Systems Engineering

WILEY

REGULAR ARTICLE

Transforming Natural Language Requirements to Formalism Using LLMs

Baoluo Meng | Robert Lorch | Kit Siu | Michael Durling | Sarat Chandra Varanasi | Saswata Paul | Abha Moitra

GE Aerospace Research, Niskayuna, New York, USA

Correspondence: Baoluo Meng (baoluo.meng@geaerospace.com)

Received: 11 December 2024 | **Revised:** 29 September 2025 | **Accepted:** 4 November 2025

Keywords: first-order logic | large language models | logical formalism | natural language requirements | temporal logic

ABSTRACT

Logical formalism is the use of mathematical models, logical symbols and operators, and logical reasoning to express ideas precisely and unambiguously. Such precision is needed in systems engineering to avoid misunderstandings and misinterpretations. However, creating logical formalism requires a deep understanding of formal logics, making it inaccessible to individuals without specialized training. The complexity of logical formalism also makes it time-consuming and costly to develop, thus acting as a barrier for practitioners seeking to apply formalism in their domains. To make formalism more approachable, we present, implement, and evaluate a large language model (LLM)–based approach to interactively translate unstructured natural language (NL) requirements to logical formalism. Our translation strategy employs a *NL-based* interface that provides high assurance for translation correctness while still being approachable for nonexperts in formal methods. We demonstrate the effectiveness of our tool through an experimental evaluation of an industry use case—landing gear.

1 | Introduction

Logical formalism, as a practice that emphasizes the use of rigorous mathematical models and precise notations, is useful across a wide range of disciplines and applications. In software and systems engineering, logical formalism and its accompanying formal analysis techniques can assist in early error detection, identifying flaws and inconsistency in the development lifecycle, which can result in significant time savings and cost reduction. It enables thorough verification and validation of systems, ensuring that they meet their intended requirements and perform as expected. However, formal analysis requires a formal representation of the system being analyzed, which is a barrier to organizations that do not employ experts in formal methods. To help make formal methods more approachable, we propose, implement, and evaluate a large language model (LLM)–based approach to assist in the process of generating formalism from natural language (NL). To demonstrate the effectiveness of our approach, we focus on a particular use case: requirements engineering. Requirements engineering is an early phase of system development where system requirements are elicited, captured, and analyzed for defects.

This project is motivated by industrial experience. While there is a growing need to apply formal, rigorous verification, and validation techniques to systems development, especially to safety-critical systems, we experience a lack of enthusiasm for tools that require engineers to manually formalize NL. Formalized requirements lend themselves to precise analyses using theorem provers and satisfiability modulo theory (SMT) solvers [1]. When considering the Aerospace Industry, certifying that requirements are complete and conflict free are obligatory as prescribed by standards such as ARP4754B[1]. Hence, for requirements engineers, communicating to stakeholders about conflicts in requirements

is essential. For example, our requirements analysis tool ASSERT [2–6] performs formal analysis of requirements specified in the SADL ontology language [7]. Although the tool is capable of a wide variety of analyses, it is underutilized due to its reliance on a highly structured input that can be challenging to author for nonformal methods experts. Therefore, we aim to develop an approach that lends itself to a NL interface that is more approachable for all engineers.

Problem and scope. In this paper, we address the following research question: *Is it possible to develop a generic pipeline to reliably translate unrestricted natural language (NL) requirements to logical formalisms?*

Challenges. NL is difficult to formalize because it often carries ambiguity and there are often many different ways of expressing the same idea. Additionally, LLMs present many challenges as follows: (i) they "hallucinate" and generate incorrect information; (ii) they require a lot of computing resources (top-of-the-line GPU clusters) compared to traditional machine-learning or algorithmic models for NL tasks, even for inferencing; (iii) they often have license[2] or data privacy issues relevant to corporate users (if using an externally hosted LLM); and (iv) they require a substantial amount of training data to fine-tune for specific tasks.

Approach. We designed an interactive pipeline that addresses these challenges. We split the translation task into several steps. The core aspects of the pipeline are syntactic abstraction, translation, and grounding, where the LLM performs the translation step on a *lifted* version of the sentence that preserves the underlying sentence structure while abstracting away unnecessary details. To supplement these core steps, we employ two preprocessing steps of (i) identifying wordiness and ambiguity and (ii) entity recognition. Step (i) helps state the sentence as clearly and concisely as possible, while Step (ii) prepares the sentence for syntactic abstraction. Also, we include two postprocessing steps of (iii) translation to an abstract syntax tree (AST) and (iv) LLM self-correction. Step (iii) will detect if the LLM's output is ill-formed, while Step (iv) involves giving the LLM an opportunity to reflect on (and potentially revise) its prior output for higher translation accuracy. All steps of translation are performed using *in-context learning*, which does not require a large training data set. In general, our approach avoids excessive manual effort (as in reinforcement learning with human feedback or manual dataset generation), and we aim to provide an approach that generalizes across domains.

In the development of large systems, the subsystems may be produced by different companies and the requirements serve as a contractual agreement. This requires that the agreed upon requirements be complete, conflict-free and realizable and hence the usefulness of formalizing requirements. Even within a single organization, having complete, conflict-free, and realizable requirements allows for modular, streamlined, and parallel development and testing of subsystems. For these reasons, in this work, we have chosen to focus on requirement formalization instead of automatically generating artifacts from NL requirements (e.g., generating code directly from the requirements).

