1 (2024), pp. 73–95.

[19] Zhi Ma et al. “Bridging Natural Language and Formal Specification–Automated Translation of Software Requirements to LTL via Hierarchical Semantics Decomposition Using LLMs”. In: *arXiv preprint arXiv:2512.17334* (2025).

[20] Taslim Mahbub et al. “Can GPT-4 aid in detecting ambiguities, inconsistencies, and incompleteness in requirements analysis? A comprehensive case study.” In: *IEEE Access* (2024).

[21] Claudio Menghi et al. “Specification patterns for robotic missions”. In: *IEEE Transactions on Software Engineering* (2019).

[22] Hendrik Meth et al. “The state of the art in automated requirements elicitation”. In: *Information and Software Technology* 55.10 (2013), pp. 1695–1709.

[23] Alberto Tagliaferro et al. “Towards verifiable multi-agent interaction pattern specification”. In: *Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 2024, pp. 122–126.

[24] Alberto Tagliaferro et al. “Verification-Oriented Specification of Multi-agent Interaction Patterns”. In: *Workshop on Agents and Robots for reliable Engineered Autonomy*. Springer. 2024, pp. 38–53.

[25] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 2017, pp. 5998–6008.

[26] Cheng Wen et al. “Enchanting program specification synthesis by large language models using static analysis and program verification”. In: *International Conference on Computer Aided Verification*. Springer. 2024, pp. 302–328.

[27] Yuhuai Wu et al. “Autoformalization with large language models”. In: *Advances in neural information processing systems* 35 (2022), pp. 32353–32368.

[28] Rongjie Yan et al. “Formal consistency checking over specifications in natural languages”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 1677–1682.

[29] Mohammad Amin Zadenoori et al. “Large language models (LLMs) for requirements engineering (RE): A systematic literature review”. In: *arXiv preprint arXiv:2509.11446* (2025).

[30] Mengyan Zhao et al. “NL2CTL: automatic generation of formal requirements specifications via large language models”. In: *International Conference on Formal Engineering Methods*. Springer. 2024, pp. 1–17.