Paper structure. In Section 2, we provide background information and definitions to introduce the relevant terms necessary for understanding our translation pipeline. Then, in Section 3, we give context to where our work stands relative to current literature. In Section 4, we outline the approach of our translation pipeline. Then we give insight on our implementation in Section 5. The results, discussion, and conclusion and future work are presented in Sections 6–8, respectively.

2 | Background and Definitions

In this section, we discuss requirements engineering and introduce relevant concepts necessary to understand the translation pipeline.

Requirements engineering is an early stage of the system development phase where system requirements are elicited, captured, and analyzed. A common problem is that defects in requirements (e.g., logical inconsistencies between requirements, missing requirements, redundancies, etc.) are not caught until later in system development, and these design errors are extremely expensive and time-consuming to fix retroactively. One way to rigorously analyze requirements is to state them formally and use formal methods tools to perform requirements analysis, but formalizing requirements by hand is difficult, as previously discussed. Our tool can be utilized to interactively formalize a set of requirements as a prerequisite to comprehensive formal analysis. Lorch et al. [8] have conducted a comprehensive survey of formal techniques used in requirements analysis.

Next, some relevant concepts: *In-context learning* involves feeding a task description, instructions, and examples as part of an LLM's input in order to increase the LLM's capability of completing the specified task. Feeding examples is also referred to as *few-shot learning*. It is a specific method of prompt engineering. *Fine-tuning* involves performing additional training on the model with a task-specific dataset (input and output pairs). *Priming* involves giving an LLM a textual description of initial relevant context and knowledge before querying a specific task. *Syntactic abstraction* involves replacing details of a sentence with atomic propositions, leaving only the underlying sentence structure. The resulting abstracted sentence is called a *lifted sentence*. After translating lifted sentences to lifted logical formulas, we *ground* the logical formulas by reintroducing the details that were originally abstracted and tying them to a glossary of core concepts. This process ensures that any two references to the same core concept are mapped to the same atomic proposition or predicate.

The formalisms we consider are extensions and fragments of *first-order logic* (FOL) and *linear temporal logic* (LTL) [9]. FOL is a formal expression that uses variables, quantifiers, and predicates. For example, the NL sentence, "All men are mortal" would be expressed as, "For all X, if X is a man, then X is moral," where X is a variable, "for all" is a quantifier, "is a man" and "is mortal" are predicates. LTL is an extension of FOL and adds time to the logic expressions. For example, when a condition will eventually be true, or a condition will be true until another condition becomes true. Some temporal operators are **G** for always (or globally); **E** for eventually; **X** for next; **U** for until. And finally, an *AST* is a data structure used in computer science to represent the structure of a code snippet, or in our case, a logical formula. The nodes of the tree are operators and the children of the nodes are operands. For

example, the logical expression `a>b` would have > as the root and `a` and `b` as the leaves.

3 | Related Work

In this section, we give context to where our work is situated with respect to the literature. All of the following papers involve NL to formal logic translations.

Rajasekharan et al. [10] illustrate the core idea of using formal methods to assist LLMs with complicated reasoning tasks. However, their work is based on answer set programming, a fundamentally different formalism. He et al. [11] introduce the idea to algorithmically generate training data based on a grammar, where the grammar is informed by a study of real-world signal temporal logic specification data. However, this approach requires a data analysis for the target formalism (which, of course, also assumes such data exists in the first place). The model might also perform poorly for test data outside of the grammar. Chen et al. [12] employ a combination of lifting and fine-tuning using a generated dataset. The dataset is generated by (roughly speaking) prompting an LLM for NL-TL (temporal logic) pairs and then correcting them using reinforcement learning with human feedback. In our work, we avoid manual effort. Fuggitti et al. [13] use a pipeline going from (i) boilerplate pattern to (ii) LLM translation to (iii) postprocessing. The boilerplate input restriction allows naive translation to be fairly accurate. But, we aim to be smarter in the translation phase in order to free the user from input restrictions. Yang et al. [14] propose two main approaches. In the first, a fine-tuning dataset is generated using GPT-4, where the validity of the logical formulas (with respect to the NL text) is verified heuristically. The data are used to fine-tune an LLM for direct translation. Yang et al. [14] also propose an approach where a (relatively small) LLM corrects output from a naive GPT translation, trained using reinforcement learning with human feedback. We aim to avoid manual work. Similar to our approach, Cosler et al. [15] also focus on interactive translation. However, the interactivity is through translations of subformulas that are more readable than the whole formula, which differs from our approach, which provides a purely NL interface to the user. The work of Liu et al. [16] is similar to our work. It involves the starting point for our pipeline, including the lifting, translation, and grounding phases, but it is designed specifically for robotics. However, we contribute by (i) adding robustness on the ends (input and output processing); (ii) introducing interactive aspects to boost translation accuracy; and (iii) expanding the applicability beyond the robotics domain through more sophisticated methods of lifting and grounding. The work of Gärtner and Göhlich [17] detects different types of contradictions after formalizing requirements using LLMs. However, they do not formalize complex LTL-based requirements as handled by our pipeline.

Arvidsson et al. [18] provide prompt engineering guidelines for LLMs in requirements engineering. Arora et al. [19] discuss the role of llms in advancing requirements engineering. Marques et al. [20] and Hemmat et al. [21] conduct comprehensive reviews of using LLMs in software requirements engineering. Wei [22] introduces a tailored LLM for automating the generation of code snippets from well-structured requirements documents. Spoletini et al. [23] discuss the integration of automatic formal requirements engineering techniques as a complement to LLM code generation and explore how LLMs can facilitate the broader acceptance of formal requirements, thus making the vision proposed in the paper realizable. Ronanki et al. [24] introduce prompts and prompting patterns for requirements engineering using generative AI. Norheim et al. [25] discuss challenges in applying LLMs to requirements engineering tasks. Vogelsang and Fischbach [26] provide systematic guidelines in applying LLMs to Requirements Engineering centric NLP tasks. However, they do not provide guidelines for formal methods–based tasks as tackled by our pipeline. Nonetheless, our emphasis on correct curation of ground entities and the interactive nature of our pipeline in case of missing ground entities from human experts is consistent with their recommended use of GPT-like [27] and BERT-like [28] models, as precisely used by our pipeline in Figure 1.

4 | Methodology

The general approach is illustrated in Figure 1. We decompose the pipeline into three core stages: *preprocessing*, *translation*, and *postprocessing*. We will outline each core stage, highlighting novel areas and interactive aspects.

4.1 | Preprocessing

The preprocessing stage is comprised solely of Step ①.

In Step ①, the LLM takes in an NL requirement and *interactively* produces a clear, concise NL representation via conversation with the engineer.

This interactivity can ensure that the semantics of the requirement are not changed. Further, this interactivity is about requirements in NL and thus does not require the engineer to be knowledgeable about LTL formalism. We also believe that future enhancements where the LLM model learns from past translations will be able to reduce the amount of interactivity needed and support scalability of our approach.

Step ① leverages the LLM's chatbot functionality, as the engineer can ask questions about the LLM's output and provide user feedback. We *prime* the LLM to maximize performance by using the following prompt:

> *Question: Pretend that you are a formal methods expert advising an engineer on how to state a system specification concisely and unambiguously. Be sure to omit wordiness and unnecessary details that are not relevant to the system specification, and be sure to not alter the semantics of the specification. In the following sentence, remove any unnecessary wordiness and alert the engineer of any natural-language ambiguities. Be sure to keep any temporal aspects expressed in the specification: <sentence>. Answer:''*

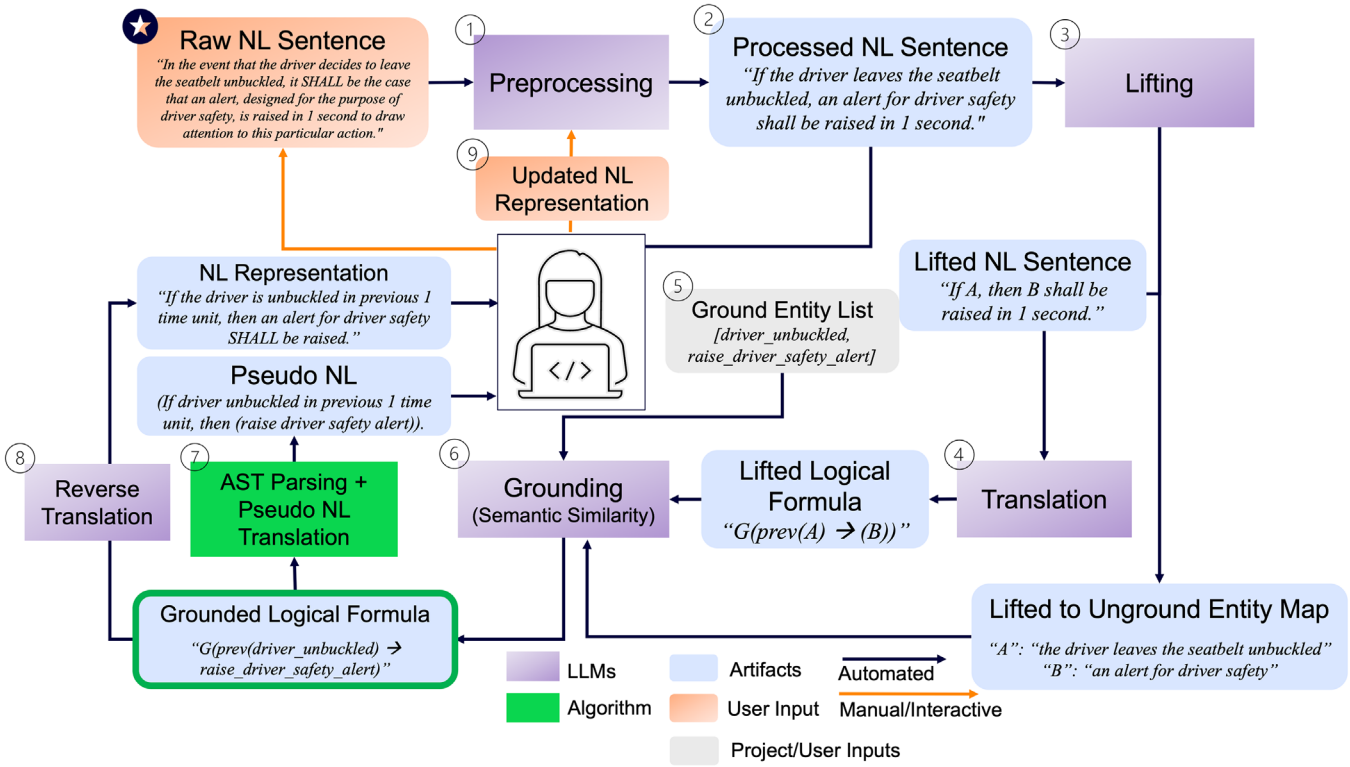

FIGURE 1 | Translation pipeline. Orange ↦ user inputs; gray ↦ project/user inputs (could be from the project glossary); green ↦ output; purple ↦ processing steps; blue ↦ intermediate artifacts. "prev" is a temporal operator in past-time linear temporal logic that refers to the value at the previous 1 time unit (i.e., 1 s in the example).

Then, the engineer converses with the chatbot until the engineer is satisfied with a (potentially) revised version of the input requirement. Interaction is vital to ensure that the semantics of the requirement are unchanged.

In Figure 1, the raw NL sentence reads as follows: "In the event that the driver decides to leave the seatbelt unbuckled, it SHALL be the case that an alert, designed for the purpose of driver safety, is raised in 1 second to draw attention to this particular action." The preprocessed NL sentence is much more concise: "If the driver leaves the seatbelt unbuckled, an alert for driver safety shall be raised in 1 second."

4.2 | Translation

Translation takes place in Steps ③, ④, and ⑥.

Step ③ performs *lifting*, which can also be thought of as *atomic entity recognition*. Here, the LLM takes as input the processed NL from Step ① and abstracts away details of the sentence leaving only the underlying sentence structure. Similar to Step ①, we prime the input with a precise description of the task:

> *Your task is to take English sentences as input and perform syntactic abstraction, producing "lifted" English sentences as output, where some phrases are abstracted away and replaced with atomic entities of the form [Pn\abstracted details]. Start with "P1", then "P2", "P3", and so on. Aspects of the input sentence that should be maintained (not abstracted) include conditional statements, conjunctions, disjunctions, negations, equality, comparisons, temporal aspects, and units.*

Then, we give several examples for *in-context learning* similar to the following:

> *Input: If you are born, then eventually you will die.*
>
> *Output: If [P1\you are born], then eventually [P2\you will die].*

In the above example, text of the form `[Pn\extra details]` means that the extra details are abstracted away and replaced with the atomic entity `Pn`.

Once we have the lifted sentence where details are abstracted and replaced with atomic entities, we perform *translation* to a lifted logical formula in Step ④. The priming description is as follows:

> *Your task is to translate lifted English sentences to linear temporal logic (LTL). The lifted English sentences have some details abstracted away in terms of atomic entities*

of the form [Pn], where "n" is some number. In LTL, only use the operators & (and), | (or),! (not), = (equals), > (greater than), < (less than), Globally, Eventually, Next, Since, and Until. < Description of the semantics of the operators.[3] *> The LTL translation should only include these operators and abstraction entities with no extra English verbiage. Each entity in the input should appear somewhere in the output.*

Continuing with our earlier example, we give the LLM context as so:

Input: If [P1], then eventually [P2].

Output: [P1] -> Eventually [P2].

The translation task is easier for the LLM when operating over lifted sentences with atomic entities rather than full NL sentences with (possibly) complex vocabulary and wordy phrases.

The final aspect of translation is *grounding* performed in Step ⑥. In this step, we replace the lifted atomic entities of the form `Pn` with concrete versions representing real concepts, for example, `born` and `die`. The purpose of grounding is to ensure that multiple references to the same core concept (within or across formulas) are mapped to the same (grounded) atomic entity. For example, consider a situation where we are translating two NL sentences: `If you are born, then eventually you will die`; `If you died, then once you were born`. The two NL phrases `you will die` (from the first sentence) and `you died` (from the second sentence) should be referenced by the same entity `die`. To achieve this, phrase embeddings are retrieved from the LLM for (i) each value in the Lifted to Unground Entity Map of representation produced by Step ③ (which associates each lifted atomic entity to the details that were abstracted away), as well as (ii) each value from input ⑤, the Ground Entity List representing the core concepts of the underlying system. Then, each value from the Lifted to Unground Entity Map produced by Step ③ is associated with the most similar ground entity from input ⑤ by computing the highest cosine similarity between the associated phrase embeddings. The Ground Entity List in Step ⑤ can be generated from a project glossary or provided by engineers manually.

4.3 | Postprocessing

Postprocessing occurs in Steps ⑦ and ⑧, and it involves performing correctness checks on the grounded logical formula from Step ⑥. First, in Step ⑦, the string representation of the grounded logical formula is parsed into an AST representation. If parsing fails, then the formula is syntactically invalid. Rather than immediately failing, the pipeline performs *LLM self-correction* where the LLM is instructed to review its translation and look for mistakes using the following prompt:

The following displays a natural language sentence along with a corresponding linear temporal logic (LTL) translation. However, the LTL translation is syntactically ill-formed. Please review the translation and output an updated translation to LTL.

This gives an opportunity to mend the failed translation, rather than completely starting over. Then, in Step ⑧, the pipeline performs *reverse translation* on the AST representation. This converts the structured formalism back to NL such that the engineer can determine whether the underlying formalism generated by the pipeline matches the engineer's intent without the engineer having to manually process the logical formalism. It should be emphasized that the engineer does not have to work with the formalized requirement but rather with the requirement in NL. Step ⑧ produces a reverse translation of the logical formula generated by LLM, giving a natural and readable translation. We give the following instruction:

Translate the following linear temporal logic formula, expressed in a structured representation, to natural language.

However, just using an LLM for reverse translation does not fully validate the underlying formalism, as there is no guarantee that the LLM's reverse translation leaves the semantics unchanged (which is the very problem we attempt to protect against). So, we also present a rule-based translation in Step ⑦ to a pseudo-NL that includes parentheses denoting scope, which guarantees that the reverse translation semantically match the underlying formula. For example, for the structured formula,

('IMPLIES', ('AND', ('EQUALS', 'command_line', 'normal_mode'), ('EQUALS', 'command_handle', 'down')), ('NOT', 'retraction_sequence')),

the LLM reverse translation gives the following:

If the command line is in normal mode and the command handle is down, then the retraction sequence should not occur.

while the rule-based pseudo-NL translation gives the following:

(if ((command line is normal mode) and (command handle is down)), then (retraction sequence does not happen)).

5 | Implementation

We instantiate our framework in Python using the Mistral LLM v0.3[4] [27] with about 7B parameters for Steps ①, ③, ④, and ⑧ and Sentence-BERT [28] with about 100M parameters for Step ⑥. The LLM processing steps use the `transformers` and `sentence-transformers` libraries, and Step ⑦ uses `ply` for lexing and parsing. All LLM processing steps involve priming, and Steps ③ and ④ involve in-context learning. None of the steps require additional fine-tuning, which is an attractive feature of

our approach because it can be applied to domains that lack training data.

An attractive feature of our implementation is that it is comprised entirely of free and open-source components, and is executable using a single NVIDIA V100 GPU. Thus, it is feasible for corporations with licensing and data privacy restrictions, as well as users with limited computing power, to run our tool locally.

6 | Results

To test the effectiveness of our approach, we perform a case study involving the formalization of some open source landing gears requirements [29]. In the study, we formalize 19 requirements, referencing a total of 47 ground entities. The target formalism is LTL extended with equality and enumeration types. An example requirement is as follows:

> *When the command line is working (normal mode), the stimulation of the gears outgoing or the retraction electro-valves can only happen when the three doors are locked open.*

We evaluate and score the output of the pipeline. We did this only on the fully automated steps (noninteractive portions) of the pipeline because the interactive steps are difficult to objectively evaluate and can have wildly different performance if the Mistral LLM were to be replaced with LLaMA [30] for instance. Consequently, we manually preprocessed the requirement for wordiness and ambiguity. We also fleshed out the details in each requirement based on well-known knowledge about the landing gear system. For example, we manually rewrote the above example by replacing *the three doors* with the notion of a front door, left door, and right door such that it becomes:

> *When the normal mode is true, the gear_extend_EV to ON or gear_retract_EV to ON can be stimulated only when the F_door and L_door and R_door are OPEN.*

We then manually wrote the corresponding logical formula which we use to compare each requirement output from the pipeline. For the example above:

G (**NOT** (gear_extend_EV = ON) **UNTIL** (normal_mode

= TRUE ∧ F_door = OPEN ∧ L_door = OPEN ∧ R_door = OPEN)

∧ **NOT** (gear_retract_EV = ON) **UNTIL** (normal_mode

= TRUE ∧ F_door = OPEN ∧ L_door = OPEN ∧ R_door = OPEN))

For each of the 19 input requirements, we score the quality of the output translation according to the following scale:

(2 out of 2) The translation is perfect.

(1 out of 2) The translation is correct, save for including an incorrect term.

(1 out of 2) The translation is correct, save for an operator precedence error.

(1 out of 2) The translation is correct, save for a single incorrect operator (e.g., a missing temporal operator, or an **E** in place of an **X**.)

(0 out of 2) The translation is incorrect, and is not close to a correct translation.

An example of a perfect score is the output translation for the requirement from above. The outputs from each step of the pipeline are as follows. Note that to save space, we did not print **G** in the LTL translation as it is always present.

```
***** LIFTING ***** - Generate lifted
sentence:
If [P1] = [P2], [P3] to [P4] or [P5] to [P6]
can be stimulated
only when [P7] = [P8] and [P9] = [P8] and
[P10] = [P8].

Lifted to unground props map:
{'P1': 'normal mode', 'P2': 'true', 'P3':
'gear_extend_EV', 'P4': 'ON', 'P5':
'gear_retract_EV',
'P6': 'ON', 'P7': 'F_door', 'P8': 'open',
'P9': 'L_door', 'P10': 'R_door'}

***** LIFTING TO LTL ***** - Translation
of lifted sentence to lifted LTL:
Not ([P3] = [P4]) Until ([P1] = [P2] and
[P7] = [P8] and [P9] = [P8] and [P10] = [P8])
and Not ([P5] = [P6]) Until ([P1] = [P2] and
[P7] = [P8] and [P9] = [P8] and [P10] =
[P8]).

***** GROUNDING ***** - Translation of
lifted LTL to grounded LTL:
Not ([gear_extend_EV] = [ON]) Until ([output_
normal_mode] = [true]
and [F_door] = [OPEN] and [L_door] = [OPEN]
and [R_door] = [OPEN])
and Not ([gear_retract_EV] = [ON]) Until
([output_normal_mode] = [true]
and [F_door] = [OPEN] and [L_door] = [OPEN]
and [R_door] = [OPEN]).
```

Here is another example, this time of an output translation that we score as imperfect. This requirement describes a failure mode and contains a specific number of seconds:

> *If one of the three doors is still seen locked in the closed position more than 7 seconds after stimulating the opening electro-valve, then the boolean output normal mode is set to false.*

Just as before, we did a manual preprocessing step on the original NL requirement text to make the sentence more concise, less ambiguous, and more concrete. We again enumerated *the three doors* in the text. The input and pipeline step outputs are as follows:

```
***** INPUT ***** - Input sentence:
If F_door or L_door or R_door is CLOSED for
more than 7 seconds after the opening_EV
is ON,
then the boolean output normal mode is
set to false.

***** LIFTING ***** - Generate lifted
sentence:
If [P1] or [P2] or [P3] is [P4] for more than
7 seconds after [P5] is [P6],
then [P7] is set to [P8].

Lifted to unground props map:
{'P1': 'F_door', 'P2': 'L_door', 'P3':
'R_door', 'P4': 'CLOSED', 'P5': 'opening_EV',
'P6': 'ON',
'P7': 'boolean output normal mode', 'P8':
'false'}
```

This is an example of a real-time requirement. We chose 1 second as the timing unit, which is a common practice to do when formalizing requirements that involve real-time. Here's where the output of the pipeline was less than perfect. We tried to teach the LLM through prompting with examples on how to use the NEXT operator in the LTL translation. The example shows that "*more than N seconds*" should be translated to print the NEXT operator $N + 1$ times.

> *Input: If [P1] for more than 3 seconds after [P2], then [P3].*
>
> *Output: ([P2] Release Not [P1]) and (NEXT [P1] and NEXT NEXT [P1] and NEXT NEXT NEXT [P1] and NEXT NEXT NEXT NEXT [P1]) -> [P3].*

While LLMs can sometimes approximate counting, they are not inherently equipped for precise symbolic reasoning due to their design as statistical sequence models. Even when we prompt the LLM with an additional example to show an $M + 1$ case, where $N \neq M$, it is not able to learn the pattern. Since the rest of the translation was correct, we give this output a score of 1.

```
***** LIFTING TO LTL ***** - Translation of
lifted sentence to lifted LTL:
(([P5] = [P6] Release Not ([P1] = [P4] or
[P2] = [P4] or [P3] = [P4]))
and (NEXT ([P1] = [P4] or [P2] = [P4] or
[P3] = [P4])
and NEXT NEXT ([P1] = [P4] or [P2] = [P4] or
[P3] = [P4])
and NEXT NEXT NEXT ([P1] = [P4] or [P2] =
[P4] or [P3] = [P4])
and NEXT NEXT NEXT NEXT ([P1] = [P4] or
[P2] = [P4] or [P3] = [P4])
and NEXT NEXT NEXT NEXT NEXT ([P1] =
[P4] or [P2] = [P4] or [P3] = [P4])
and NEXT NEXT NEXT NEXT NEXT NEXT NEXT
([P1] = [P4] or [P2] = [P4] or [P3] = [P4])
and NEXT NEXT NEXT NEXT NEXT NEXT NEXT
([P1] = [P4] or [P2] = [P4] or [P3] = [P4])))
-> [P7] = [P8].

***** GROUNDING ***** - Translation of lifted
LTL to grounded LTL:
(([opening_EV] = [ON] Release Not ([F_door] =
[CLOSED] or [L_door] = [CLOSED] or [R_door] =
[CLOSED]))
and (NEXT ([F_door] = [CLOSED] or [L_door] =
[CLOSED] or [R_door] = [CLOSED])
and NEXT NEXT ([F_door] = [CLOSED] or
[L_door] = [CLOSED] or [R_door] = [CLOSED])
and NEXT NEXT NEXT ([F_door] = [CLOSED] or
[L_door] = [CLOSED] or [R_door] = [CLOSED])
and NEXT NEXT NEXT NEXT ([F_door] = [CLOSED]
or [L_door] = [CLOSED] or ...)
and NEXT NEXT NEXT NEXT NEXT ([F_door] =
[CLOSED] or [L_door] = [CLOSED] or ...)
and NEXT NEXT NEXT NEXT NEXT NEXT ([F_door] =
[CLOSED] or [L_door] = [CLOSED] or ...)
and NEXT NEXT NEXT NEXT NEXT NEXT NEXT NEXT
([F_door] = [CLOSED] or [L_door] = [CLOSED]
or ...)
and NEXT NEXT NEXT NEXT NEXT NEXT NEXT NEXT
([F_door] = [CLOSED] or [L_door] = [CLOSED]
or ...)))
-> [output_normal_mode] = [false].
```

Finally, here is an example of a requirement that is translated incorrectly. This requirement describes a failure mode of the landing gear system and references the command handle position *before* and *after* a specific period of time.

> *When at least one computing module is working, if the landing gear command handle has been down for 15 seconds, and if the gears are not locked down after 15 seconds, then the red light "landing gear system failure" is on.*

For this requirement, the Mistral LLM we use in our pipeline is unable to generate the required number of NEXT operators, as noted in the previous example. But more than that, it was unable to follow prompts that explained 15 steps must be generated for "past 15 seconds" and 15 more steps for "after 15 seconds."

Of the 19 input requirements, nine received a score of 2; eight received a score of 1; and two received a score of 0. All eight that received a score of 1 were real-time requirements with the phrase "more than N seconds." The two requirements that received a score of 0 contained real-time specifications on how long a state "has been" set *and* how long a state is maintained "after." Finally, we did not see any incorrect term errors in the translation.

7 | Discussion

Based on our experience using the tool, we identify two limitations.

First, NL sentences sometimes reference higher-level propositions. For example, in the landing gears requirements, there are three doors and three gears, a set in each of these positions—front, left, and right. However, a requirement may mention the status of "one of the three doors." One way to translate this requirement is to introduce a new proposition—`one_of_three_doors_status`. However, this requires extra constraints to be imposed in the requirements analysis phase, that is, tying the value of `one_of_three_doors_status` to `front_door_status`, `left_door_status`, and `right_door_status`. An alternative approach to avoid introducing an additional proposition is to reword the requirement, which we have demonstrated in our manual preprocessing step, or split it into multiple requirements referencing each of the doors explicitly.

Second, it is important to note that the person who conducted this case study is familiar with formal logic and LTL, which impacted the requirements preprocessing stage. Users who are not familiar with formal logic will have less intuition regarding which requirements will be difficult for the machine to parse and translate into LTL, and how to word the requirements clearly such that a translation is feasible. We argue that the tool still saves time and manual effort, but it is not a panacea that allows complete ignorance of the underlying formalism.

Overall, requirements engineering is an important component of designing industrial, safety-critical systems. This importance is evident in the available resources—INCOSE's Guide to Writing Requirements [31]; International Requirements Engineering Conference [32] that has been held annually since 1993; safety-critical software standards such as DO-178C [33] which puts great emphasis on requirements (i.e., requirements-based tests, requirements traceability, requirements management plan). In particular, for software with high-risk classification, meeting necessary DO-178C objectives involves showing that requirements are accurate, consistent, and conform to standards. In safety-critical systems, ambiguity in requirements can lead to severe consequences including system failure and safety hazards. Our pipeline works towards reducing these risks by ensuring that requirements are clear and precise. We are aware that scalability to an industrial size problem will be important. At this point, we have tested our tool based on real, industrial *type* requirements (beyond what's demonstrated in this paper); scaling to industrial *size* is on our horizon. However, we are confident that LLMs have significant potential in understanding and processing text, making them valuable for automating the extraction of requirements from industrial size documents. LLMs can process a vast amount of text data efficiently, given sufficient computing resources which are available in industry.

8 | Conclusion and Future Work

In this paper, we have proposed an interactive pipeline to translate NL sentences to formal logic with a usable interface that allows the user to communicate with the tool solely in NL. We implemented a prototype version of the tool in Python with the (relatively) small, completely open source LLMs: Mistral [27] and Sentence-BERT [28]. In our experimental evaluation over a requirements engineering case study with open-source landing gear requirements, we demonstrate that our implementation performs well, even though it is using LLMs that lag well behind larger, state-of-the-art models (e.g., GPT4 [34], Claude 2 [35]). Users without licensing or computational restrictions can easily swap in more capable LLMs to boost performance.

We leveraged LLMs to facilitate the translation of NL requirements into their intended formalization in stages. Although the application of LLM-aided formalization of potentially ambiguous NL input compounded by the fact that LLMs can hallucinate may seem counterproductive, our application of LLMs particularly at the lifting and translation stages of the pipeline have produced promising results and their outputs will always be checked downstream by humans-in-the-loop.

We also emphasize that while bigger LLMs such as reasoning models can improve the accuracy of the pipeline, but due to their inadequacies in solving complicated planning problems [36], we will still rely on human-in-the-loop validation of formalized requirements and use mature automated reasoning tools [2] for detecting conflicts and completeness errors in formalized requirements. In summary, we use LLMs as an aid to accelerate the formalization of NL requirements in stages rather than using them for end-to-end formalization and reasoning.

We identify three promising directions for future work. First, the approach could be extended to handle *requirement document formalization* (rather than formalization of individual requirement sentences), which would require partitioning the input document into a set of separate logical concepts. Second, we could make the pipeline more usable by using the LLM to automatically generate an initial list of ground entities, minimizing one of the required, manual inputs from the user. Third, the approach could be enhanced to remember and learn from past translations in a particular project or domain such that translation accuracy improves over the course of the project.

Acknowledgments

Distribution Statement "A" (approved for public release, distribution unlimited). This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Endnotes

[1] ARP4754B Standard: https://www.sae.org/standards/content/arp4754b

[2] LLaMA-3 use policy: https://www.llama.com/llama3/use-policy/

[3] For the sake of space, the description of semantics of the operators is omitted.

[4] Mistral 7B v0.3 model: https://huggingface.co/mistralai/Mistral-7B-v0.3, last updated in July 2024 on HuggingFace.

References

1. S. C. Varanasi, B. Meng, R. Lorch, et al., "Trace: Toolkit for Requirements Analysis, Capture, and Elicitation," in *NASA Formal Methods Symposium* (Springer, 2025), 380–399.

2. A. Crapo, A. Moitra, C. McMillan, and D. Russell, "Requirements Capture and Analysis in ASSERT™," in *2017 IEEE 25th International Requirements Engineering Conference (RE)* (IEEE, 2017), 283–291.

3. A. W. Crapo and A. Moitra, "Using OWL Ontologies as a Domain-Specific Language for Capturing Requirements for Formal Analysis and Test Case Generation," in *2019 IEEE 13th International Conference on Semantic Computing (ICSC)* (IEEE, 2019), 361–366.

4. M. Li, B. Meng, H. Yu, et al., "Requirements-Based Automated Test Generation for Safety Critical Software," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)* (IEEE, 2019), 1–10.

5. A. Moitra, K. Siu, A. W. Crapo, et al., "Automating Requirements Analysis and Test Case Generation," *Requirements Engineering* 24 (2019): 341–364.

6. K. Siu, A. Moitra, M. Durling, et al., "Flight Critical Software and Systems Development Using ASSERT™," in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)* (IEEE, 2017), 1–10.

7. A. Crapo and A. Moitra, "Toward a Unified English-Like Representation of Semantic Models, Data, and Graph Patterns for Subject Matter Experts," *International Journal of Semantic Computing* 7, no. 3 (2013): 215–236.

8. R. Lorch, B. Meng, K. Siu, et al., "Formal Methods in Requirements Engineering: Survey and Future Directions," in *Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE)* (2024), 88–99.

9. M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems* (Cambridge University Press, 2004).

10. A. Rajasekharan, Y. Zeng, P. Padalkar, and G. Gupta, "Reliable Natural Language Understanding With Large Language Models and Answer Set Programming," preprint, arXiv:2302.03780, 2023.

11. J. He, E. Bartocci, D. Ničković, H. Isakovic, and R. Grosu, "DeepSTL: From English Requirements to Signal Temporal Logic," in *Proceedings of the 44th International Conference on Software Engineering* (2022), 610–622.

12. Y. Chen, R. Gandhi, Y. Zhang, and C. Fan, "Nl2TL: Transforming Natural Languages to Temporal Logics Using Large Language Models," preprint, arXiv:2305.07766, 2023.

13. F. Fuggitti and T. Chakraborti, "NL2LTL–a Python Package for Converting Natural Language (NL) Instructions to Linear Temporal Logic (LTL) Formulas," in *AAAI Conference on Artificial Intelligence* (2023).

14. Y. Yang, S. Xiong, A. Payani, E. Shareghi, and F. Fekri, "Harnessing the Power of Large Language Models for Natural Language to First-Order Logic Translation," preprint, arXiv:2305.15541, 2023.

15. M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics With Large Language Models," preprint, arXiv:2303.04864, 2023.

16. J. X. Liu, Z. Yang, B. Schornstein, et al., "Lang2LTL: Translating Natural Language Commands to Temporal Specification With Large Language Models," in *Workshop on Language and Robotics at CoRL 2022* (2022).

17. A. E. Gärtner and D. Göhlich, "Automated Requirement Contradiction Detection Through Formal Logic and LLMs," *Automated Software Engineering* 31, no. 2 (2024): 49, https://doi.org/10.1007/s10515-024-00452-x.

18. K. Ronanki, S. Arvidsson, and J. Axell, "Prompt Engineering Guidelines for Using Large Language Models in Requirements Engineering," in *Euromicro Conference on Software Engineering and Advanced Applications* (Springer, 2023), 245–262.

19. C. Arora, J. Grundy, and M. Abdelrazek, "Advancing Requirements Engineering Through Generative AI: Assessing the Role of LLMs," in *Generative AI for Effective Software Development* (Springer, 2024), 129–148.

20. N. Marques, R. R. Silva, and J. Bernardino, "Using Chatgpt in Software Requirements Engineering: A Comprehensive Review," *Future Internet* 16, no. 6 (2024): 180.

21. A. Hemmat, M. Sharbaf, S. Kolahdouz-Rahimi, K. Lano, and S. Y. Tehrani, "Research Directions for Using LLM in Software Requirement Engineering: A Systematic Review," *Frontiers in Computer Science* 7 (2025): 1519437.

22. B. Wei, "Requirements Are All You Need: From Requirements to Code With LLMs," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)* (IEEE, 2024), 416–422.

23. P. Spoletini and A. Ferrari, "The Return of Formal Requirements Engineering in the Era of Large Language Models," in *International Working Conference on Requirements Engineering: Foundation for Software Quality* (Springer, 2024), 344–353.

24. K. Ronanki, B. Cabrero-Daniel, J. Horkoff, and C. Berger, "Requirements Engineering Using Generative AI: Prompts and Prompting Patterns," in *Generative AI for Effective Software Development* (Springer, 2024), 109–127.

25. J. J. Norheim, E. Rebentisch, D. Xiao, L. Draeger, A. Kerbrat, and O. L. de Weck, "Challenges in Applying Large Language Models to Requirements Engineering Tasks," *Design Science* 10 (2024): e16.

26. A. Vogelsang and J. Fischbach, "Using Large Language Models for Natural Language Processing Tasks in Requirements Engineering: A Systematic Guideline," in *Handbook on Natural Language Processing for Requirements Engineering* (Springer, 2025), 435–456.

27. Mistral AI, "Mistral 7b, the Best 7b Model to Date, Apache 2.0," September 2023, https://mistral.ai/news/announcing-mistral-7b/.

28. N. Reimers and I. Gurevych, "Sentence-Bert: Sentence Embeddings Using Siamese Bert-Networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing* (Association for Computational Linguistics, 2019), http://arxiv.org/abs/1908.10084.

29. F. Boniol and V. Wiels, "The Landing Gear System Case Study," in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z* (Springer, 2014), 1–18.

30. H. Touvron, L. Martin, K. Stone, et al., "Llama 2: Open Foundation and Fine-Tuned Chat Models," preprint, arXiv:2307.09288, 2023.

31. International Council on Systems Engineering Requirements Working Group, "Guide to Writing Requirements," 2023, https://www.incose.org/docs/default-source/working-groups/requirements-wg/gtwr/incose_rwg_gtwr_v4_040423_final_drafts.pdf?sfvrsn=5c877fc7_2.

32. IEEE, "Requirements Engneering Conference," 2025, https://conf.researchr.org/home/RE-2025.

33. Radio Technical Commission for Aeronautics (RTCA), *DO-178C: Software Considerations in Airborne Systems and Equipment Certification* (2012), https://store.accuristech.com/standards/rtca-do-178c?product_id=2200105.

34. OpenAI, "Chatgpt," 2023, https://chat.openai.com/chat.

35. Anthropic, "Claude AI," 2023, https://claude.ai.

36. P. Shojaee, I. Mirzadeh, K. Alizadeh, M. Horton, S. Bengio, and M. Farajtabar, "The Illusion of Thinking: Understanding the Strengths and Limitations of Reasoning Models via the Lens of Problem Complexity," preprint, arXiv:2506.06941, 2025.